DIGITAL NOTES
ON
Data Structures
(R22A0503)

B.TECH II YEAR–I-SEM
(R22)(2023-2024)

Department of Computer Science and Engineering

MALLAREDDYCOLLEGEOFENGINEERING&TECHNOLOGY

(Autonomous Institution– UGC, Govt. of India)

Recognizedunder2(f)and12(B)ofUGCACT 1956

(AffiliatedtoJNTUH,Hyderabad,ApprovedbyAICTE-AccreditedbyNBA&NAAC –'A'Grade-ISO9001:2015Certified)

Maisammaguda,Dhulapally(PostVia.Hakimpet),Secunderabad–500100,TelanganaState,India

SYLLABUS

MALLAREDDY COLLEGE OF ENGINEERING ANDTECHNOLOGY

II Year B.Tech CSE-I SEM

L   T/P/DC

3   -/ -/ -3

(R22A0503)DATA STRUCTURES

COURSE OBJECTIVES:

This course will enable students to
1. Implement Object Oriented Programming concepts in Python.
2. Understand Lists, Dictionaries and Regular expressions in Python.
3. Understanding how searching and sorting is performed in Python.
4. Understanding how linear and non-linear data structures works.
5. To learn the fundamentals of writing Python scripts.

UNIT – I
Oops Concepts- class, object, constructors, types of variables, types of methods. **Inheritance:** single, multiple, multi-level, hierarchical, hybrid, **Polymorphism:** with functions and objects, with class methods, with inheritance, **Abstraction:** abstract classes.

UNIT – II
Searching- Linear Search and Binary Search.
Sorting - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort.

UNIT -III
Data Structures –Definition, Linear Data Structures, Non-Linear Data Structures,
Stacks - Overview of Stack, Implementation of Stack (List), Applications of Stack

Queues: Overview of Queue, Implementation of Queue (List), Applications of Queues, Priority Queues
Linked Lists – Implementation of Singly Linked Lists, Doubly Linked Lists, Circular Linked Lists.   Implementation of Stack and Queue using Linked list.

UNIT -IV
Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching.

Graphs - Introduction, Directed vs Undirected Graphs, Weighted vs Unweighted Graphs, Representations,   Breadth First Search, Depth First Search.

UNIT -V

Trees - Overview of Trees, Tree Terminology, Binary Trees: Introduction, Implementation, Applications. Tree   Traversals, Binary Search Trees: Introduction, Implementation, AVL Trees: Introduction, Rotations, Implementation B-Trees and B+ Trees.

TEXTBOOKS:
1. Data structures and algorithms in python by Michael T. Goodrich
2. Data Structures and Algorithmic Thinking with Python by Narasimha Karumanchi

REFERENCE BOOKS:
1. Hands-On Data Structures and Algorithms with Python: Write complex and powerful code using the latestfeatures of Python 3.7, 2nd Edition by Dr. Basant Agarwal, Benjamin Baka.
2. Data Structures and Algorithms with Python by Kent D. Lee and Steve Hubbard.
3. Problem Solving with Algorithms and Data Structures Using Python by Bradley N Miller and David L.Ranum.
4. Core Python Programming -Second Edition,R. Nageswara Rao, Dreamtech Press

COURSE OUTCOMES:
The students should be able to:
1. Examine Python syntax and semantics and apply Python flow control and functions.
2. Create, run and manipulate Python Programs using core data structures like Lists,
3. Apply Dictionaries and use Regular Expressions.
4. Interpret the concepts of Object-Oriented Programming as used in Python.
5. Master object-oriented programming to create an entire python project using objects and classes

<h1 style="text-align:center">UNIT – I</h1>

Oops Concepts- class, object, constructors, types of variables, types of methods. **Inheritance:** single, multiple, multi-level, hierarchical, hybrid, **Polymorphism:** with functions and objects, with class methods, with inheritance, **Abstraction:** abstract classes.

OOPs in Python

OOPs in Python is a programming approach that focuses on using objects and classes as same as other general programming languages. The objects can be any real-world entities. Python allows developers to develop applications using the OOPs approach with the major focus on code reusability.

Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

```
class Parrot:

    pass
```

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Example:
```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init_(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class_.species))
print("Woo is also a {}".format(woo._class_.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```
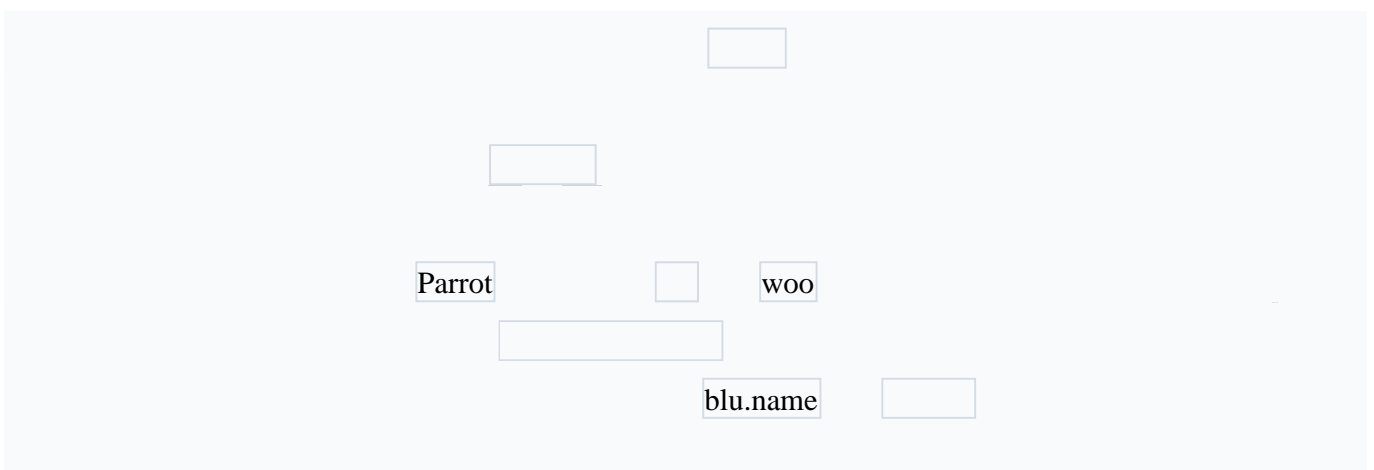
Output

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```
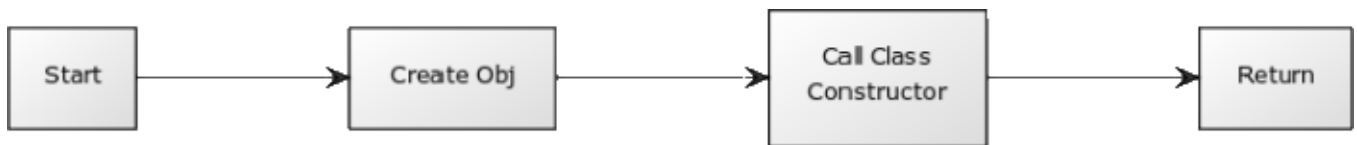
constructor

The constructor is a method that is called when an object is created. This method is defined in the class and can be used to initialize basic variables.

If you create four objects, the class constructor is called four times. Every class has a constructor, but its not required to explicitly define it.

Example:

Each time an object is created a method is called. That methods is named the **constructor**.

The constructor is created with the function **init**. As parameter we write the self keyword, which refers to itself (the object). The process visually is:

Inside the constructor we initialize two variables: legs and arms. Sometimes variables are named properties in the context of object oriented programming. We create one object (bob) and just by creating it, its variables are initialized.

```
classHuman:
def__init__(self):
    self.legs = 2
    self.arms = 2

bob = Human()
print(bob.legs)
```

The newly created object now has the variables set, without you having to define them manually. You could create tens or hundreds of objects without having to set the values each time.

python __init__

The function **init**(self) builds your object. Its not just variables you can set here, you can call class methods too. Everything you need to initialize the object(s).

Lets say you have a class Plane, which upon creation should start flying. There are many steps involved in taking off: accelerating, changing flaps, closing the wheels and so on.

The default actions can be defined in methods. These methods can be called in the constructor.

```python
classPlane:
def__init_(self):
    self.wings = 2

# fly
    self.drive()
    self.flaps()
    self.wheels()

defdrive(self):
        print('Accelerating')

defflaps(self):
        print('Changing flaps')

defwheels(self):
        print('Closing wheels')

ba = Plane()
```

To summarize: A constructor is called if you create an object. In the constructor you can set variables and call methods.

Default value

The constructor of a class is unique: initiating objects from different classes will call different constructors.

Default values of newly created objects can be set in the constructor.

The example belwo shows two classes with constructors. Then two objects are created but different constructors are called.

```python
classBug:
def__init_(self):
    self.wings = 4

classHuman:
def__init_(self):
    self.legs = 2
    self.arms = 2

bob = Human()
tom = Bug()

print(tom.wings)
print(bob.arms)
```

But creating multiple objects from one class, will call the same constructor.

Python Variable Types: Local & Global

There are two types of variables in Python, Global variable and Local variable. When you want to use the same variable for rest of your program or module you declare it as a global variable, while if you want to use the variable in a specific function or method, you use a local variable while Python variable declaration.

Let's understand this Python variable types with the difference between local and global variables in the below program.

1. Let us define variable in Python where the variable "f" is **global** in scope and is assigned value 101 which is printed in output
2. Variable f is again declared in function and assumes **local** scope. It is assigned value "I am learning Python." which is printed out as an output. This Python declare variable is different from the global variable "f" defined earlier
3. Once the function call is over, the local variable f is destroyed. At line 12, when we again, print the value of "f" is it displays the value of global variable f=101

```python
# Declare a variable and initialize it
f = 101
print(f)

# Global vs. local variables in functions
def someFunction():
# global f
    f = 'I am learning Python'
    print(f)

someFunction()
print(f)
```
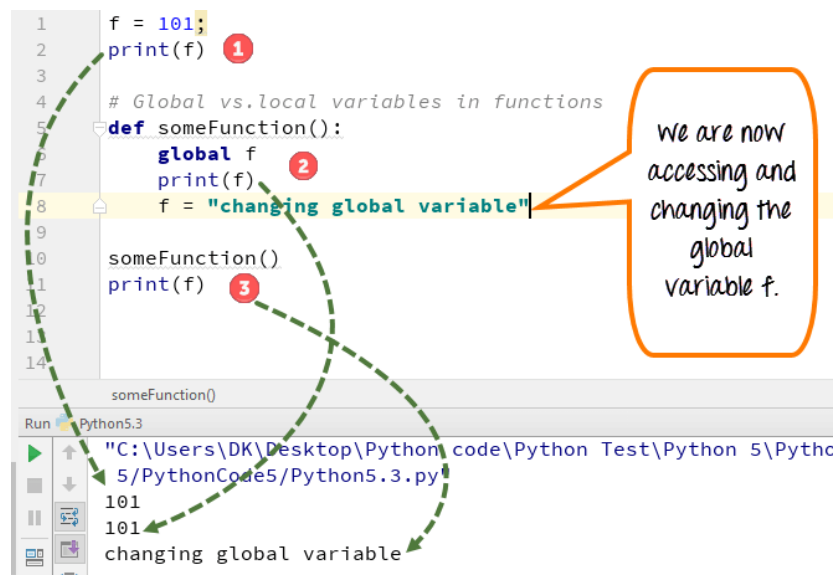
```
"C:\Users\DK\Desktop\Python code\Python Test\Python 5\PythonCode
5/PythonCode5/Python5.2.py"
101
I am learning Python
101
```

> f is a local variable declared inside the function.

Python 2 Example

```
# Declare a variable and initialize it
f = 101
print f
# Global vs. local variables in functions
def someFunction():
# global f
    f = 'I am learning Python'
    print f
```

```
someFunction()
print f
```

Python 3 Example

```python
# Declare a variable and initialize it
f = 101
print(f)
# Global vs. local variables in functions
def someFunction():
# global f
  f = 'I am learning Python'
  print(f)
someFunction()
print(f)
```

While Python variable declaration using the keyword **global,** you can reference the global variable inside a function.

1. Variable "f" is **global** in scope and is assigned value 101 which is printed in output
2. Variable f is declared using the keyword **global**. This is **NOT** a **local variable**, but the same global variable declared earlier. Hence when we print its value, the output is 101
3. We changed the value of "f" inside the function. Once the function call is over, the changed value of the variable "f" persists. At line 12, when we again, print the value of "f" is it displays the value "changing global variable"

Python 2 Example

```python
f = 101;
print f
# Global vs.local variables in functions
def someFunction():
```

```
 global f
 print f
 f = "changing global variable"
someFunction()
print f
```

Python 3 Example

```
f = 101;
print(f)
# Global vs.local variables in functions
def someFunction():
 global f
 print(f)
 f = "changing global variable"
someFunction()
print(f)
```

Types of methods:
Generally, there are three types of methods in Python:

1. Instance Methods.
2. Class Methods
3. Static Methods

Before moving on with the topic, we have to know some key concepts.

Class Variable: A class variable is nothing but a variable that is defined outside the constructor. A class variable is also called as a **static variable**.

Accessor(Getters): If you want to fetch the value from an instance variable we call them accessors.

Mutator(Setters): If you want to modify the value we call them mutators.

1. Instance Method

This is a very basic and easy method that we use regularly when we create classes in python. If we want to print an instance variable or instance method we must create an object of that required class.

If we are using self as a function parameter or in front of a variable, that is nothing but the calling instance itself.

As we are working with instance variables we use self keyword.

Note: Instance variables are used with instance methods.

Look at the code below

```python
# Instance Method Example in Python
class Student:

    def __init_(self, a, b):
        self.a = a
        self.b = b

    def avg(self):
        return(self.a + self.b)/2

s1 = Student(10,20)
print( s1.avg())
```
Copy

Output:

```
15.0
```

In the above program, a and b are instance variables and these get initialized when we create an object for the Student class. If we want to call avg() function which is an instance method, we must create an object for the class.

If we clearly look at the program, the self keyword is used so that we can easily say that those are instance variables and methods.

2. Class Method

classsmethod() function returns a class method as output for the given function.

Here is the syntax for it:

```
classmethod(function)
```

The classmethod() method takes only a function as an input parameter and converts that into a class method.

There are two ways to create class methods in python:

1. Using classmethod(function)
2. Using @classmethod annotation

A class method can be called either using the class (such as C.f()) or using an instance (such as C().f()). The instance is ignored except for its class. If a class method is called from a derived class, the derived class object is passed as the implied first argument.

As we are working with ClassMethod we use the cls keyword. Class variables are used with class methods.

Look at the code below.

```python
# Class Method Implementation in python

class Student:

    name ='Student'

def__init_(self, a, b):

        self.a = a

        self.b = b


    @classmethod

    def info(cls):

    return cls.name



print(Student.info())
```
Copy

Output:

```
Student
```

In the above example, name is a class variable. If we want to create a class method we must use @classmethod decorator and cls as a parameter for that function.

3. Static Method

A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.

For example, If you want to print factorial of a number then we don't need to use class variables or instance variables to print the factorial of a number. We just simply pass a number to the static method that we have created and it returns the factorial.

Look at the below code

```python
# Static Method Implementation in python

class Student:

    name ='Student'

def __init_(self, a, b):

    self.a = a

    self.b = b


@staticmethod

def info():

return "This is a student class"


print(Student.info())
```

Copy
Output

```
This a student class
```

Types of inheritances:

The inheritance is a very useful and powerful concept of object-oriented programming. Using the inheritance concept, we can use the existing features of one class in another class.

The inheritance is the process of acquiring the properties of one class to another class.

In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.

The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.

The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass**.

In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

There are five types of inheritances, and they are as follows.

- **Simple Inheritance (or) Single Inheritance**
- **Multiple Inheritance**
- **Multi-Level Inheritance**
- **Hierarchical Inheritance**
- **Hybrid Inheritance**

The following picture illustrates how various inheritances are implemented.

Simple Inheritance
ParentClass
ChildClass

Multiple Inheritance
ParentClass_1   ParentClass_2
ChildClass

Multi Level Inheritance
ParentClass
ChildClass_1
ChildClass_2

Hierarchical Inheritance
ParentClass
ChildClass_1   ChildClass_2

Hybrid Inheritance
ParentClass
ChildClass_1   ChildClass_2
ChildClass_3
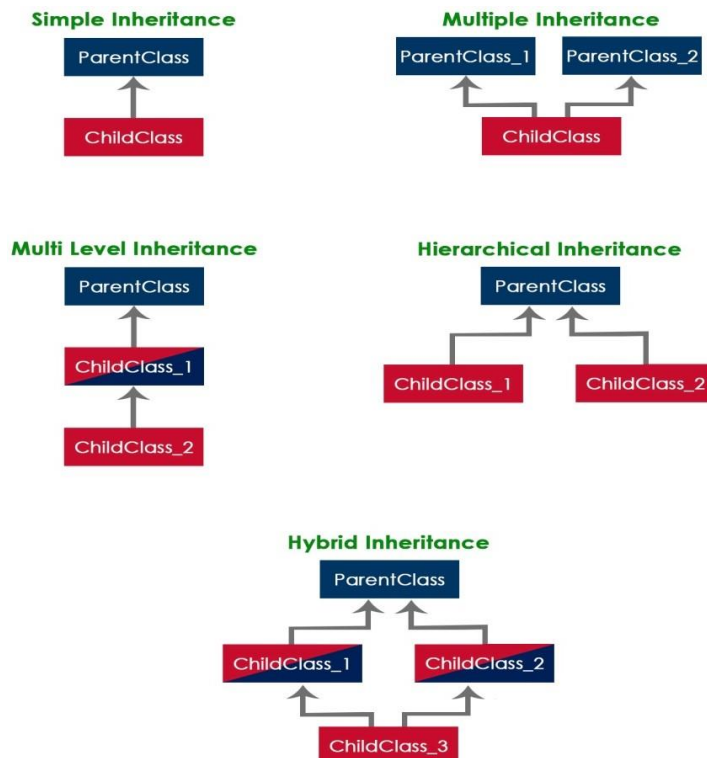
Creating a Child Class

In Python, we use the following general structure to create a child class from a parent class.

Syntax

```
classChildClassName(ParentClassName):
   ChildClass implementation
.
.
```

Let's look at individual inheritance type with an example.

Simple Inheritance (or) Single Inheritance

In this type of inheritance, one child class derives from one parent class. Look at the following example code.

Example

```
classParentClass:

deffeature_1(self):
print('feature_1 from ParentClass is running...')

deffeature_2(self):
print('feature_2 from ParentClass is running...')


classChildClass(ParentClass):

deffeature_3(self):
print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
```

obj.feature_3()

When we run the above example code, it produces the following output.

Multiple Inheritance

In this type of inheritance, one child class derives from two or more parent classes. Look at the following example code.

Example

```
classParentClass_1:

deffeature_1(self):
print('feature_1 from ParentClass_1 is running...')


classParentClass_2:

deffeature_2(self):
print('feature_2 from ParentClass_2 is running...')
```

```python
classChildClass(ParentClass_1, ParentClass_2):

deffeature_3(self):
print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```
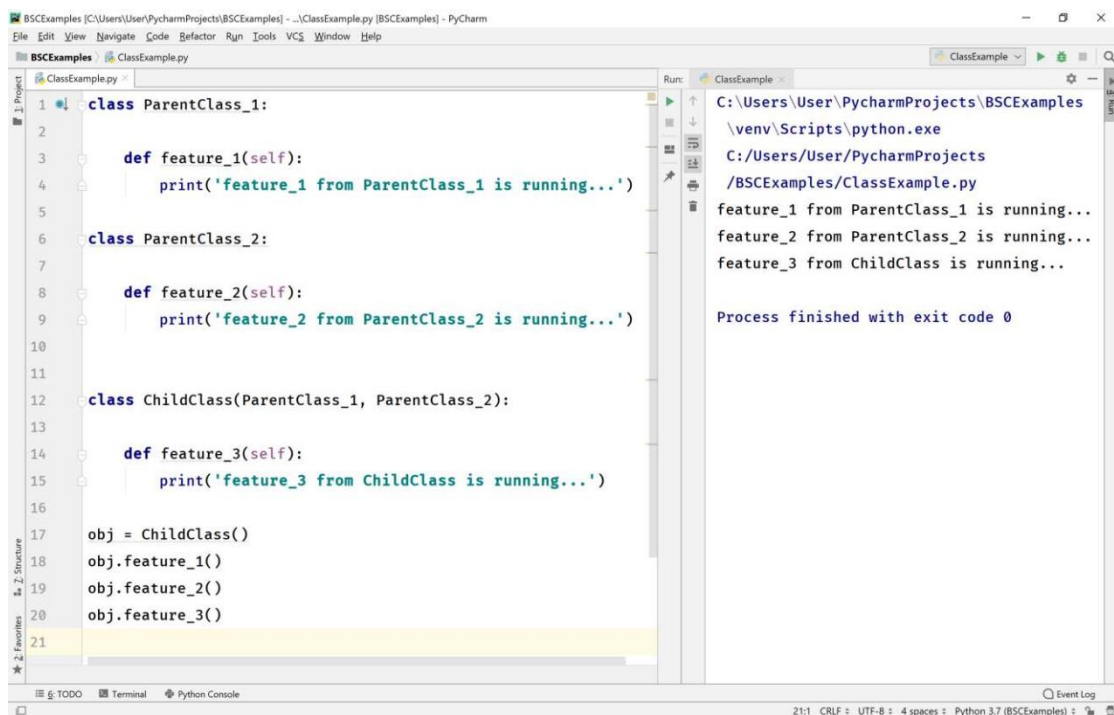
When we run the above example code, it produces the following output.

Multi-Level Inheritance

In this type of inheritance, the child class derives from a class which already derived from another class. Look at the following example code.

Example

```python
classParentClass:
```

```python
    def feature_1(self):
        print('feature_1 from ParentClass is running...')


class ChildClass_1(ParentClass):

    def feature_2(self):
        print('feature_2 from ChildClass_1 is running...')



class ChildClass_2(ChildClass_1):

    def feature_3(self):
        print('feature_3 from ChildClass_2 is running...')


obj = ChildClass_2()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

When we run the above example code, it produces the following output.

Hierarchical Inheritance

In this type of inheritance, two or more child classes derive from one parent class. Look at the following example code.

Example

```python
classParentClass_1:

deffeature_1(self):
print('feature_1 from ParentClass_1 is running...')


classParentClass_2:

deffeature_2(self):
print('feature_2 from ParentClass_2 is running...')



classChildClass(ParentClass_1, ParentClass_2):

deffeature_3(self):
print('feature_3 from ChildClass is running...')

obj = ChildClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

When we run the above example code, it produces the following output.

Hybrid Inheritance

The hybrid inheritance is the combination of more than one type of inheritance. We may use any combination as a single with multiple inheritances, multi-level with multiple inheritances, etc.,

Polymorphism:

Polymorphism is a concept of object oriented programming, which means multiple forms or more than one form. Polymorphism enables using a single interface with input of different datatypes, different class or may be for different number of inputs.

In python as everything is an object hence by default a function can take anything as an argument but the execution of the function might fail as every function has some logic that it follows.

For example,

```
len("hello")# returns 5 as result

len([1,2,3,4,45,345,23,42])# returns 8 as result
```

In this case the function len is polymorphic as it is taking **string** as input in the first case and is taking **list** as input in the second case.

In python, polymorphism is a way of making a function accept objects of different classes if they behave similarly.

Method overriding is a type of polymorphism in which a child class which is extending the parent class can provide different definition to any function defined in the parent class as per its own requirements.

Method Overloading

Method overriding or function overloading is a type of polymorphism in which we can define a number of methods with the same name but with a different number of parameters as well as parameters can be of different types. These methods can perform a similar or different function.

Python doesn't support method overloading on the basis of different number of parameters in functions.

Defining Polymorphic Classes

Imagine a situation in which we have a different class for shapes like Square, Triangle etc which serves as a resource to calculate the area of that shape. Each shape has a different number of dimensions which are used to calculate the area of the respective shape.

Now one approach is to define different functions with different names to calculate the area of the given shapes. The program depicting this approach is shown below:

```python
classSquare:

    side =5

defcalculate_area_sq(self):

return self.side * self.side



classTriangle:

    base =5

    height =4

defcalculate_area_tri(self):

return0.5* self.base * self.height
```

```
sq = Square()

tri = Triangle()

print("Area of square: ", sq.calculate_area_sq())

print("Area of triangle: ", tri.calculate_area_tri())
```

```
Area of square: 25

Area of triangle: 10.0
```

The problem with this approach is that the developer has to remember the name of each function separately. In a much larger program, it is very difficult to memorize the name of the functions for every small operation. Here comes the role of method overloading.

Now let's change the name of functions to calculate the area and give them both same name calculate_area() while keeping the function separately in both the classes with different definitions. In this case the type of object will help in resolving the call to the function. The program below shows the implementation of this type of polymorphism with class methods:

```
class Square:

    side = 5

def calculate_area(self):

return self.side * self.side


class Triangle:

    base = 5

    height = 4

def calculate_area(self):

return 0.5 * self.base * self.height


sq = Square()
```

```
tri = Triangle()

print("Area of square: ", sq.calculate_area())

print("Area of triangle: ", tri.calculate_area())
```

Area of square: 25

Area of triangle: 10.0

As you can see in the implementation of both the classes i.e. Square as well as Triangle has the function with same name calculate_area(), but due to different objects its call get resolved correctly, that is when the function is called using the object sq then the function of class Square is called and when it is called using the object tri then the function of class Triangle is called.

Polymorphism with Class Methods

What we saw in the example above is again obvious behaviour. Let's use a loop which iterates over a tuple of objects of various shapes and call the area function to calculate area for each shape object.

```
sq = Square()

tri = Triangle()


for(obj in(sq, tri)):

    obj.calculate_area()
```

Now this is a better example of polymorphism because now we are treating objects of different classes as an object on which same function gets called.

Here python doesn't care about the type of object which is calling the function hence making the class method polymorphic in nature.

Polymorphism with Functions

Just like we used a loop in the above example, we can also create a function which takes an object of some shape class as input and then calls the function to calculate area for it. For example,

```
find_area_of_shape(obj):

    obj.calculate_area()
```

```
sq = Square()

tri = Triangle()



# calling the method with different objects

find_area_of_shape(sq)

find_area_of_shape(tri)
```

In the example above we have used the same function find_area_of_shape to calculate area of two different shape classes. The same function takes different class objects as arguments and executes perfectly to return the result. This is polymorphism.

Polymorphism with Inheritance
Polymorphism in python defines methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. Also, it is possible to modify a method in a child class that it has inherited from the parent class.

This is mostly used in cases where the method inherited from the parent class doesn't fit the child class. This process of re-implementing a method in the child class is known as Method Overriding. Here is an example that shows polymorphism with inheritance:

```
1   class Bird:
2       def intro(self):
3           print("There are different types of birds")
4
5       def flight(self):
6           print("Most of the birds can fly but some cannot")
7
8   class parrot(Bird):
9       def flight(self):
10          print("Parrots can fly")
11
12  class penguin(Bird):
13      def flight(self):
14          print("Penguins do not fly")
15
16  obj_bird = Bird()
17  obj_parr = parrot()
18  obj_peng = penguin()
19
20  obj_bird.intro()
21  obj_bird.flight()
22
23  obj_parr.intro()
24  obj_parr.flight()
25
26  obj_peng.intro()
27  obj_peng.flight()
```

Output:

There are different types of birds
Most of the birds can fly but some cannot
There are different types of bird
Parrots can fly
There are many types of birds
Penguins do not fly
These are different ways to define polymorphism in Python. With this, we have come to the end of our article. I hope you understood what is polymorphism and how it is used in Python.

Abstraction

Abstraction is one of the most important features of object-oriented programming. It is used to hide the background details or any unnecessary implementation.

Pre-defined functions are similar to data abstraction.

For example, when you use a washing machine for laundry purposes. What you do is you put your laundry and detergent inside the machine and wait for the machine to perform its task. How does it perform it? What mechanism does it use? A user is not required to know the engineering behind its work. This process is typically known as *data abstraction*, when all the unnecessary information is kept hidden from the users.

Code

In Python, we can achieve abstraction by incorporating abstract classes and methods.

Any class that contains abstract method(s) is called an **abstract class**. Abstract methods do not include any implementations – they are always defined and implemented as part of the methods of the sub-classes inherited from the abstract class. Look at the sample syntax below for an abstract class:

```python
from abc import ABC
// abc is a library from where a class ABC is being imported. However, a separate class can also be created.
The importing from the library has nothing to do with abstraction.

Class type_shape(ABC):
```

The class type_shape is inherited from the ABC class. Let's define an abstract method area inside the class type_shape:

```python
from abc import ABC
class type_shape(ABC):
 // abstract method area
 def area(self):
   pass
```

The implementation of an abstract class is done in the sub-classes, which will inherit the class type_shape. We have defined four classes that inherit the abstract class type_shape in the code below

Example:
```python
from abc import ABC
class type_shape(ABC):
  def area(self):
    #abstract method
    pass

class Rectangle(type_shape):
  length = 6
  breadth = 4
  def area(self):
    return self.length * self.breadth

class Circle(type_shape):
  radius = 7
  def area(self):
    return 3.14 * self.radius * self.radius

class Square(type_shape):
  length  = 4
  def area(self):
    return self.length*self.length

class triangle:
  length = 5
  width = 4
  def area(self):
    return 0.5 * self.length * self.width


r = Rectangle() # object created for the class 'Rectangle'
c = Circle() # object created for the class 'Circle'
s = Square() # object created for the class 'Square'
t = triangle() # object created for the class 'triangle'
print("Area of a rectangle:", r.area()) # call to 'area' method defined inside the class.
print("Area of a circle:", c.area()) # call to 'area' method defined inside the class.
print("Area of a square:", s.area()) # call to 'area' method defined inside the class.
print("Area of a triangle:", t.area()) # call to 'area' method defined inside the class.
```

Output
1.14s
Area of a rectangle: 24

Area of a circle: 153.86

Area of a square: 16

Area of a triangle: 10.0

<p style="text-align:center">**UNIT – II**</p>

Data Structures – Definition,Linear Data Structures,Non-Linear Data Structures,Python Specific Data Structures, List,Tuples, Set, Dictionaries, Comprehensions and its Types,Strings,slicing.

Data Structure

Organizing, **managing** and **storing** data is important as it enables easier access and efficient modifications. Data Structures allows you to organize your data in such a way that enables you to store collections of data, relate them and perform operations on them accordingly.

A data structure is classified into two categories:

- o Linear data structure
- o Non-linear data structure

Now let's have a brief look at both these data structures.

What is the Linear data structure?

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

The types of linear data structures are Array, Queue, Stack, Linked List.

Let's discuss each linear data structure in detail.

- o **Array:** An array consists of data elements of a same data type. For example, if we want to store the roll numbers of 10 students, so instead of creating 10 integer type variables, we will create an array having size 10. Therefore, we can say that an array saves a lot of memory and reduces the length of the code.
- o **Stack:** It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.
- o **Queue:** It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue **front** end and **rear** The insertion operation performed at the back end is known ad enqueue, and the deletion operation performed at the front end is known as dequeue.
- o **Linked list:** It is a collection of nodes that are made up of two parts, i.e., data element and reference to the next node in the sequence.
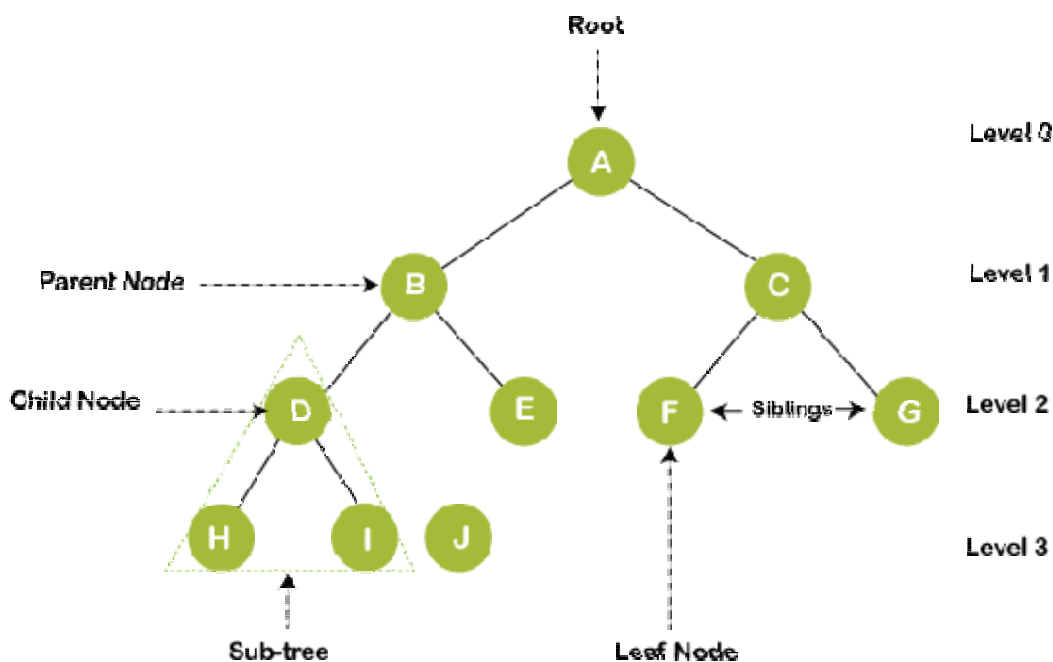
What is a Non-linear data structure?

A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is nonsequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

Trees and **Graphs** are the types of non-linear data structure.

Let's discuss both the data structures in detail.

- o   **Tree**

It is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a **tree** data structure is shown below:

For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the above figure, **A** represents a manager, **B** and **C** represent the officers, and other nodes represent the clerks.

- o   **Graph**

A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices. The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Facebook. In the case of facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.
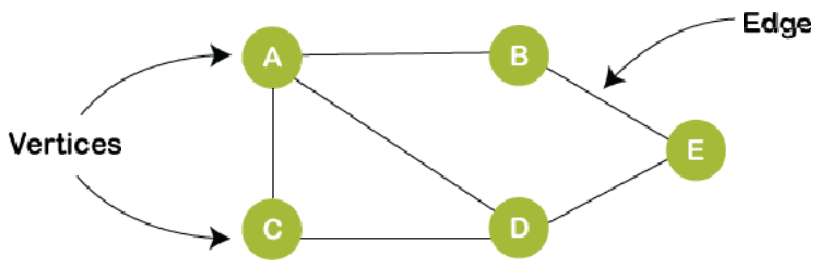
Python Specific Data Structures:

List:
- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:
```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
>>> print(list1)
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
-----------------------
>>> x=list()
>>> x
[]
----------------------------
>>> tuple1=(1,2,3,4)
>>> x=list(tuple1)
>>> x
[1, 2, 3, 4]
```

The list data type has some more methods. Here are all of the methods of list objects:
List Operations:
- Del()
- Append()
- Extend()
- Insert()
- Pop()
- Remove()
- Reverse()
- Sort()

Delete: Delete a list or an item from a list
```
>>> x=[5,3,8,6]
>>> del(x[1])          #deletes the index position 1 in a list

>>> x
```

[5, 8, 6]

--------------- -

>>> del(x)

>>> x   # complete list gets deleted

Append: Append an item to a list

>>> x=[1,5,8,4]

>>> x.append(10)

>>> x

[1, 5, 8, 4, 10]

Extend: Append a sequence to a list.

>>> x=[1,2,3,4]

>>> y=[3,6,9,1]

>>> x.extend(y)

>>> x

[1, 2, 3, 4, 3, 6, 9, 1]

Insert: To add an item at the specified index, use the insert () method:

>>> x=[1,2,4,6,7]

>>> x.insert(2,10) #insert(index no, item to be inserted)

>>> x

[1, 2, 10, 4, 6, 7]

>>> x.insert(4,['a',11])

>>> x

[1, 2, 10, 4, ['a', 11], 6, 7]

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

>>> x=[1, 2, 10, 4, 6, 7]

>>> x.pop()

7

>>> x

[1, 2, 10, 4, 6]

>>> x=[1, 2, 10, 4, 6]

>>> x.pop(2)

10

>>> x

[1, 2, 4, 6]

Remove: The **remove()** method removes the specified item from a given list.

>>> x=[1,33,2,10,4,6]

>>> x.remove(33)

>>> x

[1, 2, 10, 4, 6]

>>> x.remove(4)

>>> x

[1, 2, 10, 6]

Reverse: Reverse the order of a given list.

>>> x=[1,2,3,4,5,6,7]

>>> x.reverse()

>>> x

[7, 6, 5, 4, 3, 2, 1]

Sort: Sorts the elements in ascending order

>>> x=[7, 6, 5, 4, 3, 2, 1]

>>> x.sort()

>>> x

[1, 2, 3, 4, 5, 6, 7]

>>> x=[10,1,5,3,8,7]

>>> x.sort()

>>> x

[1, 3, 5, 7, 8, 10]

    Slicing: Slice out substrings, sub lists, sub Tuples using index.

[Start: stop: steps]

- Slicing will start from index and will go up to **stop** in **step** of steps.

- Default value of start is 0,

- Stop is last index of list

- And for step default is 1

Example:

>>> x='computer'

>>> x[1:4]

'omp'

>>> x[1:6:2]

'opt'

>>> x[3:]

'puter'

>>> x[:5]

'compu'

>>> x[-1]

'r'

>>> x[-3:]

'ter'

>>> x[:-2]

'comput'

>>> x[::-2]

'rtpo'

>>> x[::-1]

'retupmoc'

List:

>>> list1=range(1,6)

>>> list1

range(1, 6)

>>> print(list1)

range(1, 6)

>>> list1=[1,2,3,4,5,6,7,8,9,10]

>>> list1[1:]

[2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> list1[:1]

[1]

>>> list1[2:5]

[3, 4, 5]

>>> list1[:6]

[1, 2, 3, 4, 5, 6]

>>> list1[1:2:4]

[2]

>>> list1[1:8:2]

[2, 4, 6, 8]

Tuple:

>>> list1=(11,12,13,14)

>>> list1[:2]

(11, 12)

To create a slice:

>>> print(slice(3))

slice(None, 3, None)

>>> print(slice(2))

    slice(None, 2, None)

    >>> print(slice(1,6,4))

    slice(1, 6, 4)

To get substring from a given string using slice object:

>>> pystr='python'

>>> x=slice(3)

>>> print(pystr[x])

Pyt

Using –ve index:

>>> pystr='python'

>>> x=slice(1,-3,1)

>>> print(pystr[x])

>>> yt

To get sublist and sub-tuple from a given list and tuple respectively:

>>> list1=['m','r','c','e','t']

>>> tup1=('c','o','l','l','e','g','e')

>>> x=slice(1,4,1)

>>> print(tup1[x])

('o', 'l', 'l')

>>> print(list1[x])

['r', 'c', 'e']

>>> x=slice(1,5,2)

>>> print(list1[x])

['r', 'e']

>>> print(tup1[x])

('o', 'l')

>>> x=slice(-1,-4,-1) #negative index

>>> print(list1[x])

['t', 'e', 'c']

>>> x=slice(-1,-4,-1) #negative index

>>> print(tup1[x])

('e', 'g', 'e')

>>> print(list1[0:3]) #extending indexing syntax

['m', 'r', 'c']

Tuples:
A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

- Supports all operations for sequences.
- Immutable, but member objects may be mutable.
- If the contents of a list shouldn't change, use a tuple to prevent items from accidently being added, changed, or deleted.
- Tuples are more efficient than list due to python's implementation.

We can construct tuple in many ways:
X=()   #no item tuple
X=(1,2,3)
X=tuple(list1)
X=1,2,3,4

Example:
>>> x=(1,2,3)
>>> print(x)
(1, 2, 3)
>>> x
(1, 2, 3)

>>> x=()
>>> x
()

>>> x=[4,5,66,9]
>>> y=tuple(x)
>>> y
(4, 5, 66, 9)

>>> x=1,2,3,4
>>> x
(1, 2, 3, 4)

Some of the operations of tuple are:

- Access tuple items
- Change tuple items
- Loop through a tuple
- Count()
- Index()
- Length()

Access tuple items: Access tuple items by referring to the index number, inside square brackets
>>> x=('a','b','c','g')
>>> print(x[2])
c
Change tuple items: Once a tuple is created, you cannot change its values. Tuples are unchangeable.

>>> x=(2,5,7,'4',8)
>>> x[1]=10
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    x[1]=10
TypeError: 'tuple' object does not support item assignment
>>> x
(2, 5, 7, '4', 8)   # the value is still the same

Loop through a tuple: We can loop the values of tuple using for loop
>>> x=4,5,6,7,2,'aa'
>>> for i in x:
    print(i)

4
5
6
7
2
aa

Count (): Returns the number of times a specified value occurs in a tuple
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.count(2)
4

Index (): Searches the tuple for a specified value and returns the position of where it was found
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)

>>> x.index(2)
1

(Or)

>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=x.index(2)
>>> print(y)
1

Length (): To know the number of items or values present in a tuple, we use len().
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=len(x)
>>> print(y)
12

Set:
A set is a collection which is unordered and unindexed with no duplicate elements. In Python sets are written with curly brackets.

- To create an empty set we use **set()**
- Curly braces '{}' or the set() function can be used to create sets

We can construct tuple in many ways:
X=set()
X={3,5,6,8}
X=set(list1)

Example:
>>> x={1,3,5,6}
>>> x
{1, 3, 5, 6}

>>> x=set()
>>> x
set()

>>> list1=[4,6,"dd",7]
>>> x=set(list1)
>>> x
{4, 'dd', 6, 7}

- We cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Some of the basic set operations are:
- Add()
- Remove()
- Len()
- Item in x
- Pop
- Clear

Add (): To add one item to a set use the add () method. To add more than one item to a set use the update () method.

```
>>> x={"mrcet","college","cse","dept"}
>>> x.add("autonomous")
>>> x
{'mrcet', 'dept', 'autonomous', 'cse', 'college'}
```

```
----
>>> x={1,2,3}
>>> x.update("a","b")
>>> x
{1, 2, 3, 'a', 'b'}
```

```
---
>>> x={1,2,3}
>>> x.update([4,5],[6,7,8])
>>> x
{1, 2, 3, 4, 5, 6, 7, 8}
```
Remove (): To remove an item from the set we use remove or discard methods.
```
>>> x={1, 2, 3, 'a', 'b'}
>>> x.remove(3)
>>> x
{1, 2, 'a', 'b'}
```
Len (): To know the number of items present in a set, we use len().
```
>>> z={'mrcet', 'dept', 'autonomous', 'cse', 'college'}
>>> len(z)
5
```

Item in X: you can loop through the set items using a for loop.
```
>>> x={'a','b','c','d'}
>>> for item in x:
    print(item)
```

```
c
d
a
b
```

pop (): This method is used to remove an item, but this method will remove the **last** item.
Remember that sets are unordered, so you will not know what item that gets removed.

```
>>> x={1, 2, 3, 4, 5, 6, 7, 8}
>>> x.pop()
1
>>> x
{2, 3, 4, 5, 6, 7, 8}
```

Clear (): This method will the set as empty.

```
>>> x={2, 3, 4, 5, 6, 7, 8}
>>> x.clear()
>>> x
set()
```

The set also consist of some mathematical operations like:

Intersection	AND	&
Union	OR	|
Symmetric Diff	XOR	^
Diff	In set1 but not in set2	set1-set2
Subset	set2 contains set1	set1<=set2
Superset	set1 contains set2	set1>=set2

Some examples:

```
>>> x={1,2,3,4}
>>> y={4,5,6,7}
>>> print(x|y)
{1, 2, 3, 4, 5, 6, 7}
```

```
>>> x={1,2,3,4}
>>> y={4,5,6,7}
>>> print(x&y)
{4}
```

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> print(A-B)
{1, 2, 3}
```

```
>>> B = {4, 5, 6, 7, 8}
>>> A = {1, 2, 3, 4, 5}
>>> print(B^A)
{1, 2, 3, 6, 7, 8}
```

Dictionaries:
A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

- Key-value pairs

- Unordered

We can construct or create dictionary like:
X={1:'A',2:'B',3:'c'}
X=dict([('a',3) ('b',4)]
X=dict('A'=1,'B' =2)

Examples:
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2004}

To access specific value of a dictionary, we must pass its key,
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> x=dict1["brand"]
>>> x
'mrcet'

To access keys and values and items of dictionary:
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1.keys()
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['mrcet', 'college', 2004])
>>> dict1.items()
dict_items([('brand', 'mrcet'), ('model', 'college'), ('year', 2004)])

>>> for items in dict1.values():
    print(items)

mrcet
college
2004

>>> for items in dict1.keys():
    print(items)

brand
model
year

>>> for i in dict1.items():
    print(i)

('brand', 'mrcet')

('model', 'college')

('year', 2004)

Some of the operations are:
- Add/change
- Remove
- Length
- Delete

Add/change values: You can change the value of a specific item by referring to its key name

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1["year"]=2005
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2005}
```

Remove(): It removes or pop the specific item of dictionary.

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> print(dict1.pop("model"))
college
>>> dict1
{'brand': 'mrcet', 'year': 2005}
```

Delete: Deletes a particular item.

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> del x[5]
>>> x
```

Length: we use len() method to get the length of dictionary.

```
>>>{1: 1, 2: 4, 3: 9, 4: 16}
{1: 1, 2: 4, 3: 9, 4: 16}
>>> y=len(x)
>>> y
4
```
Iterating over (key, value) pairs:
```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> for key in x:
    print(key, x[key])


1 1
2 4
3 9
4 16
5 25
>>> for k,v in x.items():
```

```
      print(k,v)
```

```
1 1
2 4
3 9
4 16
5 25
```

List of Dictionaries:

```
>>> customers = [{"uid":1,"name":"John"},
   {"uid":2,"name":"Smith"},
       {"uid":3,"name":"Andersson"},
        ]
>>>>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'Andersson'}]
```

```
## Print the uid and name of each customer
>>> for x in customers:
   print(x["uid"], x["name"])
```

```
1 John
2 Smith
3 Andersson
```

```
##Modify an entry, This will change the name of customer 2 from Smith to Charlie
>>> customers[2]["name"]="charlie"
>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'charlie'}]
```

```
##Add a new field to each entry
```

```
>>> for x in customers:
   x["password"]="123456" # any initial value
```

```
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 2, 'name': 'Smith', 'password': '123456'},
{'uid': 3, 'name': 'charlie', 'password': '123456'}]
```

```
##Delete a field
>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]
```

```
>>> del customers[1]
>>> print(customers)
```

[{'uid': 1, 'name': 'John', 'password': '123456'}]

##Delete all fields

```
>>> for x in customers:
    del x["uid"]
```


```
>>> x
{'name': 'John', 'password': '123456'}
```


Sequences:
A sequence is a succession of values bound together by a container that reflects their type. Almost every stream that you put in python is a sequence. Some of them are:

- String
- List
- Tuples
- Range object

String: A string is a group of characters. Since Python has no provision for arrays, we simply use strings. This is how we declare a string. We can use a pair of single or double quotes. Every string object is of the type 'str'.

```
>>> type("name")
<class 'str'>
>>> name=str()
>>> name
''
>>> a=str('mrcet')
>>> a
'mrcet'
>>> a=str(mrcet)
>>> a[2]
'c'
```

List: A list is an ordered group of items. To declare it, we use square brackets.

```
>>> college=["cse","it","eee","ece","mech","aero"]
>>> college[1]
'it'
>>> college[:2]
['cse', 'it']
>>> college[:3]
['cse', 'it', 'eee']
>>> college[3:]
['ece', 'mech', 'aero']
```

>>> college[0]="csedept"

>>> college
['csedept', 'it', 'eee', 'ece', 'mech', 'aero']

Tuple: It is an immutable group of items. When we say immutable, we mean we cannot change a single value once we declare it.

>>> x=[1,2,3]
>>> y=tuple(x)
>>> y
(1, 2, 3)

>>> hello=tuple(["mrcet","college"])
>>> hello
('mrcet', 'college')

Range object: A range() object lends us a range to iterate on; it gives us a list of numbers.

>>> a=range(4)
>>> type(a)
<class 'range'>

>>> for i in range(1,6,2):
   print(i)

1
3
5

Some of the python sequence operations and functions are:
1. Indexing
2. Slicing
3. Adding/Concatenation
4. Multiplying
5. Checking membership
6. Iterating
7. Len()
8. Min()
9. Max()
10. Sum()
11. Sorted()
12. Count()
13. Index()

1. Indexing
   Access any item in the sequence using its index.

string	List

>>> x='mrcet'
>>> print(x[2])
c
	>>> x=['a','b','c']
	>>> print(x[1])
	b

2. Slicing
   Slice out substrings, sub lists, sub tuples using index
   [start : stop : step size]

>>> x='computer'

>>> x[1:4]

'omp'

>>> x[1:6:2]

'opt'

>>> x[3:]

'puter'

>>> x[:5]

'compu'

>>> x[-1]

'r'

>>> x[-3:]

'ter'

>>> x[:-2]

'comput'

>>> x[::-2]

'rtpo'

>>> x[::-1]

'retupmoc'

3. Adding/concatenation:
   Combine 2 sequences of same type using +.

string	List

>>> x='mrcet' + 'college'
>>> print(x)
Mrcetcollege
	>>> x=['a','b'] + ['c']
	>>> print(x)
	['a', 'b', 'c']

4. Multiplying:
   Multiply a sequence using *.

string	List

>>> x='mrcet'*3
    >>> x
'mrcetmrcetmrcet'
	>>> x=[3,4]*2
	>>> x
	[3, 4, 3, 4]

5. Checking Membership:
   Test whether an item is in or not in a sequence.

string	List

>>> x='mrcet'
>>> print('c' in x)
True
	>>> x=['a','b','c']
	>>> print('a' not in x)
	False

6. Iterating:
   Iterate through the items in asequence

```
>>> x=[1,2,3]
>>> for item in x:
        print(item*2)
```

```
2
4
6
```

If we want to display the items of a given list with index then we have to use "enumerate" keyword.

```
>>> x=[5,6,7]
>>> for item,index in enumerate(x):
        print(item,index)
```

```
0 5
1 6
2 7
```

7. len():
   It will count the number of items in a given sequence.

string	List

>>> x="mrcet"
>>> print(len(x))
5
	>>> x=["aa","b",'c','cc']
	>>> print(len(x))
	4

8. min():
   Finds the minimum item in a given sequence lexicographically.

string	List

>>> x="mrcet"
>>> print(min(x))
c
	>>> x=["apple","ant1","ant"]
	>>> print(min(x))
	ant

It is an alpha-numeric type but cannot mix types.

>>> x=["apple","ant1","ant",11]
>>> print(min(x))

Traceback (most recent call last):
 File "<pyshell#73>", line 1, in <module>
  print(min(x))
TypeError: '<' not supported between instances of 'int' and 'str'

9. max():
   Finds the maximum item in a given sequence

string	List

>>> x='cognizant'
>>> print(max(x))
z
	>>> x=["hello","yummy","zebra"]
	>>> print(max(x))
	zebra

It is an alpha-numeric type but cannot mix types.

>>> x=["hello","yummy1","zebra1",22]
>>> print(max(x))

Traceback (most recent call last):
 File "<pyshell#79>", line 1, in <module>
  print(max(x))
TypeError: '>' not supported between instances of 'int' and 'str'

10. Sum:
    Finds the sum of items in a sequence

>>> x=[1,2,3,4,5]
>>> print(sum(x))

15

```
>>> print(sum(x[-2:]))
9
```

Entire string must be numeric type.
```
>>> x=[1,2,3,4,5,"mrcet"]
>>> print(sum(x))
```

```
Traceback (most recent call last):
  File "<pyshell#83>", line 1, in <module>
    print(sum(x))
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

11. Sorted():
Returns a new list of items in sorted order but does not change the original list.

string	List

>>> x='college'
>>> print(sorted(x))
['c', 'e', 'e', 'g', 'l', 'l', 'o']
	>>> x=['a','r','g','c','j','z']
	>>> print(sorted(x))
	['a', 'c', 'g', 'j', 'r', 'z']

12. Count():
It returns the count of an item

string	List

>>> x='college'
>>> print(x.count('l'))
2
>>> 'college'.count('l')
2
	>>> x=['a','b','a','a','c','a']
	>>> print(x.count('a'))
	4

13. Index()
Returns the index of first occurrence

string	List

>>> x='college'
>>> print(x.index('l'))
2
	>>> x=['a','b','a','a','c','a']
	>>> print(x.index('a'))
	0

Comprehensions:
List:

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

>>> list1=[]

>>> for x in range(10):

list1.append(x**2)

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

(or)

This is also equivalent to

>>> list1=list(map(lambda x:x**2, range(10)))

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

(or)

Which is more concise and redable.

>>> list1=[x**2 for x in range(10)]

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Similarily some examples:

```
>>> x=[m for m in range(8)]
>>> print(x)
[0, 1, 2, 3, 4, 5, 6, 7]


>>> x=[z**2 for z in range(10) if z>4]
>>> print(x)
[25, 36, 49, 64, 81]


>>> x=[x ** 2  for x in range (1, 11)  if  x % 2 == 1]
>>> print(x)
[1, 9, 25, 49, 81]


>>> a=5
>>> table = [[a, b, a * b] for b in range(1, 11)]
>>> for i in table:
    print(i)


[5, 1, 5]
[5, 2, 10]
[5, 3, 15]
[5, 4, 20]
[5, 5, 25]
[5, 6, 30]
[5, 7, 35]
[5, 8, 40]
[5, 9, 45]
[5, 10, 50]
```

Tuple:

Tuple Comprehensions are special: The result of a tuple comprehension is special. You might expect it to produce a tuple, but what it does is produce a special "generator" object that we can iterate over.

For example:
```
>>> x = (i for i in 'abc') #tuple comprehension
>>> x
<generator object <genexpr> at 0x033EEC30>


>>> print(x)
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The

only thing that you need to know now about a generator now is that you can iterate over it, but

ONLY ONCE.
 So, given the code

```
>>> x = (i for i in 'abc')
>>> for i in x:
    print(i)


a
b
c
```

Create a list of 2-tuples like (number, square):

```
>>> z=[(x, x**2) for x in range(6)]
>>> z
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

Set:

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

```
>>> x={3*x for x in range(10) if x>5}
>>> x
{24, 18, 27, 21}
```

Dictionary:

Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> z={x: x**2 for x in (2,4,6)}
>>> z
{2: 4, 4: 16, 6: 36}
```

```
>>> dict11 = {x: x*x for x in range(6)}
>>> dict11
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

String Sliceing in Python

If you want to slice strings in Python, it's going to be as simple as this one line below.

res_s =s[start_pos:end_pos:step]
Here,

- **res_s** stores the returned sub-string,
- s is the given string,
- **start_pos** is the starting index from which we need to slice the string s,
- **end_pos** is the ending index, before which the slicing operation would end,
- **step** is the steps the slicing process would take from start_pos to end_pos.
  Note: All of the above three parameters are optional. By default, start_pos is set to **0**, end_pos is considered equal to the length of the string, and step is set to **1**.
  Now let us take some examples to understand how to slice strings in Python in a better way.

Slice Strings in Python – Examples

Slicing Python strings can be done in different ways.

Usually, we access string elements(characters) using simple indexing that starts from *0* up to **n-1**(n is the length of the string). Hence for accessing the **1st** element of a string string1, we can simply use the below code.
s1 =String1[0]
Again, there is another way to access these characters, that is, using **negative indexing**. Negative indexing starts from **-1** to **-n**(n is the length for the given string). Note, negative indexing is done from the other end of the string. Hence, to access the first character this time we need to follow the below-given code.
s1 =String1[-n]
Now let us look at some ways following which we can slice a string using the above concept.

1. Slice Strings in Python with Start and End

We can easily slice a given string by mentioning the starting and ending indices for the desired sub-string we are looking for. Look at the example given below, it explains string slicing using starting and ending indices for both usual and negative indexing method.

#string slicing with two parameters

s ="Hello World!"

res1 =s[2:8]
res2 =s[-4:-1] #using negative indexing

print("Result1 = ",res1)
print("Result2 = ",res2)

Output:
Result1 = llo Wo
Result2 = rld

Here,

- We initialize a string, s as **"Hello World!"**,
- At first, we slice the given string with starting index **2** and ending index as **8**. This means that the resultant sub-string would contain the characters from **s[2]** to **s[8-1]**,
- Similarly, for the next one, the resultant sub-string should contain characters from **s[-4]** to **s[(-1)-1]**. Hence, our output is justified.

2. Slice Strings Using Only the Start or End

As mentioned earlier, all of the three parameters for string slicing are optional. Hence, we can easily accomplish our tasks using one parameter. Look at the code below to get a clear understanding.

#string slicing with one parameter
s1="Charlie"
s2="Jordan"

res1 =s1[2:] #default value of ending position is set to the length of string
res2 =s2[:4] #default value of starting position is set to 0

print("Result1 = ",res1)
print("Result2 = ",res2)

Output:
Result1 = arlie
Result2 = Jord

Here,

- We firstly initialize two strings, **s1** and **s2**,
- For slicing both of them we just mention the **start_pos** for s1, and **end_pos** for s2 only,
- Hence for **res1**, it contains a sub-string of s1 from index 2(as mentioned) up to the last(by default it is set to

n-1). Whereas for res2, the range of the indices lies from 0 upto 4(mentioned).

3. Slice Strings in Python with Step Parameter

The step value decides the jump the slicing operation would take from one index to the other. Look at the example below carefully.

```
#string slicing with step parameter
s="Python"
s1="Kotlin"

res =s[0:5:2]
res1 =s1[-1:-4:-2] #using negative parameters

print("Resultant sliced string = ",res)
print("Resultant sliced string(negative parameters) = ",res1)
```

Output:
Resultant sliced string = Pto
Resultant sliced string(negative parameters) = nl
In the code above,

- We initialize two strings **s** and **s1**, and try to slice them for the given starting and ending indices as we did for our first example,
- But this time we have mentioned a **step** value which was set to 1 by default for the previous examples,
- For res, having a step size 2 means, that while the traversal to get the substring from index 0 to 4, each time the index would be increased by a value of 2. That is the first character is **s[0]** ('P'), the next characters in the sub-string would be **s[0+2]**, and **s[2+2]**, until the index is just less than 5.
- For the next one i.e. **res1**, the step mentioned is (-2). Hence, similar to the previous case, the characters in the substring would be **s1[-1]** , then **s1[(-1)+(-2)]** or **s1[-3]** until the index is just less than (-4).

4. Reversing a String using Slicing in Python

With the use of negative index string slicing in Python, we can also reverse the string and store it in another variable. For doing so we just need to mention a step size of **(-1)**.

Let us see how that works in the example given below.

```
#reversing string using string slicing
s="AskPython"
rev_s =s[::-1] #reverse string stored into rev_s

print(rev_s)
```

Output:
nohtyPksA
As we can see, the string s is reversed and stored into rev_s. **Note**: For this case too, the original string remains intact and untouched.

<div align="center">

UNIT – II

</div>

Searching
- Linear Search and Binary Search.
- **Sorting** - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort

Searching - Linear Search and Binary Search.

There are two types of searching -

- o   Linear Search

- o   Binary Search

Both techniques are widely used to search an element in the given list.

Linear Search

What is a Linear Search?

Linear search is a method of finding elements within a list. It is also called a sequential search. It is the simplest searching algorithm because it searches the desired element in a sequential manner.

It compares each and every element with the value that we are searching for. If both are matched, the element is found, and the algorithm returns the key's index position.

Concept of Linear Search

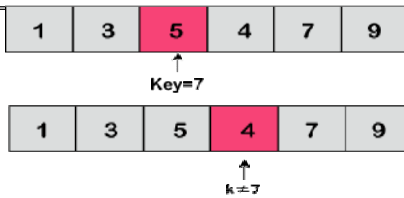Let's understand the following steps to find the element key = 7 in the given list.

Step - 1: Start the search from the first element and Check key = 7 with each element of list x.
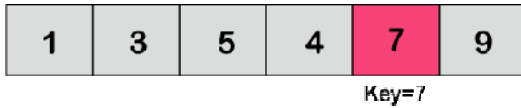

List to be Searched for

Step - 2: If element is found, return the index position of the key.
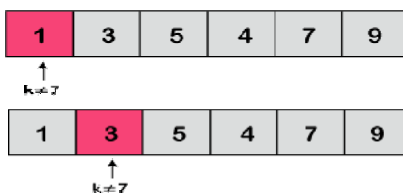
Step - 3: If element is not found, return element is not present.

Linear Search Algorithm

There is list of n elements and key value to be searched.

Below is the linear search algorithm.

1. LinearSearch(list, key)
2.   for each item in the list
3.   if item == value
4.   return its index position
5.   return -1

Python Program

Let's understand the following Python implementation of the linear search algorithm.

Program

```
1. def linear_Search(list1, n, key):
2.
3.       # Searching list1 sequentially
4.       for i in range(0, n):
5.         if (list1[i] == key):
6.             return i
7.       return -1
8.
9.
10. list1 = [1 ,3, 5, 4, 7, 9]
11. key = 7
```

12.

13. n = len(list1)

14. res = linear_Search(list1, n, key)

15. if(res == -1):

16.   print("Element not found")

else:

17.   print("Element found at index: ", res)

Output:

Element found at index: 4

Explanation:

In the above code, we have created a function **linear_Search(),** which takes three arguments - list1, length of the list, and number to search. We defined for loop and iterate each element and compare to the key value. If element is found, return the index else return -1 which means element is not present in the list.

Linear Search Complexity

Time complexity of linear search algorithm -

- o   Base Case - O(1)

- o   Average Case - O(n)

- o   Worst Case -O(n)

Linear search algorithm is suitable for smaller list (<100) because it check every element to get the desired number. Suppose there are 10,000 element list and desired element is available at the last position, this will consume much time by comparing with each element of the list.

To get the fast result, we can use the binary search algorithm.

Binary Search:

A binary search is an algorithm to find a particular element in the list. Suppose we have a list of thousand elements, and we need to get an index position of a particular element. We can find the element's index position very fast using the binary search algorithm.

There are many searching algorithms but the binary search is most popular among them.

The elements in the list must be sorted to apply the binary search algorithm. If elements are not sorted then sort them first.

Let's understand the concept of binary search.

Concept of Binary Search

The divide and conquer approach technique is followed by the recursive method. In this method, a function is called itself again and again until it found an element in the list.

A set of statements is repeated multiple times to find an element's index position in the iterative method. The **while** loop is used for accomplish this task.

Binary search is more effective than the linear search because we don't need to search each list index. The list must be sorted to achieve the binary search algorithm.

Let's have a step by step implementation of binary search.

We have a sorted list of elements, and we are looking for the index position of 45.

[12, 24, 32, 39, 45, 50, 54]

So, we are setting two pointers in our list. One pointer is used to denote the smaller value called **low** and the second pointer is used to denote the highest value called **high**.

Next, we calculate the value of the **middle** element in the array.

1. mid = (low+high)/2
2. Here, the low is 0 and the high is 7.
3. mid = (0+7)/2
4. mid = 3 (Integer)

Now, we will compare the searched element to the mid index value. In this case, **32** is not equal to **45.** So we need to do further comparison to find the element.

If the number we are searching equal to the mid. Then return **mid** otherwise move to the further comparison.

The number to be search is greater than the **middle** number, we compare the **n** with the middle element of the elements on the right side of **mid** and set low to **low = mid + 1.**

Otherwise, compare the **n** with the **middle element** of the elements on the left side of **mid** and set **high** to **high = mid - 1.**
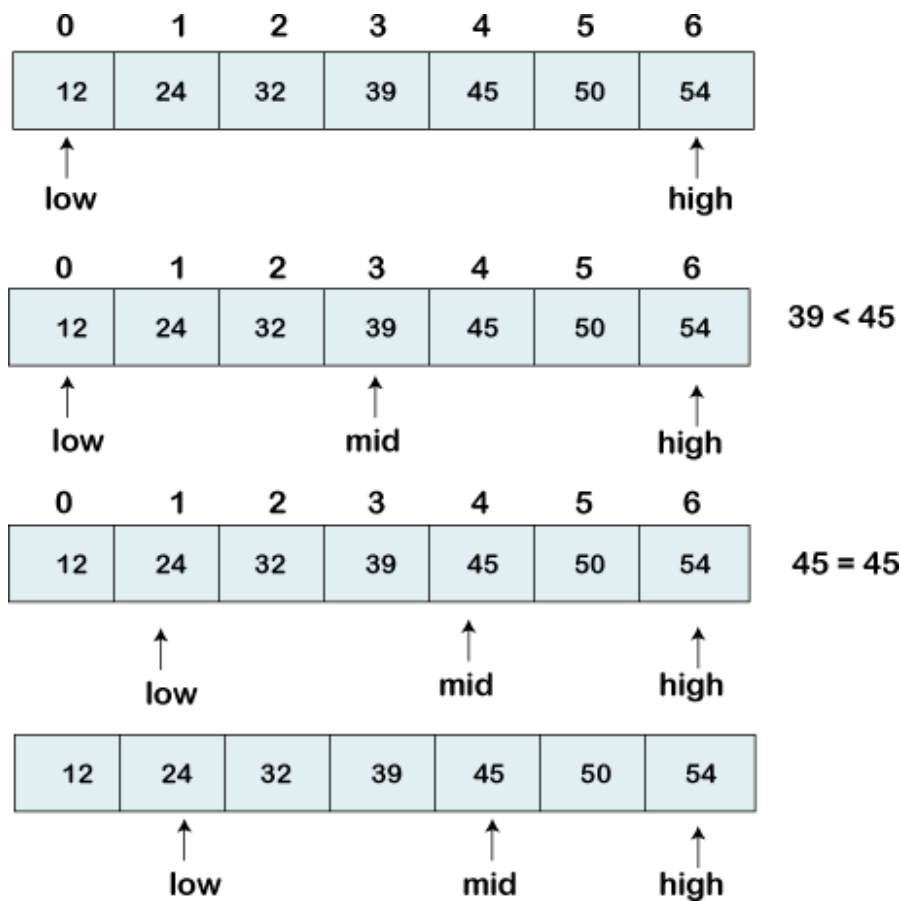
Repeat until the number that we are searching for is found.

Implement a Binary Search in Python

First, we implement a binary search with the iterative method. We will repeat a set of statements and iterate every item of the list. We will find the middle value until the search is complete.

Let's understand the following program of the iterative method.

Python Implementation

1. # Iterative Binary Search Function method Python Implementation
2. # It returns index of n in given list1 if present,
3. # else returns -1
4. def binary_search(list1, n):
5.   low = 0
6.   high = len(list1) - 1
7.   mid = 0

```
8.
9.      while low <= high:
10.         # for get integer result
11.         mid = (high + low) // 2
12.
13.         # Check if n is present at mid
14.         if list1[mid] < n:
15.             low = mid + 1
16.
17.         # If n is greater, compare to the right of mid
18.         elif list1[mid] > n:
19.             high = mid - 1
20.
21.         # If n is smaller, compared to the left of mid
22.         else:
23.             return mid
24.
25.             # element was not present in the list, return -1
26.     return -1
27.
28.
29. # Initial list1
30. list1 = [12, 24, 32, 39, 45, 50, 54]
31. n = 45
32.
33. # Function call
34. result = binary_search(list1, n)
35.
36. if result != -1:
37.     print("Element is present at index", str(result))
38. else:
39.     print("Element is not present in list1")
```

Output:

Element is present at index 4

Explanation:

In the above program -

- o We have created a function called **binary_search()** function which takes two arguments - a list to sorted and a number to be searched.

- o We have declared two variables to store the lowest and highest values in the list. The low is assigned initial value to 0, **high** to **len(list1)** - 1 and mid as 0.

- o Next, we have declared the **while** loop with the condition that the **lowest** is equal and smaller than the **highest** The while loop will iterate if the number has not been found yet.

- o In the while loop, we find the mid value and compare the index value to the number we are searching for.

- o If the value of the mid-index is smaller than **n**, we increase the mid value by 1 and assign it to The search moves to the left side.

- o Otherwise, decrease the mid value and assign it to the **high**. The search moves to the right side.

- o If the n is equal to the mid value then return **mid**.

- o This will happen until the **low** is equal and smaller than the **high**.

- o If we reach at the end of the function, then the element is not present in the list. We return -1 to the calling function.

Sorting - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort.
Bubble Sort:
It is a simple sorting algorithm which sorts 'n' number of elements in the list by comparing the ach pair of adjacent items and swaps them if they are in wrong order.
Algorithm:
1. Starting with the first element (index=0), compare the current element with the next element of a list.
2. If the current element is greater (>) than the next element of the list then swap them.
3. If the current element is less (<) than the next element of the list move to the next element.
4. Repeat step 1 until it correct order is framed.

For ex: list1= [10, 15, 4, 23, 0]   so here we are comparing values again
If > --- yes ---- swap   and again, so we use loops.
If < --- No ---- Do nothing/remains same
#Write a python program to arrange the elements in ascending order using bubble sort:
list1=[9,16,6,26,0]

print("unsorted list1 is", list1)

for j in range(len(list1)-1):

  for i in range(len(list1)-1):

    if list1[i]>list1[i+1]:

      list1[i],list1[i+1]=list1[i+1],list1[i]

      print(list1)

    else:

      print(list1)

  print( )

print("sorted list is",list1)

Output:

unsorted list1 is [9, 16, 6, 26, 0]
[9, 16, 6, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 0, 26]

[6, 9, 16, 0, 26]
[6, 9, 16, 0, 26]
[6, 9, 0, 16, 26]
[6, 9, 0, 16, 26]

[6, 9, 0, 16, 26]
[6, 0, 9, 16, 26]
[6, 0, 9, 16, 26]
[6, 0, 9, 16, 26]

[0, 6, 9, 16, 26]
[0, 6, 9, 16, 26]
[0, 6, 9, 16, 26]
[0, 6, 9, 16, 26]

sorted list is [0, 6, 9, 16, 26]

#If we want to reduce no of iterations/steps in output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/bubb.py

```
list1=[9,16,6,26,0]
print("unsorted list1 is", list1)
for j in range(len(list1)-1,0,-1):
    for i in range(j):
        if list1[i]>list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/bubb2.py
unsorted list1 is [9, 16, 6, 26, 0]
[9, 16, 6, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 0, 26]

[6, 9, 16, 0, 26]
[6, 9, 16, 0, 26]
[6, 9, 0, 16, 26]

[6, 9, 0, 16, 26]
[6, 0, 9, 16, 26]

[0, 6, 9, 16, 26]

sorted list is [0, 6, 9, 16, 26]

 # In a different way:

```
list1=[9,16,6,26,0]
print("unsorted list1 is", list1)
for j in range(len(list1)-1):
    for i in range(len(list1)-1-j):
        if list1[i]>list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/bubb3.py

unsorted list1 is [9, 16, 6, 26, 0]
[9, 16, 6, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 26, 0]
[9, 6, 16, 0, 26]

[6, 9, 16, 0, 26]
[6, 9, 16, 0, 26]
[6, 9, 0, 16, 26]

[6, 9, 0, 16, 26]
[6, 0, 9, 16, 26]

[0, 6, 9, 16, 26]

sorted list is [0, 6, 9, 16, 26]

 # Program to give input from the user to sort the elements
```python
list1=[]
num=int(input("enter how many numbers:"))
print("enter values")
for k in range(num):
    list1.append(int(input()))
print("unsorted list1 is", list1)
for j in range(len(list1)-1):
    for i in range(len(list1)-1):
        if list1[i]>list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/bubb4.py

enter how many numbers:5
enter values
5
77
4
66
30
unsorted list1 is [5, 77, 4, 66, 30]
[5, 77, 4, 66, 30]

[5, 4, 77, 66, 30]
[5, 4, 66, 77, 30]
[5, 4, 66, 30, 77]

[4, 5, 66, 30, 77]
[4, 5, 66, 30, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]

[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]

[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]
[4, 5, 30, 66, 77]

sorted list is [4, 5, 30, 66, 77]

#bubble sort program for descending order

```python
list1=[9,16,6,26,0]
print("unsorted list1 is", list1)
for j in range(len(list1)-1):
    for i in range(len(list1)-1):
        if list1[i]<list1[i+1]:
            list1[i],list1[i+1]=list1[i+1],list1[i]
            print(list1)
        else:
            print(list1)
    print( )
print("sorted list is",list1)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-2/pyyy/bubbdesc.py
unsorted list1 is [9, 16, 6, 26, 0]
[16, 9, 6, 26, 0]
[16, 9, 6, 26, 0]
[16, 9, 26, 6, 0]
[16, 9, 26, 6, 0]

[16, 9, 26, 6, 0]
[16, 26, 9, 6, 0]
[16, 26, 9, 6, 0]
[16, 26, 9, 6, 0]

[26, 16, 9, 6, 0]

[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]

[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]
[26, 16, 9, 6, 0]

sorted list is [26, 16, 9, 6, 0]

Selection Sort:
Sort (): Built-in list method

Sorted (): built in function

- Generally this algorithm is called as in-place comparison based algorithm. We compare numbers and place them in correct position.
- Search the list and find out the min value, this we can do it by min () method.
- We can take min value as the first element of the list and compare with the next element until we find small value.

Algorithm:

1. Starting from the first element search for smallest/biggest element in the list of numbers.
2. Swap min/max number with first element
3. Take the sub-list (ignore sorted part) and repeat step 1 and 2 until all the elements are sorted.

#Write a python program to arrange the elements in ascending order using selection sort:
```
list1=[5,3,7,1,9,6]
print(list1)
for  i in  range(len(list1)):
    min_val=min(list1[i:])
    min_ind=list1.index(min_val)
    list1[i],list1[min_ind]=list1[min_ind],list1[i]
    print(list1)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/selectasce.py
[5, 3, 7, 1, 9, 6]
[1, 3, 7, 5, 9, 6]
[1, 3, 7, 5, 9, 6]
[1, 3, 5, 7, 9, 6]
[1, 3, 5, 6, 9, 7]
[1, 3, 5, 6, 7, 9]
[1, 3, 5, 6, 7, 9]
#Write a python program to arrange the elements in descending order using selection sort:
```
list1=[5,3,7,1,9,6]
print(list1)
for i in range(len(list1)):
    min_val=max(list1[i:])
    min_ind=list1.index(min_val)
```

```
      list1[i],list1[min_ind]=list1[min_ind],list1[i]
print(list1)
```
Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/selecdecs.py
[5, 3, 7, 1, 9, 6]
[9, 7, 6, 5, 3, 1]
Note: If we want the elements to be sorted in descending order use max () method in place of min ().

Insertion Sort:

- Insertion sort is not a fast sorting algorithm. It is useful only for small datasets.
- It is a simple sorting algorithm that builds the final sorted list one item at a time.

Algorithm:
1. Consider the first element to be sorted & the rest to be unsorted.
2. Take the first element in unsorted order (u1) and compare it with sorted part elements(s1)
3. If u1<s1 then insert u1 in the correct order, else leave as it is.
4. Take the next element in the unsorted part and compare with sorted element.
5. Repeat step 3 and step 4 until all the elements get sorted.

Write a python program to arrange the elements in ascending order using insertion sort (with functions)
```
def insertionsort(my_list):
#we need to sorrt the unsorted part at a time.
   for index in range(1,len(my_list)):
      current_element=my_list[index]
      pos=index
      while current_element<my_list[pos-1]and pos>0:
         my_list[pos]=my_list[pos-1]
         pos=pos-1
      my_list[pos]=current_element
list1=[3,5,1,0,10,2]              ──────────▶  num=int(input("enter how many elements to be in list"))
insertionsort(list1)                           list1=[int(input()) for i in range (num)]
print(list1)
```
Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/inserti.py
[0, 1, 2, 3, 5, 10]
Write a python program to arrange the elements in descending order using insertion sort (with functions)
```
def insertionsort(my_list):
#we need to sorrt the unsorted part at a time.
   for index in range(1,len(my_list)):
      current_element=my_list[index]
      pos=index
      while current_element>my_list[pos-1]and pos>0:
         my_list[pos]=my_list[pos-1]
         pos=pos-1
      my_list[pos]=current_element
```

#list1=[3,5,1,0,10,2]

```
#insertionsort(list1)
#print(list1)
num=int(input("enter how many elements to be in list"))
list1=[int(input())for i in range(num)]
insertionsort(list1)
print(list1)
```
Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/insertdesc.py

enter how many elements to be in list 5

8

1

4

10

2

[10, 8, 4, 2, 1]

Merge Sort:
Generally this merge sort works on the basis of divide and conquer algorithm. The three steps need to be followed is divide, conquer and combine. We will be dividing the unsorted list into sub list until the single element in a list is found.
Algorithm:
1. Split the unsorted list.
2. Compare each of the elements and group them
3. Repeat step 2 until whole list is merged and sorted.

Write a python program to arrange the elements in ascending order using Merge sort (with functions)

```
def mergesort(list1):
    if len(list1)>1:
        mid=len(list1)//2
        left_list=list1[:mid]
        right_list=list1[mid:]
        mergesort(left_list)
        mergesort(right_list)
        i=0
        j=0
        k=0
        while i<len(left_list) and j<len(right_list):
            if left_list[i]<right_list[j]:
                list1[k]=left_list[i]
                i=i+1
                k=k+1
            else:
                list1[k]=right_list[j]
                j=j+1
                k=k+1
```

```
while i<len(left_list):
```

```
        list1[k]=left_list[i]
        i=i+1
        k=k+1
    while  j<len(right_list):
        list1[k]=right_list[j]
        j=j+1
        k=k+1
num=int(input("how many numbers in list1"))
list1=[int(input()) for x in range(num)]
mergesort(list1)
print("sorted list1",list1)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/merg.py

how many numbers in list15
5
9
10
1
66
sorted list1 [1, 5, 9, 10, 66]

Quick Sort:
Algorithm:
1. Select the pivot element
2. Find out the correct position of pivot element in the list by rearranging it.
3. Divide the list based on pivot element
4. Sort the sub list recursively

Note: Pivot element can be first, last, random elements or median of three values.
In the following program we are going to write 3 functions. The first function is to find pivot element and its correct position. In second function we divide the list based on pivot element and sort the sub list and third function (main fun) is to print input and output.
Write a python program to arrange the elements in ascending order using Quick sort (with functions)
#To get the correct position of pivot element:
```
def pivot_place(list1,first,last):
    pivot=list1[first]
    left=first+1
    right=last
    while True:
        while left<=right and list1[left]<=pivot:
            left=left+1
        while left<=right and list1[right]>=pivot:
            right=right-1
        if right<left:
            break
        else:
```

list1[left],list1[right]=list1[right],list1[left]

```
      list1[first],list1[right]=list1[right],list1[first]
      return right
#second function
def quicksort(list1,first,last):
   if first<last:
      p=pivot_place(list1,first,last)
      quicksort(list1,first,p-1)
      quicksort(list1,p+1,last)
#main fun
list1=[56,25,93,15,31,44]
n=len(list1)
quicksort(list1,0,n-1)
print(list1)
```

Output:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/qucksort.py
[15, 25, 31, 44, 56, 93]

Write a python program to arrange the elements in descending order using Quick sort (with functions)
```
#To get the correct position of pivot element:
def pivot_place(list1,first,last):
   pivot=list1[first]
   left=first+1
   right=last
   while True:
      while left<=right and list1[left]>=pivot:
         left=left+1
      while left<=right and list1[right]<=pivot:
         right=right-1
      if right<left:
         break
      else:
         list1[left],list1[right]=list1[right],list1[left]
   list1[first],list1[right]=list1[right],list1[first]
   return right
def quicksort(list1,first,last):
   if first<last:
      p=pivot_place(list1,first,last)
      quicksort(list1,first,p-1)
      quicksort(list1,p+1,last)
#main fun
list1=[56,25,93,15,31,44]
n=len(list1)
quicksort(list1,0,n-1)
print(list1)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/qukdesc.py

[93, 56, 44, 31, 25, 15]

UNIT – III

Data Structures –Definition, Linear Data Structures, Non-Linear Data Structures
Linked Lists - Singly Linked Lists, Doubly Linked Lists, Circular Linked Lists.
Stacks - Overview of Stack, Implementation of Stack (List & Linked list), Applications of Stack
Queues: Overview of Queue, Implementation of Queue(List & Linked list), Applications of Queues,
Priority Queues.

A linear data structure is one where data items are arranged in a linear fashion. Each member is attached to its neighboring elements. The structure permits single-level data storage because the data elements are stored in a linear fashion. The data can be traversed in one run. A linear data structure does not maximize memory.

A linear data structure is one where data items are ordered either sequentially or in a linear fashion, and each member is attached to its neighboring elements. The linear data structure's elements can all be accessed in one run. There are four primary types of linear data structures: Queue, Linked Lists, Stack, and Arrays.

A data structure can be defined as a system of storage that organizes and stores data. Data structures are the method of organizing data on computers to make it easy to access and modify. Choosing the appropriate data structure according to your needs and your project is essential. If you require storing sequential data in memory, then you may opt for the array data structure.

The types of data structures or their kind can be broken into two:

1. Linear Data Structure

Linear data structures are organized in a particular order, and it is done so because the elements are ordered in a particular pattern; they are simple to implement. Nevertheless, linear data structures might not be the best choice for sophisticated systems because of their operational complexity.

2. Nonlinear Data Structure

Nonlinear elements of data structures aren't ordered in a particular way, as opposed to linear structures. They are arranged in a hierarchical manner in which each element may be linked to one element. The graph and tree-based structures divide those that are nonlinear.

What Is Linear Data Structure?

A linear data structure is known as a data structure that allows data elements to be arranged in a sequential or linear fashion. Each element is attached with its next and previous adjacent. A linear data structure only has one level and performs linear searching in the data structure. We can therefore traverse all elements in a single run. Because computer memory is linearly arranged, linear data structures are simple to implement. Linear data structure examples are array, linked list, stack, queue, etc.

Characteristics of Linear Data Structure

This is a data structure in which the data arrangement follows a linear trend. Each element in the data structure is arranged in a linear fashion so that it can be linked to the previous and next elements. The elements are stored in a linear fashion, which allows for single-level data storage. The data can be accessed in one run.

- It's a data structure that stores and manages data in a linear order.
- The data elements of the sequence are linked from one to the next.
- Implementing the linear structure of data within a computer's RAM is simple, provided that the data are organized sequentially.
- Array, queue. Stack, linked list, etc., are a few examples of such a structure.
- Only one relationship exists between the data elements in the data structure.
- Because the data elements are stored on a single level, it is possible to traverse the data elements in one run.
- If a linear data storage structure is used, it is not well utilized.
- The complexity of the structure's time increases with an increase in its size.

Types in Linear Data Structure

The Array, Linked List, Stack, and Queue are all kinds of linear data structure types. Let's look at each in greater detail.

1. Array

An Array is a type of framework that stores homogeneous parts in connected memory locations. It is precisely the same kinds of objects that are saved sequentially inside an array. The fundamental concept behind linear arrays in the data structure is that multiple pieces of data similar can be stored together. Before saving the data in an array, it is essential that the size of an array need to be established. Every aspect of the array can be accessed or modified, and the data is stored in an index to identify their places of their own.

The concept of an array can be explained through an easy example of saving the marks for all the students in a class if you assume that there are twenty pupils. The size of your array needs to be identified as 20. The marks of all pupils could also be saved in the array created without the need to create separate variables for markings for every pupil. A simple traversal of the array will allow access to the parts.

2. Linked List

It is this type of data system where individual objects are stored sequentially. Each object within the system has a reference and information for the next object. The last node on the linked list has an identifier for null.

The first element linked to the list is known as the "mind" of the list. There are a lot of differences between linked lists and other kinds of data structures. This concerns the allocation of memory, the inside structures of the structure, and the operation carried out on this linked list.

Finding an element of a linked list is slow when compared to arrays since the indexing of an array aids in locating the element. The process in the case of a linked list must begin in mind and work through the whole structure until the component you want is found.

Contrary to this, the advantage of linked lists would be that the addition or even the removal of elements at the beginning is completed very quickly.

There are basically two kinds of lists that are linked:

One-Linked Listing: This type of system contains the guide or address of the next node within the current node. This means that a node is the last one to have the address as well as a guide is a NULL. Example: A > B > C > D > E > NULL.

Double-Linked Listing: The title suggests that every node has two references that are related to it. One reference is directed to the previous node, and the second reference is directed to the following node. Traversal is possible both ways since an example guide is available from the previous nodes. In addition, the explicit access requirement isn't needed to delete. Example: NULL B C D E-> NULL.

- **A linked list that has a circular form:** elements in circular linked lists are linked in such a manner that the group is created. Since the linked list is circular, there is no limit, and therefore there is no null. This type of linked checklist has the ability to be used either doubly or singly. There is no specific starting point, and any node in the data may be the starting point. The guide for the last node points to the first node. Example: A > B > C > D-> E.

3. Stack
The stack is a different type of structure in that components within the data system adhere to the principles in LIFO- Last in, First out (or) FILO- First In, Last Out. Two kinds of operations can be attributed to the stack, i.e., the pushing and pop operation. Push is necessary for components that need to be added to the collection, and pop is employed when the component previously added needs to be removed from the collection. The extraction can be removed to add the final component.

Some examples of stacks include the elimination of the recursion. In situations where words have to be reversed or when using editors, if the last word entered will be removed, stacks are used to begin with (using the undo function).

4. Queue
The queue is the type of system that stores data in which components are stored in accordance with an order of First In, First Out (FIFO). The exact sequence is followed to carry out the required actions by the components. The main difference between the queue and stacks lies in eliminating an element. It is in the area where the object that was added the most recently is removed first in the stack. At the same time, the case with a queue element that was initially added is removed first.

Both the end and the conclusion of the system are used for the removal of data and the addition of data. The two main operations that govern the structure of the queue are dequeue and enqueue. Enqueue refers to the procedure where the addition of a component is allowed to be used in creating data. And dequeue refers to the method where the removal of components is allowed. This is the primary aspect of the queue in this case.

Examples of queues: Like the queues made when waiting for the bus or anywhere else, the data structure is similar to the pattern. It is possible to imagine someone who is waiting for the bus and then sitting in the first spot as the person who made it to the queue first. The person who arrived first is likely to be the first person to get on an actual bus, i.e., get out of the line. Queues are created when a number of customers share the same enthusiasm and need to be served based on the foundation which is first created for the server.

What Is a Nonlinear Data Structure?
Data structures in which the data elements do not have to be arranged in a linear or sequential fashion are referred to as nonlinear data structures. In a nonlinear structure, only one level of data is not used. So, it is impossible to traverse the entire structure in one run. Nonlinear data structures can be straightforward to design and implement compared to linear data structures. They make use of computer memory effectively when compared to a linear structure. Examples of this are graphs and trees.

1. Trees
A tree can be described as a nonlinear information system comprised of a variety of nodes. The nodes of the tree data structure are organized in order of hierarchy.

It is composed of a root node that corresponds to the different kids' nodes that are present at the next level. The tree is built from a level foundation, and root nodes are minimum kid nodes based on the tree's growing order. In the tree of binary, the position of the root has two nodes, which means it has the capacity to be able to have up to two kids per node and not more than that.

For Non-Linear Data Structure, the nonlinear system of data cannot be used directly, so it is implemented using the linear data structure, such as linked lists and arrays. The tree itself is a large info structure, and it is broken down into various kinds like Binary trees, Binary search trees, Heap, AVL trees max Heap, min-heap, and many more.

The types of trees mentioned above are different based on the properties they possess. The term "tree" refers to an acyclic, nonlinear connected graph. It's a nonlinear system of data like a tree. A node may be linked to one or more nodes. It's a collection of nodes linked by direct (or possibly indirectly) edges. It's comprised of a significant node called the 'root node.'

2. Graphs

A graph can be described as a nonlinear information system with a restricted number of vertices and edges, and these edges are used to join the vertex pairs. The graph is classified by certain characteristics. When we talk about a large graph, it is composed of a set of vertex together with every vertex that is attached to the various vertex sets, gaining an advantage over the two. The vertices hold the data elements, whereas they are the tip of the link between the vertex sets.

The graph concept is essential in many fields. The network is represented with the help of the graph principle as well as its ideas within computer networks. In Maps, we consider each spot as a vertex, and the road between two locations is considered an edge. The main goal of graph representation is to determine the distance between two vertex points through an advantage mass that is minimal. You can explore the complexity on a deeper level by opting for Computer Programming courses for beginners through KnowledgeHut.

What Is the Difference Between Linear and Non-Linear Data Structures?

Linear refers to data components that have been sequentially attached. Each component can be traversed in a single run. Nonlinear data structures are data components that have been hierarchically attached. Linear and non-linear data structures or all data components can be found at one level in a linear data structure.

Let's talk about the differences between Linear or Nonlinear data structures based on data arrangements, data elements being transferred, implementation, levels, and time complexity.

Basis of	Linear Data Structure	Nonlinear Data Structure
Data Arrangements:	Linear data structures are data elements that are organized linearly. Each element is attached to the next and previous adjacent.	Data elements in a nonlinear data system are attached hierarchically.
Levels:	A linear data structure only has one level.	Multilevels are possible in nonlinear data structures.
Implementation:	It is much easier than nonlinear data structures.	It is more complicated than a linear data structure but can be implemented.
Transferring Data Elements:	Data elements in linear data structures can only be traversed once	Nonlinear data structures can be traversed over multiple runs, but data elements cannot be traversed in one run.

Basis of	Linear Data Structure	Nonlinear Data Structure
Memory Usage:	Memory is not used efficiently in a linear data structure.	Memory is used efficiently in nonlinear data structures.
Examples:	Its examples are queue, linked list, array, stack, etc.	Its examples include graphs and trees.
Usage:	The application of linear data structures is mainly used in software development.	Artificial Intelligence and Image Processing are two examples of applications for nonlinear data structures.

Which Data Structure Is Better: Linear or Non-Linear?
Nonlinear data structures are data components that exist at multiple levels. Linear detail structures are much easier to implement. Linear data structures are much easier to comprehend and apply than nonlinear data structures. It is possible to traverse linear data structures in one run.

Therefore, it is impossible to traverse all components of each run. It is more difficult to implement nonlinear data structures than linear data structures.

Most Common Operations Performed in Linear Data Structures
Next, we'll look at the most popular data structure operations. Data structure operations refer to the methods that allow you to manipulate data within a data structure. These are the most popular data structure operations:

1. Traversal
A traversal operation is used to visit every data structure node in a particular order. This is a technique that can be used to print, search, display, and read data stored in a structure.

2. Insertion
Insertion operations add data elements to a database structure. This can be operated at any point in the data structure, including its beginning, middle, and end.

3. Deletion
Data elements are removed from a data structure by deletion operations. These operations are usually performed on nodes that no longer need them.

4. Merge
Two data structures can be combined into one using merge operations. This is used when two data structures need to be combined into one.

5. Copy
To create a duplicate data structure, copy operations can be used. You can do this by copying every element from the original structure to the new one.

6. Search Operation

Search operations can be used to locate a particular data element within a data structure. These operations often use a compare function in order to determine if two elements are equal.

7. Sort Operation

Sort operations are used for arranging data elements in a data structure in a particular order. You can use a variety of sorting algorithms to accomplish this, including bubble sort, insertion sort, and merge sort.

Conclusion

Data structures are essential for computer programs to be able to handle the increasing data volume. If data is not organized in a structured manner, it can make it difficult to achieve the desired results for projects. It is important to manage the data in order to make it easy and hassle-free. It's known as a linear system when data components are arranged in sequential order.

However, if data elements are arranged in nonlinear ways, it's known as a nonlinear structure. Machine learning languages, real-life problems, and other areas continue to have a wide range of data systems. This article discusses the different types of nonlinear structures and explains the differences between nonlinear structures and linear data structures.

Linked Lists:

Linked lists are one of the most commonly used data structures in any programming language.Linked Lists, on the other hand, are different. Linked lists, do not store data at contiguous memory locations. For each item in the memory location, linked list stores value of the item and the reference or pointer to the next item. One pair of the linked list item and the reference to next item constitutes a node.

The following are different types of linked lists.

- Single Linked List
  A single linked list is the simplest of all the variants of linked lists. Every node in a single linked list contains an item and reference to the next item and that's it.

- Doubly Linked List
- Circular Linked List
- Linked List with Header

 # Python program to create a linked list and display its elements.

The program creates a linked list using data items input from the user and displays it.

Solution:

1. Create a class Node with instance variables data and next.

2. Create a class Linked List with instance variables head and last_node.

3. The variable head points to the first element in the linked list while last_node points to the last.

4. Define methods append and display inside the class Linked List to append data and display the linked list respectively.

5. Create an instance of Linked List, append data to it and display the list.

Program:

```
class Node:
def__init__(self, data):
```

```python
self.data= data
self.next=None

class LinkedList:
def __init__(self):
self.head=None
self.last_node=None

def append(self, data):
if self.last_node is None:
self.head= Node(data)
self.last_node=self.head
else:
self.last_node.next= Node(data)
self.last_node=self.last_node.next

def display(self):
    current =self.head
while current is not None:
print(current.data, end =' ')
        current = current.next

a_llist = LinkedList()
n =int(input('How many elements would you like to add? '))
for i in range(n):
   data =int(input('Enter data item: '))
   a_llist.append(data)
print('The linked list: ', end ='')
a_llist.display()
```

Program Explanation

1. An instance of Linked List is created.
2. The user is asked for the number of elements they would like to add. This is stored in n.
3. Using a loop, data from the user is appended to the linked list n times.
4. The linked list is displayed.

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/link1.py
How many elements would you like to add? 5
Enter data item: 4
Enter data item: 4
Enter data item: 6

Enter data item: 8

Enter data item: 9
The linked list: 4 4 6 8 9

Doubly Linked List

A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes.

Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

Advantages of Doubly Linked List

1. Traversal can be done on either side means both in forward as well as backward.

2. Deletion Operation is more efficient if the pointer to delete node is given.

Disadvantages of Linked List

1. Since it requires extra pointer that is the previous pointer to store previous node reference.

2. After each operation like insertion-deletion, it requires an extra pointer that is a previous pointer which needs to be maintained.

So, a typical node in the doubly linked list consists of three fields:

- o **Data** represents the data value stored in the node.

- o **Previous** represents a pointer that points to the previous node.

- o **Next** represents a pointer that points to the next node in the list.
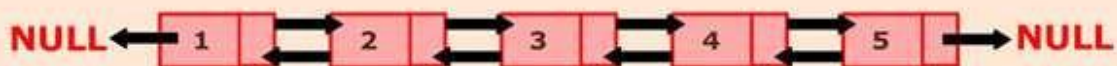
The above picture represents a doubly linked list in which each node has two pointers to point to previous and next node respectively. Here, node 1 represents the head of the list. The previous pointer of the head node will always point to NULL. Next pointer of node one will point to node 2. Node 5 represents the tail of the list whose previous pointer will point to node 4, and the next will point to NULL.

ALGORITHM:

1. Define a Node class which represents a node in the list. It will have three properties: data, previous which will point to the previous node and next which will point to the next node.

2. Define another class for creating a doubly linked list, and it has two nodes: head and tail. Initially, head and tail will point to null.

3. addNode() will add node to the list:

o It first checks whether the head is null, then it will insert the node as the head.

o Both head and tail will point to a newly added node.

o Head's previous pointer will point to null and tail's next pointer will point to null.

o If the head is not null, the new node will be inserted at the end of the list such that new node's previous pointer will point to tail.

o The new node will become the new tail. Tail's next pointer will point to null.

a. display() will show all the nodes present in the list.

o Define a new node 'current' that will point to the head.

o Print current.data till current points to null.

o Current will point to the next node in the list in each iteration.

PROGRAM:

1. #Represent a node of doubly linked list
2. **class** Node:
3.   **def** __init__(self,data):
4.   self.data = data;
5.   self.previous = None;
6.   self.next = None;
7.
8. **class** DoublyLinkedList:
9.   #Represent the head and tail of the doubly linked list
10.   **def** __init__(self):
11.   self.head = None;

```
12.        self.tail = None;
13.
14.    #addNode() will add a node to the list
15.    def addNode(self, data):
16.       #Create a new node
17.       newNode = Node(data);
18.
19.       #If list is empty
20.       if(self.head == None):
21.          #Both head and tail will point to newNode
22.          self.head = self.tail = newNode;
23.          #head's previous will point to None
24.          self.head.previous = None;
25.          #tail's next will point to None, as it is the last node of the list
26.          self.tail.next = None;
27.       else:
28.          #newNode will be added after tail such that tail's next will point to newNode
29.          self.tail.next = newNode;
30.          #newNode's previous will point to tail
31.          newNode.previous = self.tail;
32.          #newNode will become new tail
33.          self.tail = newNode;
34.          #As it is last node, tail's next will point to None
35.          self.tail.next = None;
36.
37.    #display() will print out the nodes of the list
38.    def display(self):
39.       #Node current will point to head
40.       current = self.head;
41.       if(self.head == None):
42.          print("List is empty");
43.          return;
```

44.   print("Nodes of doubly linked list: ");

```
45.        while(current != None):
46.            #Prints each node by incrementing pointer.
47.            print(current.data),;
48.            current = current.next;
49.
50. dList = DoublyLinkedList();
51. #Add nodes to the list
52. dList.addNode(1);
53. dList.addNode(2);
54. dList.addNode(3);
55. dList.addNode(4);
56. dList.addNode(5);
57.
58. #Displays the nodes present in the list
59. dList.display();
```

Output:

```
Nodes of doubly linked list:
1 2 3 4 5
```

Circular Linked List

The circular linked list is a kind of linked list. First thing first, the node is an element of the list, and it has two parts that are, data and next. Data represents the data stored in the node, and next is the pointer that will point to the next node. Head will point to the first element of the list, and tail will point to the last element in the list. In the simple linked list, all the nodes will point to their next element and tail will point to null.

The circular linked list is the collection of nodes in which tail node also point back to head node. The diagram shown below depicts a circular linked list. Node A represents head and node D represents tail. So, in this list, A is pointing to B, B is pointing to C and C is pointing to D but what makes it circular is that node D is pointing back to node A.
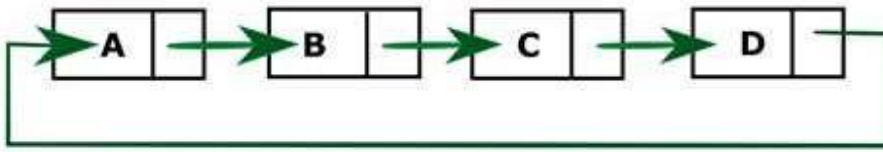
ALGORITHM:

1. Define a Node class which represents a node in the list. It has two properties data and next which will point to the next node.

2. Define another class for creating the circular linked list, and it has two nodes: head and tail. It has two methods: add() and display() .

3. add() will add the node to the list:

o It first checks whether the head is null, then it will insert the node as the head.

o Both head and tail will point to the newly added node.

o If the head is not null, the new node will be the new tail, and the new tail will point to the head as it is a circular linked list.

a. display() will show all the nodes present in the list.

o Define a new node 'current' that will point to the head.

o Print current.data till current will points to head

o Current will point to the next node in the list in each iteration.

PROGRAM:

1. #Represents the node of list.
2. **class** Node:
3.   **def** __init__(self,data):
4.   self.data = data;
5.   self.next = None;
6.

```
7.   class CreateList:
8.   #Declaring head and tail pointer as null.
9.   def __init__(self):
10.    self.head = Node(None);
11.    self.tail = Node(None);
12.    self.head.next = self.tail;
13.    self.tail.next = self.head;
14.
15.  #This function will add the new node at the end of the list.
16.  def add(self,data):
17.    newNode = Node(data);
18.    #Checks if the list is empty.
19.    if self.head.data is None:
20.      #If list is empty, both head and tail would point to new node.
21.      self.head = newNode;
22.      self.tail = newNode;
23.      newNode.next = self.head;
24.    else:
25.      #tail will point to new node.
26.      self.tail.next = newNode;
27.      #New node will become new tail.
28.      self.tail = newNode;
29.      #Since, it is circular linked list tail will point to head.
30.      self.tail.next = self.head;
31.
32.  #Displays all the nodes in the list
33.  def display(self):
34.    current = self.head;
35.    if self.head is None:
36.      print("List is empty");
37.      return;
38.    else:
```

39.   **print**("Nodes of the circular linked list: ");

```
40.        #Prints each node by incrementing pointer.
41.        print(current.data),
42.        while(current.next != self.head):
43.           current = current.next;
44.           print(current.data),
45.
46.
47. class CircularLinkedList:
48.   cl = CreateList();
49.   #Adds data to the list
50.   cl.add(1);
51.   cl.add(2);
52.   cl.add(3);
53.   cl.add(4);
54.   #Displays all the nodes present in the list
55.   cl.display();
```

Output:

```
Nodes of the circular linked list:
1 2 3 4
```

Stacks:
Stack works on the principle of "Last-in, first-out". Also, the inbuilt functions in Python make the code short and simple. To add an item to the top of the list, i.e., to push an item, we use append() function and to pop out an element we use pop() function.

#Python code to demonstrate Implementing stack using list

```
stack = ["Amar", "Akbar", "Anthony"]
stack.append("Ram")
stack.append("Iqbal")
print(stack)
print(stack.pop())
print(stack)
print(stack.pop())
print(stack)
```

Output:

['Amar', 'Akbar', 'Anthony', 'Ram', 'Iqbal']

Iqbal

['Amar', 'Akbar', 'Anthony', 'Ram']

Ram

['Amar', 'Akbar', 'Anthony']

Stack Using Linked List

A stack using a linked list is just a simple linked list with just restrictions that any element will be added and removed using push and pop respectively. In addition to that, we also keep *top* pointer to represent the top of the stack. This is described in the picture given below.
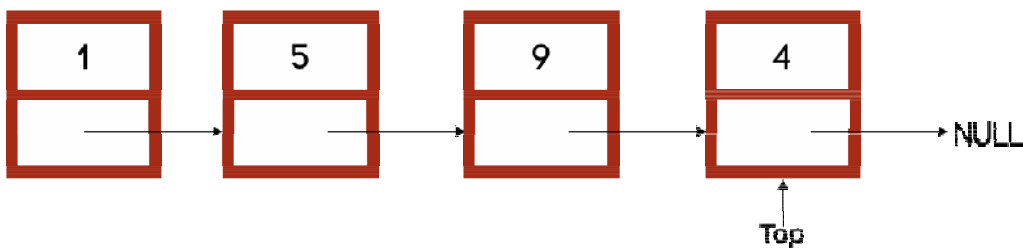
Stack Operations:
1. **push()** : Insert the element into linked list nothing but which is the top node of Stack.
2. **pop()** : Return top element from the Stack and move the top pointer to the second node of linked list or Stack.
3. **peek():** Return the top element.
4. **display():** Print all element of Stack.

A stack will be empty if the linked list won't have any node i.e., when the *top* pointer of the linked list will be null. So, let's start by making a function to check whether a stack is empty or not.

IS_EMPTY(S)

  if S.top == null

     return TRUE

  return FALSE

Now, to push any node to the stack (S) - PUSH(S, n), we will first check if the stack is empty or not. If the stack is empty, we will make the new node head of the linked list and also point the *top* pointer to it.
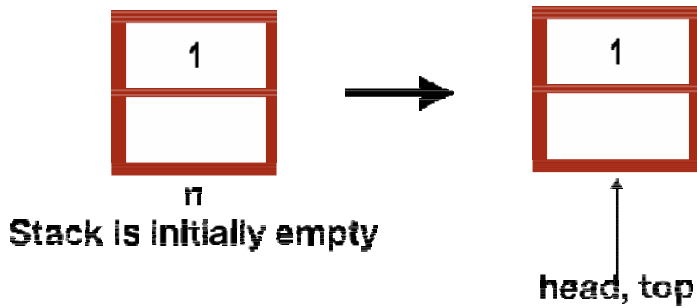
Stack is initially empty

head, top

PUSH(S, n)
    if IS_EMPTY(S) //stack is empty
        S.head = n //new node is the head of the linked list
        S.top = n //new node is the also the top

If the stack is not empty, we will add the new node at the last of the stack. For that, we will point *next* of the *top* to the new node - (S.top.next = n) and the make the new node *top* of the stack - (S.top = n).
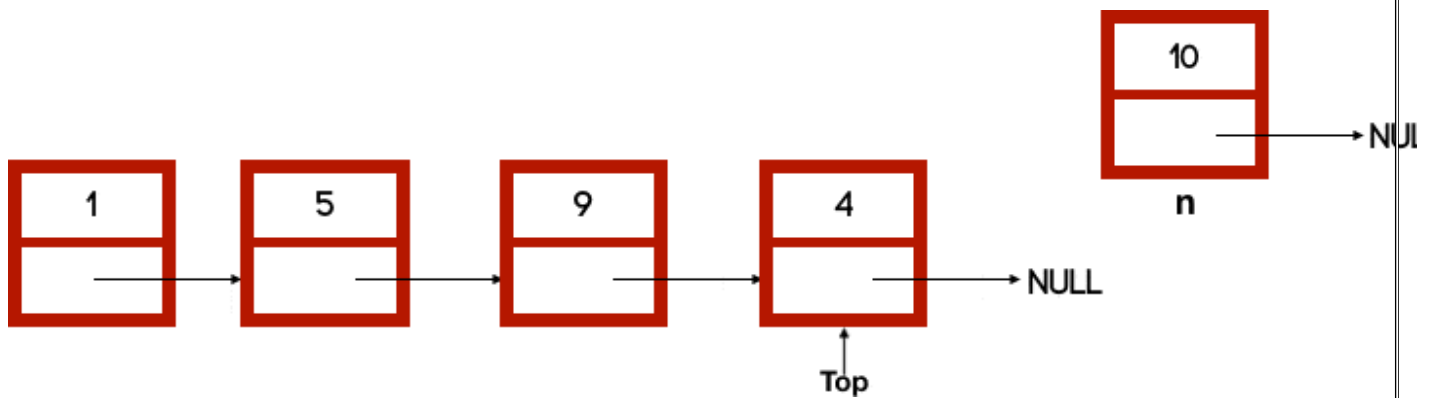
PUSH(S, n)
    if IS_EMPTY(S) //stack is empty
        ...
    else
        S.top.next = n
        S.top = n

PUSH(S, n)

 if IS_EMPTY(S) //stack is empty

   S.head = n //new node is the head of the linked list

   S.top = n //new node is the also the top

 else

   S.top.next = n

> **S.top = n**

Similarly, to remove a node (pop), we will first check if the stack is empty or not as we did in the implementation with array.

POP(S)
   if IS_EMPTY(S)
      Error "Stack Underflow"

In the case when the stack is not empty, we will first store the value in *top* node in a temporary variable because we need to return it after deleting the node.

POP(S)
   if IS_EMPTY(S)
     ...
   else
     x = S.top.data

Now if the stack has only one node (*top* and *head* are same), we will just make both *top* and *head* null.
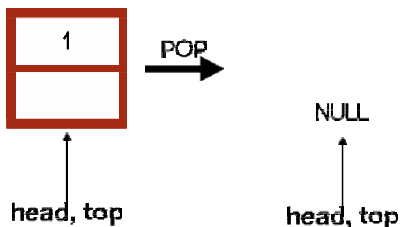
POP(S)
   if IS_EMPTY(S)
     ...
   else
     ...
     if S.top == S.head //only one node
       S.top = NULL
       S.head = NULL

If the stack has more than one node, we will move to the node previous to the *top* node and make the *next* of point it to null and also point the *top* to it.
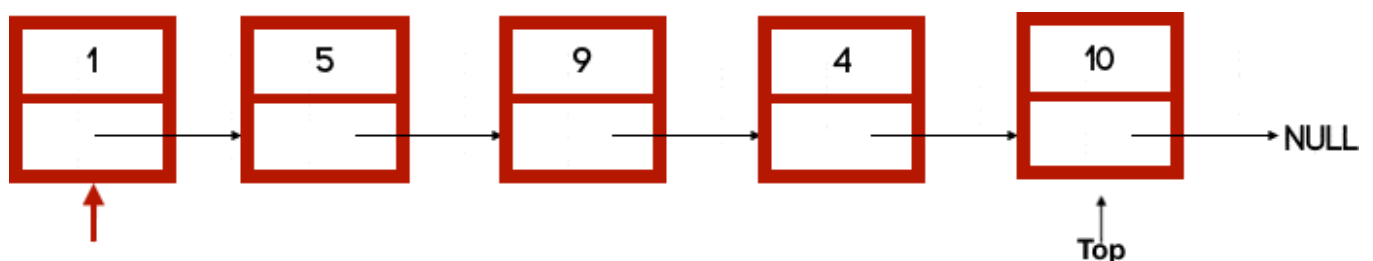
POP(S)
   ...
     ...
     if S.top == S.head //only one node
       ...

```
    else
        tmp = S.head
        while tmp.next != S.top //iterating to the node previous to top
            tmp = tmp.next
        tmp.next = NULL //making the next of the node null
        S.top = tmp //changing the top pointer
```

We first iterated to the node previous to the top node and then we marked its *next* to null - tmp.next = NULL. After this, we pointed the *top* pointer to it - S.top = tmp.

At last, we will return the data stored in the temporary variable - return x.

POP(S)

 if IS_EMPTY(S)

   Error "Stack Underflow"

 else

   x = S.top.data

   if S.top == S.head //only one node

     S.top = NULL

     S.head = NULL

   else

     tmp = S.head

     while tmp.next != S.top //iterating to the node previous to top

       tmp = tmp.next

     tmp.next = NULL //making the next of the node null

     S.top = tmp //changing the top pointer

   return x

Stack using linked list

```python
class Node():
    def __init__(self, data):
        self.data = data
        self.next = None


class Stack():
    def __init__(self):
        self.head = None
        self.top = None


def traversal(s):
    temp = s.head #temporary pointer to point to head

    a = ''
    while temp != None: #iterating over stack
        a = a+str(temp.data)+'\t'
        temp = temp.next

    print(a)


def is_empty(s):
    if s.top == None:
        return True
    return False
```

```
    def push(s, n):

    if is_empty(s): #empty

       s.head = n

       s.top = n

    else:

       s.top.next = n

       s.top = n


    def pop(s):

    if is_empty(s):

       print("Stack Underflow")

       return -1000

    else:

       x = s.top.data

       if s.top == s.head: # only one node

          s.top = None

          s.head = None

       else:

          temp = s.head

          while temp.next != s.top: #iterating to the last element

             temp = temp.next

          temp.next = None

          del s.top

          s.top = temp

       return x
```

```
if __name__ == '__main__':

s = Stack()


a = Node(10)

b = Node(20)

c = Node(30)


pop(s)

push(s, a)

push(s, b)

push(s, c)


traversal(s)

pop(s)

traversal(s)
```

Applications of Stack

There are many applications of a stack. Some of them are:

- Stacks are used in backtracking algorithms.
- They are also used to implement undo/redo functionality in a software.
- Stacks are also used in syntax parsing for many compilers.
- Stacks are also used to check proper opening and closing of parenthesis.

Queue

Similar to stacks, a queue is also an Abstract Data Type or ADT. A **queue** follows **FIFO (First-in, First out)** policy. It is equivalent to the queues in our general life. For example, a new person enters a queue at the last and the person who is at the front (who must have entered the queue at first) will be served first.

Similar to a queue of day to day life, in Computer Science also, a new element enters a queue at the last (tail of the queue) and removal of an element occurs from the front (head of the queue).
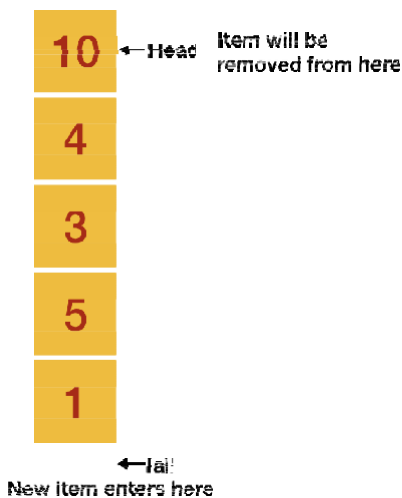
Similar to the stack, we will implement the queue using a linked list as well as with an array. But let's first discuss the operations which are done on a queue.

Enqueue → Enqueue is an operation which adds an element to the queue. As stated earlier, any new item enters at the tail of the queue, so Enqueue adds an item to the tail of a queue.

Enqueue

Dequeue → It is similar to the pop operation of stack i.e., it returns and deletes the front element from the queue.

Dequeue

isEmpty → It is used to check whether the queue has any element or not.

isFull → It is used to check whether the queue is full or not.

Front → It is similar to the top operation of a stack i.e., it returns the front element of the queue (but don't delete it).

Before moving forward to code up these operations, let's discuss the applications of a queue.

Applications of Queue

Queues are used in a lot of applications, few of them are:

- Queue is used to implement many algorithms like Breadth First Search (BFS), etc.
- It can be also used by an operating system when it has to schedule jobs with equal priority
- Customers calling a call center are kept in queues when they wait for someone to pick up the calls

Queue Using an Array

We will maintain two pointers - *tail* and *head* to represent a queue. *head* will always point to the oldest element which was added and *tail* will point where the new element is going to be added.
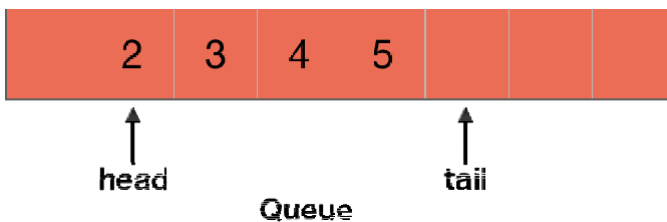
To insert any element, we add that element at *tail* and increase the *tail* by one to point to the next element of the array.
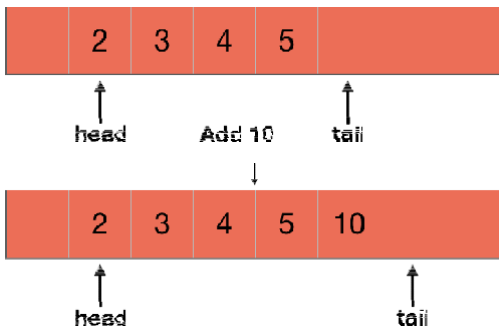
Suppose *tail* is at the last element of the queue and there are empty blocks before head as shown in the picture given below.
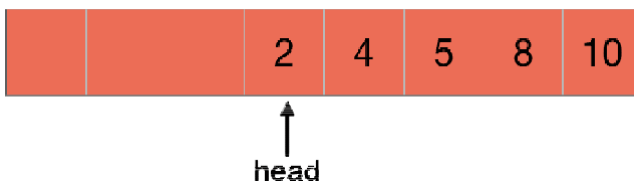
In this case, our *tail* will point to the first element of the array and will follow a circular order.
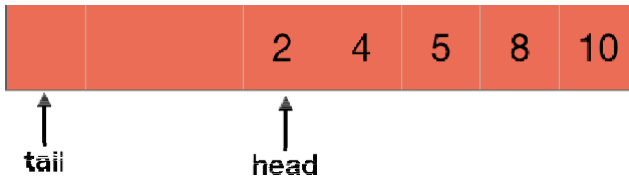
Initially, the queue will be empty i.e., both *head* and *tail* will point to the same location i.e., at index 1. We can easily check if a queue is empty or not by checking if *head* and *tail* are pointing to the same location or not at any time.

An Empty Queue

IS_EMPTY(Q)

  If Q.tail == Q.head

    return True

  return False

Similarly, we will say that if the *head* of a queue is 1 more than the *tail*, the queue is full.
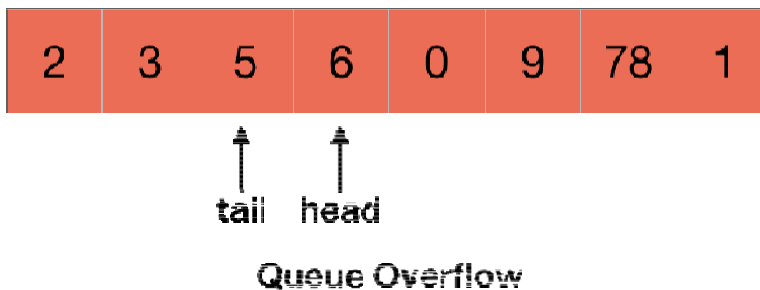
Queue Overflow

IS_FULL(Q)

  if Q.head = Q.tail+1

    return True

  Return False

Now, we have to deal with the enqueue and the dequeue operations.

To enqueue any item to the queue, we will first check if the queue is full or not i.e.,

Enqueue(Q, x)
  if isFull(Q)

Error "Queue Overflow"

else

  …

If the queue is not full, we will add the element to the *tail* i.e, Q[Q.tail] = x.
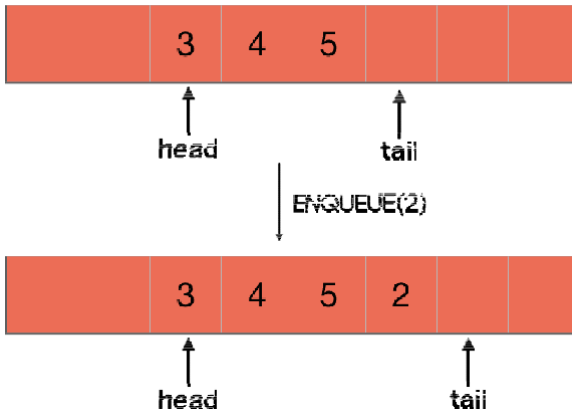
While adding the element, it might be possible that we have added the element at the last of the array and in this case, the *tail* will go to the first element of the array.
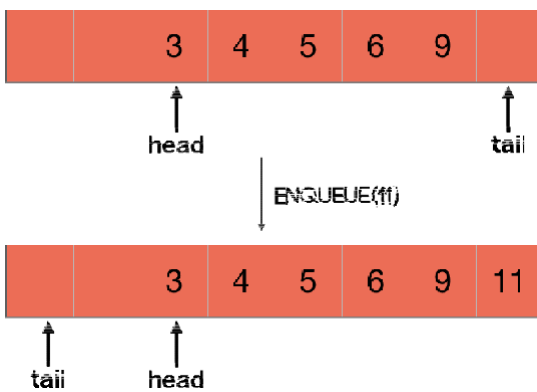
Otherwise, we will just increase the *tail* by 1.

Enqueue(Q, x)

  if isFull(Q)

    Error "Queue Overflow"

  else

    Q[Q.tail] = x

    if Q.tail == Q.size

      Q.tail = 1

    else

> **Q.tail = Q.tail+1**

To dequeue, we will first check if the queue is empty or not. If the queue is empty, then we will throw an error.

Dequeue(Q, x)
  if isEmpty(Q)
    Error "Queue Underflow"
  else
    …

To dequeue, we will first store the item which we are going to delete from the queue in a variable because we will be returning it at last.

Dequeue(Q, x)
  if isEmpty(Q)
    Error "Queue Underflow"
  else
    x = Q[Q.head]
    …

Now, we just have to increase the head pointer by 1. And in the case when the head is at the last element of the array, it will go 1.
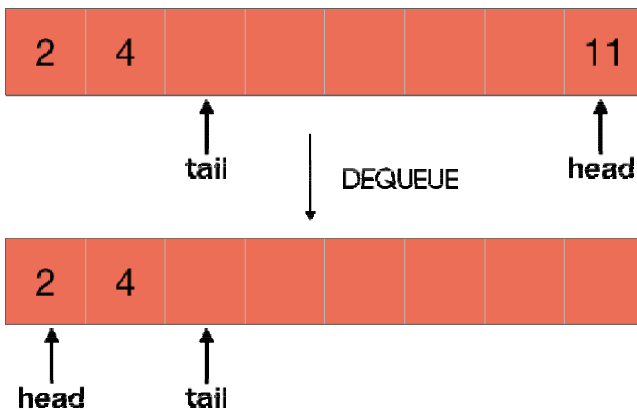
Dequeue(Q, x)

  if isEmpty(Q)

    Error "Queue Underflow"

  else

    x = Q[Q.head]

    if Q.head == Q.size

      Q.head = 1

```
      else

         Q.head = Q.head+1

      return x
```

```python
class Queue:

def __init__(self, size):

  self.head = 1

  self.tail = 1

  self.Q = [0]*(size)

  self.size = size


def is_empty(self):

  if self.tail == self.head:

    return True

  return False


def is_full(self):

  if self.head == self.tail+1:

    return True

  return False


def enqueue(self, x):

  if self.is_full():

    print("Queue Overflow")

  else:
```

```python
        self.Q[self.tail] = x

      if self.tail == self.size:

        self.tail = 1

      else:

        self.tail = self.tail+1


  def dequeue(self):

   if self.is_empty():

    print("Underflow")

   else:

     x = self.Q[self.head]

     if self.head == self.size:

       self.head = 1

     else:

       self.head = self.head+1

     return x


  def display(self):

   i = self.head

   while(i < self.tail):

     print(self.Q[i])

    if(i == self.size):

    i = 0

    i = i+1
```

```
if __name__ == '__main__':

q = Queue(10)

q.enqueue(10)

q.enqueue(20)

q.enqueue(30)

q.enqueue(40)

q.enqueue(50)

q.display()


print("")


q.dequeue()

q.dequeue()

q.display()


print("")


q.enqueue(60)

q.enqueue(70)

q.display()
```

Queue Using Linked List

As we know that a linked list is a dynamic data structure and we can change the size of it whenever it is needed. So, we are not going to consider that there is a maximum size of the queue and thus the queue will never overflow. However, one can set a maximum size to restrict the linked list from growing more than that size.

As told earlier, we are going to maintain a *head* and a *tail* pointer to the queue. In the case of an empty

queue, *head* will point to NULL.

NULL

↑

head

IS_EMPTY(Q)

  if Q.head == null

    return True

  return False

We will point the *head* pointer to the first element of the linked list and the *tail* pointer to the last element of it as shown in the picture given below.
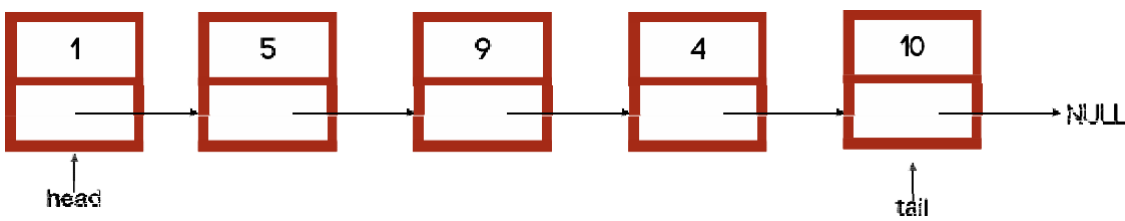
The enqueue operation simply adds a new element to the last of a linked list.
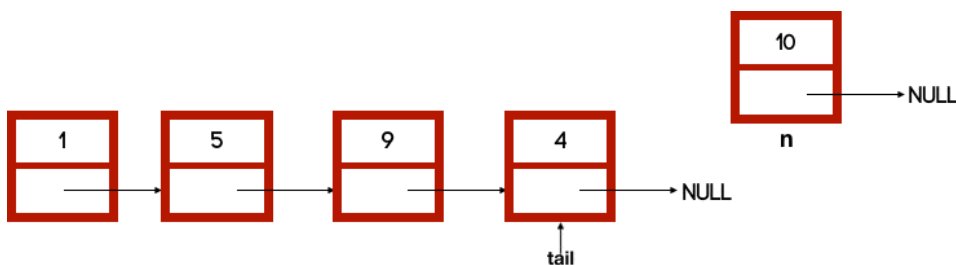
However, if the queue is empty, we will simply make the new node *head* and *tail* of the queue.
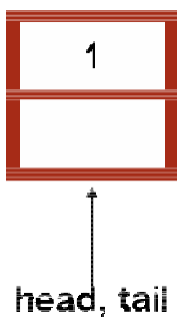
ENQUEUE(Q, n)

 if IS_EMPTY(Q)

   Q.head = n

   Q.tail = n

 else

   Q.tail.next = n

   Q.tail = n

To dequeue, we need to remove the head of the linked list. To do so, we will first store its data in a variable because we will return it at last and then point *head* to its next element.

```
x = Q.head.data
Q.head = Q.head.next
return x
```

We will execute the above codes when the queue is not empty. If it is, we will throw the "Queue Underflow" error.

DEQUEUE(Q, n)

 if IS_EMPTY(Q)

   Error "Queue Underflow"

 else

   x = Q.head.data

   Q.head = Q.head.next

   return x

Queue using Linked List

class Node():

def __init__(self, data):

  self.data = data

```python
      self.next = None


  class Queue():

  def __init__(self):

    self.head = None

    self.tail = None


  def traversal(q):

  temp = q.head #temporary pointer to point to head


  a = ''

  while temp != None: #iterating over queue

    a = a+str(temp.data)+'\t'

    temp = temp.next


  print(a)


  def is_empty(q):

  if q.head == None:

    return True

  return False


  def enqueue(q, n):

  if is_empty(q): #empty

    q.head = n
```

```python
    q.tail = n

  else:

   q.tail.next = n

   q.tail = n


  def dequeue(q):

  if is_empty(q):

   print("Queue Underflow")

   return -1000

  else:

   x  =  q.head.data

   temp  = q.head

   q.head = q.head.next

   del temp

   return x


  if __name__ == '__main__':

  q = Queue()


  a = Node(10)

  b = Node(20)

  c = Node(30)


  dequeue(q)

  enqueue(q, a)
```

enqueue(q, b)

enqueue(q, c)

traversal(q)

dequeue(q)

traversal(q)

Priority QueuesA queue has FIFO (first-in-first-out) ordering where items are taken out or accessed on a first-come-first-served basis. Examples of queues include a queue at a movie ticket stand, as shown in the illustration above. But, what is a priority queue?

A priority queue is an *abstract data structure* (a data structure defined by its behaviour) that is like a normal

queue but where each item has a special "key" to quantify its "priority". For example, if the movie cinema

decides to serve loyal customers first, it will order them by their loyalty (points or number of tickets

purchased). In such a case, the queue for tickets will no longer be first-come-first-served, but most-loyal-

first-served. The customers will be the "items" of this priority queue while the "priority" or "key" will be

their loyalty.

A priority queue can be of two types:

1. **Max Priority Queue**: Which arranges the data as per descending order of their priority.

2. **Min Priority Queue**: Which arranges the data as per ascending order of their priority.

In a priority queue, following factors come into play:

1. In priority queue, data when inserted, is stored based on its priority.

2. When we remove a data from a priority queue(min), the data at the top, which will be the data with least priority, will get removed.

3. But, this way priority queue will not be following the basic priniciple of a queue, First in First Out(FIFO). Yes, it won't! Because a priority queue is an advanced queue used when we have to arrange and manipulate data as per the given priority.

Implementing Priority Queue

So now we will design our very own minimum priority queue using python **list** and object oriented concept.

Below are the algorithm steps:

1. Node: The Node class will be the element inserted in the priority queue. You can modify the Node class as per your requirements.

2. insert: To add a new data element(Node) in the priority queue.
   - o If the priority queue is empty, we will insert the element to it.
   - o If the priority queue is not empty, we will traverse the queue, comparing the priorities of the existing nodes with the new node, and we will add the new node before the node with priority greater than the new node.
   - o If the new node has the highest priority, then we will add the new node at the end.

3. delete: To remove the element with least priority.

4. size: To check the size of the priority queue, in other words count the number of elements in the queue and return it.

5. show: To print all the priority queue elements.

Priority Queue Program

class for Node with data and priority

class Node:

    def __init__(self, info, priority):

```python
        self.info = info

        self.priority = priority


    # class for Priority queue

    class PriorityQueue:


     def __init__(self):

      self.queue = list()

       # if you want you can set a maximum size for the queue


     def insert(self, node):

       # if queue is empty

       if self.size() == 0:

         # add the new node

        self.queue.append(node)

       else:

         # traverse the queue to find the right place for new node
```

```
for x in range(0, self.size()):
```

```python
        # if the priority of new node is greater

        if node.priority >= self.queue[x].priority:

          # if we have traversed the complete queue

          if x == (self.size()-1):

            # add new node at the end

          self.queue.insert(x+1, node)

          else:

            continue

        else:

          self.queue.insert(x, node)

          return True


    def delete(self):

      # remove the first node from the queue

      return self.queue.pop(0)



    def show(self):
```

```
for x in self.queue:
```

```
        print str(x.info)+" - "+str(x.priority)



  def size(self):

    return len(self.queue)



    pQueue = PriorityQueue()

    node1 = Node("C", 3)

    node2 = Node("B", 2)

    node3 = Node("A", 1)

    node4 = Node("Z", 26)

    node5 = Node("Y", 25)

    node6 = Node("L", 12)



    pQueue.insert(node1)

    pQueue.insert(node2)

    pQueue.insert(node3)

    pQueue.insert(node4)
```

pQueue.insert(node5)

```
pQueue.insert(node6)

pQueue.show()

print("  -------")

pQueue.delete()

pQueue.show()
```

<div align="center">

UNIT -IV

</div>

Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching. **Graphs -** Introduction, Directed vs Undirected Graphs, Weighted vs Unweighted Graphs, Representations, Applications: Breadth First Search, Depth First Search.

Linear Linked list is the default linked list and a linear data structure in which data is not stored in contiguous memory locations but each data node is connected to the next data node via a pointer, hence forming a chain.

Skip list in Data structure
What is a skip list?
A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

Skip list structure

It is built in two layers: The lowest layer and Top layer.

The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

Complexity table of the Skip list

S. No	Complexity	Average case	Worst case
1.	Access complexity	O(logn)	O(n)
2.	Search complexity	O(logn)	O(n)
3.	Delete complexity	O(logn)	O(n)
4.	Insert complexity	O(logn)	O(n)
5.	Space complexity	-	O(nlogn)

Working of the Skip list

Let's take an example to understand the working of the skip list. In this example, we have 14 nodes, such that these nodes are divided into two layers, as shown in the diagram.

The lower layer is a common line that links all nodes, and the top layer is an express line that links only the main nodes, as you can see in the diagram.

Suppose you want to find 47 in this example. You will start the search from the first node of the express line and continue running on the express line until you find a node that is equal a 47 or more than 47.

You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.
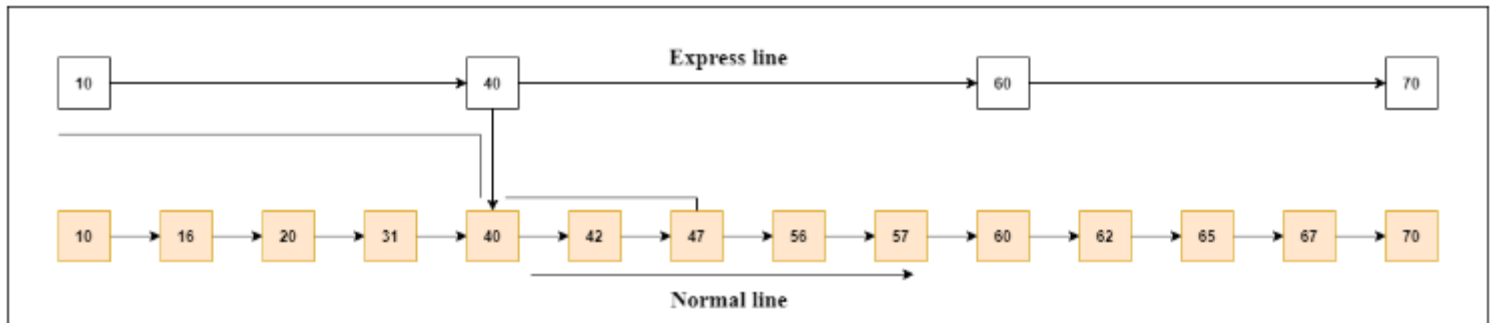
Skip List Basic Operations

There are the following types of operations in the skip list.

- o **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- o **Deletion operation:** It is used to delete a node in a specific situation.
- o **Search Operation:** The search operation is used to search a particular node in a skip list.

Algorithm of the insertion operation

1. Insertion (L, Key)
2. local update [0...Max_Level + 1]
3. a = L → header
4. **for** i = L → level down to 0 **do**.
5.   **while** a → forward[i] → key forward[i]
6. update[i] = a
7.
8. a = a → forward[0]
9. lvl = random_Level()
10. **if** lvl > L → level then
11. **for** i = L → level + 1 to lvl **do**
12.   update[i] = L → header
13.   L → level = lvl
14.
15. a = makeNode(lvl, Key, value)

16. **for** i = 0 to level **do**
17.   a → forward[i] = update[i] → forward[i]
18. update[i] → forward[i] = a

Algorithm of deletion operation

1.   Deletion (L, Key)
2.   local update [0... Max_Level + 1]
3.   a = L → header
4.   **for** i = L → level down to 0 **do**.
5.   **while** a → forward[i] → key forward[i]
6.   update[i] = a
7.   a = a → forward[0]
8.   **if** a → key = Key then
9.   **for** i = 0 to L → level **do**
10.   **if** update[i] → forward[i] ? a then **break**
11.   update[i] → forward[i] = a → forward[i]
12.   free(a)
13.   **while** L → level > 0 and L → header → forward[L → level] = NIL **do**
14.   L → level = L → level − 1

Algorithm of searching operation

1.   Searching (L, SKey)
2.   a = L → header
3.   loop invariant: a → key level down to 0 **do**.
4.   **while** a → forward[i] → key forward[i]
5.   a = a → forward[0]
6.   **if** a → key = SKey then **return** a → value
7.   **else return** failure

Advantages of the Skip list

1.   If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
2.   The skip list is simple to implement as compared to the hash table and the binary search tree.
3.   It is very simple to find a node in the list because it stores the nodes in sorted form.
4.   The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
5.   The skip list is a robust and reliable list.

Disadvantages of the Skip list

1.   It requires more memory than the balanced tree.
2.   Reverse searching is not allowed.

3. The skip list searches the node much slower than the linked list.

Applications of the Skip list

1. It is used in distributed applications, and it represents the pointers and system in the distributed applications.
2. It is used to implement a dynamic elastic concurrent queue with low lock contention.
3. It is also used with the QMap template class.
4. The indexing of the skip list is used in running median problems.
5. The skip list is used for the delta-encoding posting in the Lucene search.

Graphs – Introduction:

A **graph is an advanced data structure** that is used to organize items in an **interconnected network**. Each item in a graph is known as a **node**(or **vertex**) and these nodes are connected by **edges**.

In the figure below, we have a simple graph where there are five nodes in total and six edges.
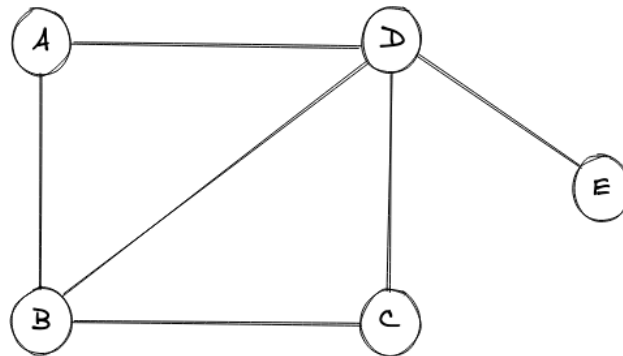
The nodes in any graph can be referred to as **entities** and the edges that connect different nodes define the **relationships between these entities**. In the above graph we have a set of nodes **{V} = {A, B, C, D, E}** and a set of edges, **{E} = {A-B, A-D, B-C, B-D, C-D, D-E}** respectively

Types of Graphs

Let's cover various different types of graphs.

1. Null Graphs

A graph is said to be null if there are no edges in that graph.

A pictorial representation of the null graph is given below:
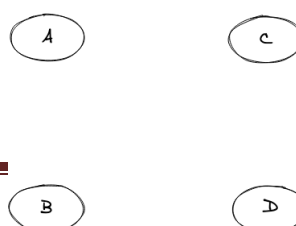
2. Undirected Graphs

If we take a look at the pictorial representation that we had in the Real-world example above, we can clearly see that different nodes are connected by a link (i.e. edge) and that edge doesn't have any kind of direction

associated with it. For example, the edge between "Anuj" and "Deepak" is bi-directional and hence the relationship between them is two ways, which turns out to be that "Anuj" knows "Deepak" and "Deepak" also knows about "Anuj". These types of graphs where the relation is bi-directional or there is not a single direction, are known as Undirected graphs.

A pictorial representation of another undirected graph is given below:
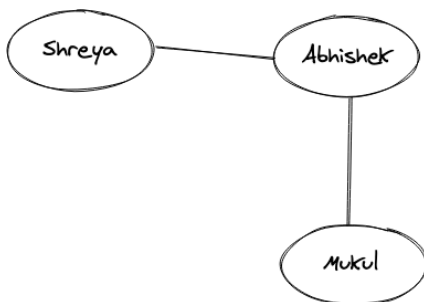
3. Directed Graphs

What if the relation between the two people is something like, "Shreya" know "Abhishek" but "Abhishek" doesn't know "Shreya". This type of relationship is one-way, and it does include a direction. The edges with arrows basically denote the direction of the relationship and such graphs are known as directed graphs.

A pictorial representation of the graph is given below:
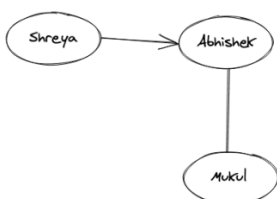
It can also be noted that the edge from "Shreya" to "Abhishek" is an outgoing edge for "Shreya" and an incoming edge for "Abhishek".

4. Cyclic Graph

A graph that contains at least one node that traverses back to itself is known as a cyclic graph. In simple words, a graph should have at least one cycle formation for it to be called a cyclic graph.

A pictorial representation of a cyclic graph is given below:
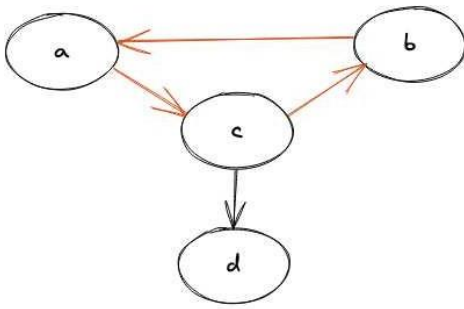
It can be easily seen that there exists a cycle between the nodes (a, b, c) and hence it is a cyclic graph.

5. Acyclic Graph

A graph where there's no way we can start from one node and can traverse back to the same one, or simply doesn't have a single cycle is known as an acyclic graph.

A pictorial representation of an acyclic graph is given below:
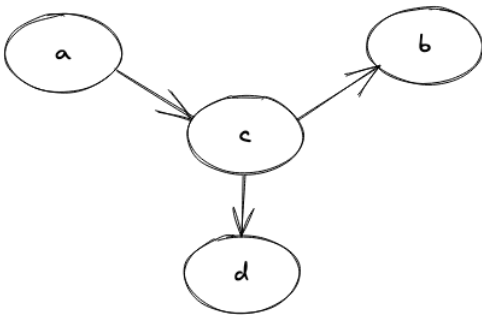
6. Weighted Graph

When the edge in a graph has some weight associated with it, we call that graph as a weighted graph. The weight is generally a number that could mean anything, totally dependent on the relationship between the nodes of that graph.

A pictorial representation of the weighted graph is given below:
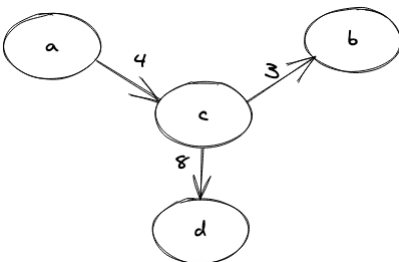
It can also be noted that if any graph doesn't have any weight associated with it, we simply call it an unweighted graph.

7. Connected Graph

A graph where we have a path between every two nodes of the graph is known as a connected graph. A path here means that we are able to traverse from a node "A" to say any node "B". In simple terms, we can say that if we start from one node of the graph we will always be able to traverse to all the other nodes of the graph from that node, hence the connectivity.

A pictorial representation of the connected graph is given below:
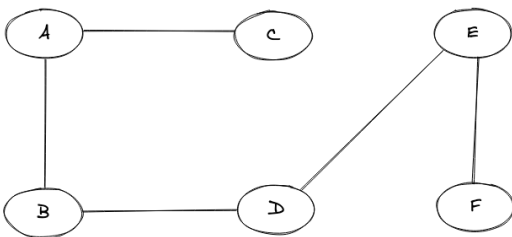
8. Disconnected Graph

A graph that is not connected is simply known as a disconnected graph. In a disconnected graph, we will not be able to find a path from between every two nodes of the graph.

A pictorial representation of the disconnected graph is given below:
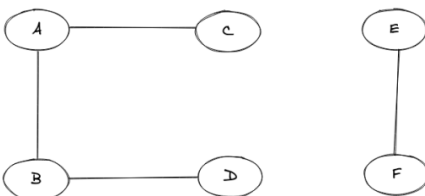
9. Complete Graph

A graph is said to be a complete graph if there exists an edge for every pair of vertices(nodes) of that graph.

A pictorial representation of the complete graph is given below:
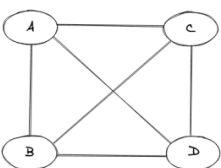
10. Multigraph

A graph is said to be a multigraph if there exist two or more than two edges between any pair of nodes in the graph.

A pictorial representation of the multigraph is given below:
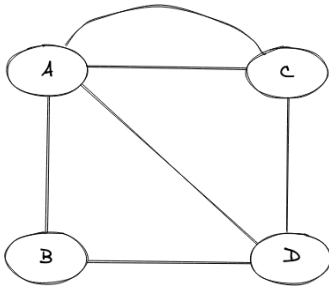
Commonly Used Graph Terminologies

- **Path** - A sequence of alternating nodes and edges such that each of the successive nodes are connected by the edge.
- **Cycle** - A path where the starting and the ending node is the same.
- **Simple Path** - A path where we do not encounter a vertex again.
- **Bridge** - An edge whose removal will simply make the graph disconnected.
- **Forest** - A forest is a graph without cycles.
- **Tree** - A connected graph that doesn't have any cycles.
- **Degree** - The degree in a graph is the number of edges that are incident on a particular node.
- **Neighbour** - We say vertex "A" and "B" are neighbours if there exists an edge between them.

Weighted vs UnWeighted Graph

To understand difference between weighted vs unweighted graph, first we need to understand what **weight** represent ?

A **weight** is a numerical value attached to each individual edge in the graph.

Weighted Graph will contains weight on each edge where as **unweighted does not.**
Following is an example, where both graphs looks exactly the same but one is weighted another is not.
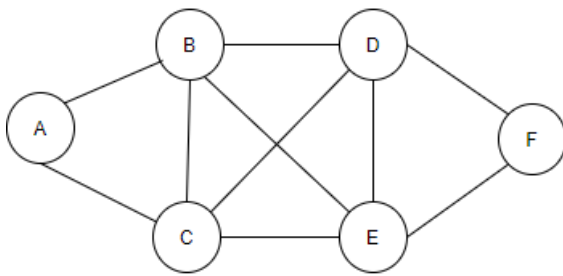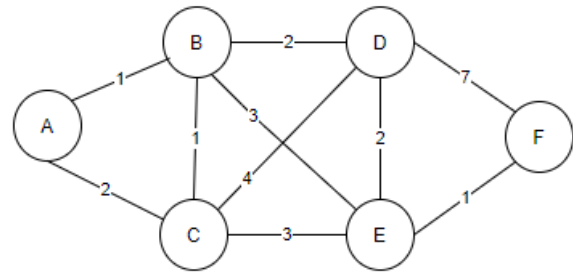
UnWeighted Graph   Weighted Graph

What difference does it make ?

Let's take the same example to **find shortest path from A to F**. Result is different, just because one is weighted another doesn't.
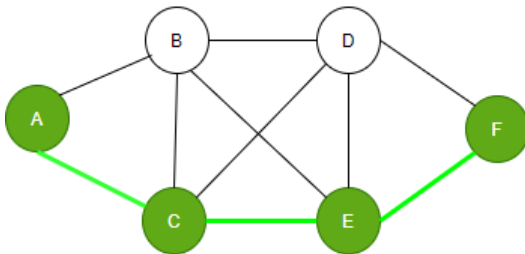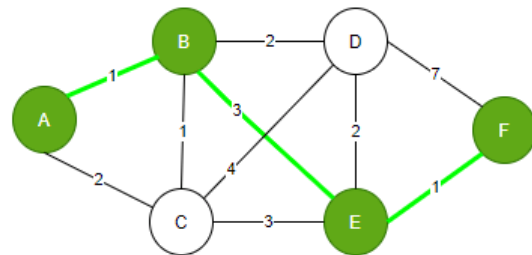
UnWeighted Graph   Weighted Graph

But how?

Because when you doesn't have weight, all edges are considered equal. Shortest distance means less number of nodes you travel.

But in case of weighted graph, calculation happens on the sum of weights of the travelled edges.

Breadth First Search:

BFS is an algorithm that is used to graph data or searching tree or traversing structures. The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion.

This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Once the algorithm visits and marks the starting node, then it moves towards the nearest unvisited nodes and analyses them.

Once visited, all nodes are marked. These iterations continue until all the nodes of the graph have been successfully visited and marked. The full form of BFS is the Breadth-first search.

Steps in Breadth First Search

1. Start by putting any one of the graph's vertices at the back of a queue.

2. Take the front item of the queue and add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

4. Keep repeating steps 2 and 3 until the queue is empty.

Example of BFS

In the following example of DFS, we have used graph having 6 vertices.

Example of BFS

Step 1)
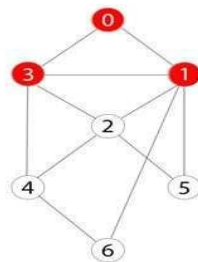
1. Mark 0 as visited
2. Insert 0 to the queue
3. Traverse the un-visited adjacent nodes which are 3 and 1

5.
You have a graph of seven numbers ranging from 0 – 6.

Step 2)
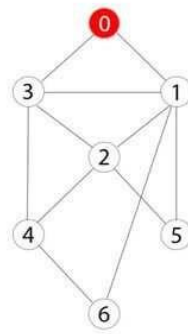
Root node = 0

6.

0 or zero has been marked as a root node.

Step 3)
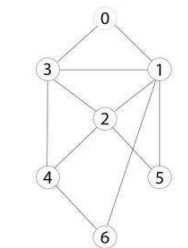
Queue:

0	1	2	3	4	5	6
0	1	2	3	4	5	6

Delete values from queue
and Print as result

Result = 0, 1, 2, 3, 4, 5, 6

7.

0 is visited, marked, and inserted into the queue data structure.

Step 4)
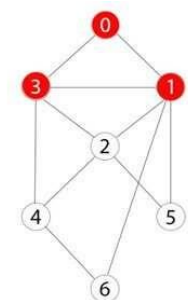
Visit 3 and 1 in any
sequence and mark them
as visited and add them
to the queue

8.

Remaining 0 adjacent and unvisited nodes are visited, marked, and inserted into the queue.

Step 5)
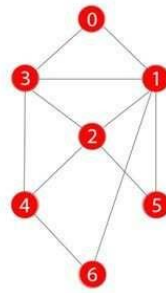
Visit all adjacent and
un-visited nodes of
the previous node
and iterate until all
visited

9.
Traversing iterations are repeated until all nodes are visited.

Depth First Search.

DFS is an algorithm for finding or traversing graphs or trees in depth-ward direction. The execution of the algorithm begins at the root node and explores each branch before backtracking. It uses a stack data structure to remember, to get the subsequent vertex, and to start a search, whenever a dead-end appears in any iteration. The full form of DFS is Depth-first search.

Steps in Depth First Search

1. Start by putting any one of the graph's vertices on top of a stack.

2. Take the top item of the stack and add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

4. Keep repeating steps 2 and 3 until the stack is empty.

Example of DFS

In the following example of DFS, we have used an undirected graph having 5 vertices.
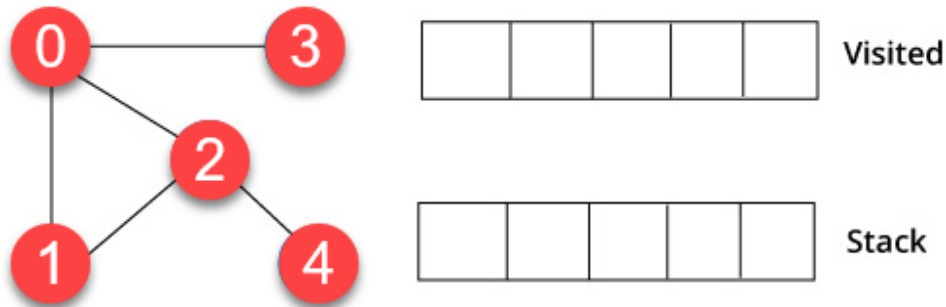
Step 1)
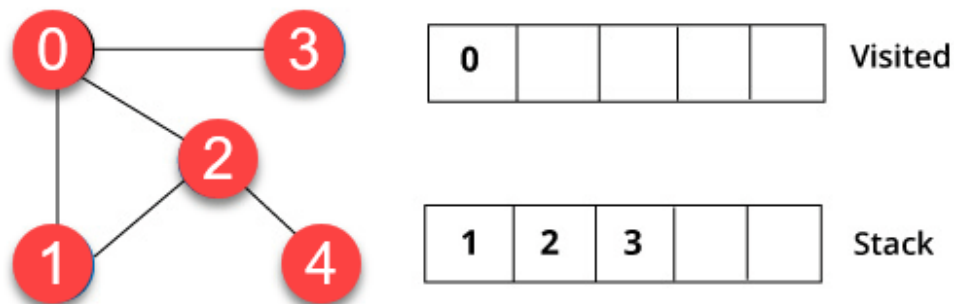
We have started from vertex 0. The algorithm begins by putting it in the visited list and simultaneously putting all its adjacent vertices in the data structure called stack.

Step 2)
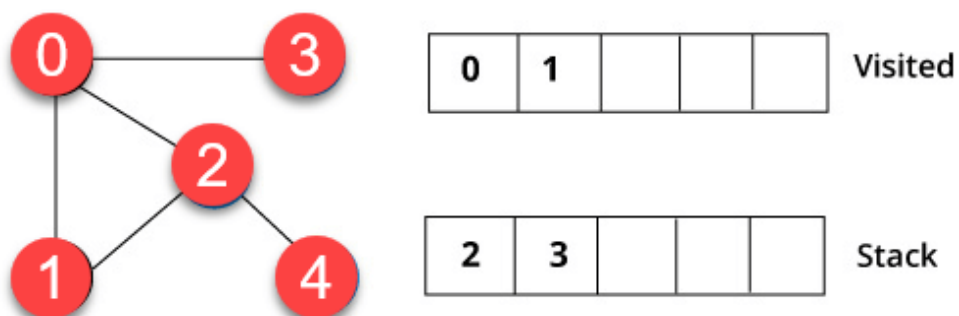
You will visit the element, which is at the top of the stack, for example, 1 and go to its adjacent nodes. It is because 0 has already been visited. Therefore, we visit vertex 2.

Step 3)
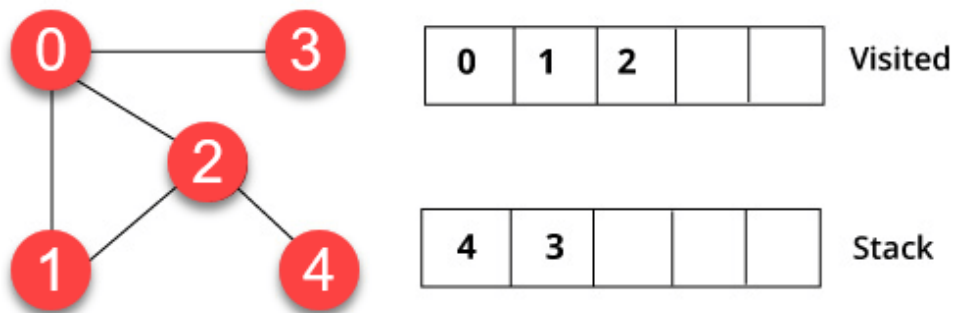
Vertex 2 has an unvisited nearby vertex in 4. Therefore, we add that in the stack and visit it.

Step 4)
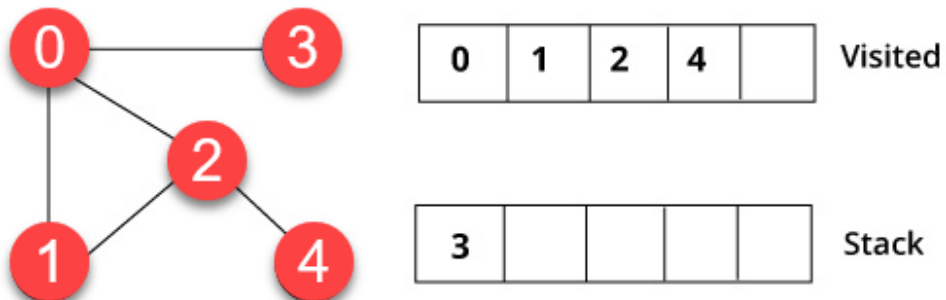
Finally, we will visit the last vertex 3, it doesn't have any unvisited adjoining nodes. We have completed the traversal of the graph using DFS algorithm.
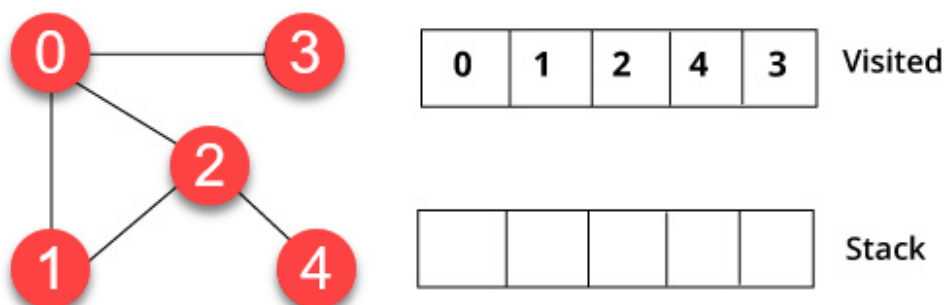
UNIT -V

Trees - Overview of Trees, Tree Terminology, Binary Trees: Introduction, Applications, Implementation. Tree Traversals, Binary Search Trees: Introduction, Implementation, AVL Trees: Introduction, Rotations, Implementation B-Trees and B+ Trees

Implementation.

Tree Data Structures

Trees are non-linear data structures that represent nodes connected by edges. Each tree consists of a root node as the Parent node, and the left node and right node as Child nodes.
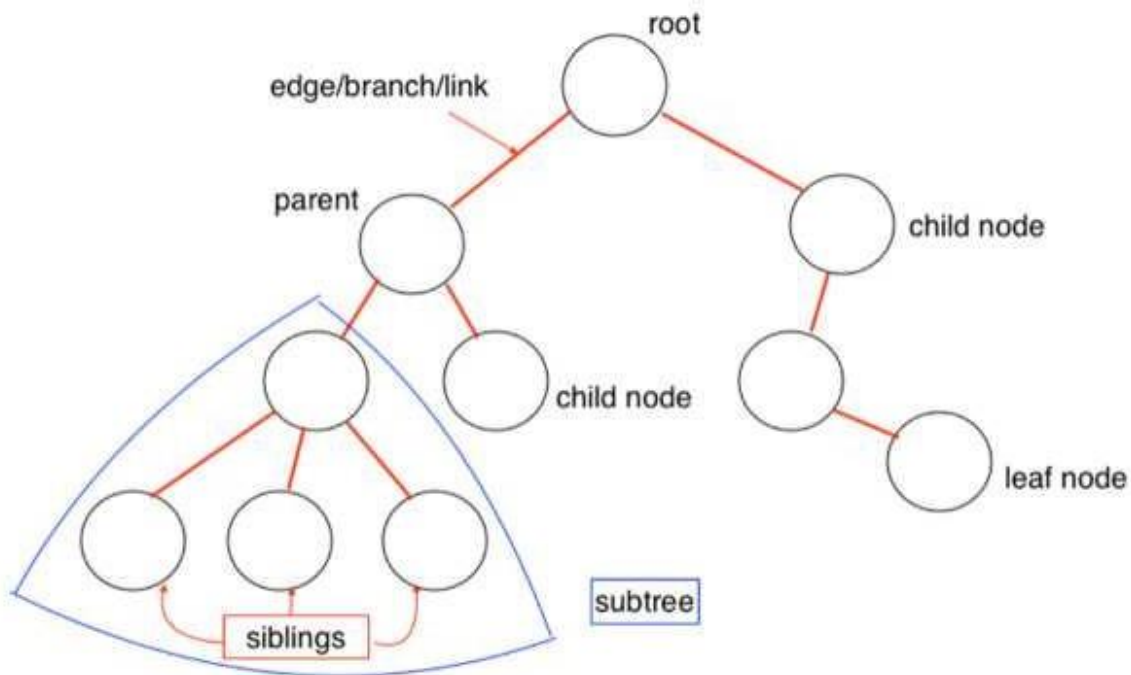
- **Root** → The topmost node of the hierarchy is called the root of the tree.
- **Child** → Nodes next in the hierarchy are the children of the previous node.
- **Parent** → The node just previous to the current node is the parent of the current node.
- **Siblings** → Nodes with the same parent are called siblings.
- **Ancestors** → Nodes which are higher in the hierarchy are ancestors of a given node.
- **Descendents** → Nodes which are lower in the hierarchy are descendants of a given node.
- **Internal Nodes** → Nodes with at least one child are internal nodes.
- **External Nodes/Leaves** → Nodes which don't have any child are called leaves of a tree.
- **Edge** → The link between two nodes is called an edge.
- **Level** → The root of a tree is at level 0 and the nodes whose parent is root are at level 1 and so on.
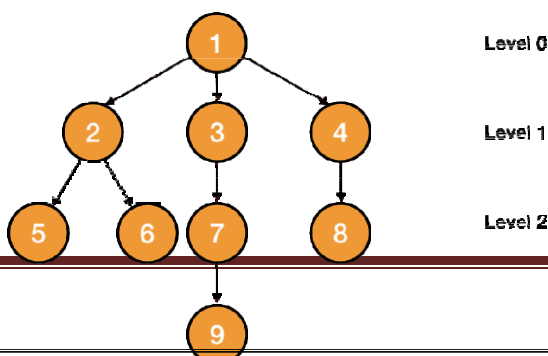
- **Height** → The height of a node is the number of nodes (excluding the node) on the longest path from the node to a leaf.
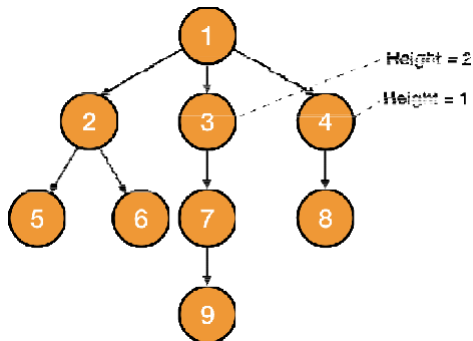
- **Height of Tree** → Height of a tree is the height of its root.
- **Depth** → The depth of a node is the number of nodes (excluding the node) on the path from the root to the node.
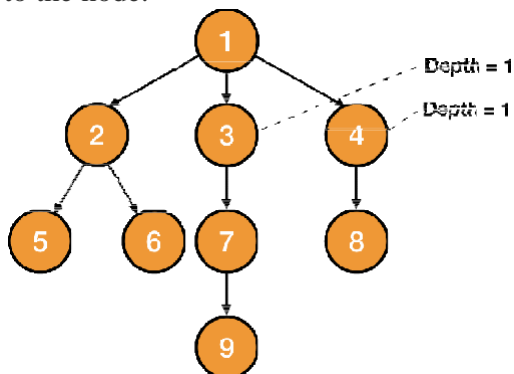
- **Node Degree** → It is the maximum number of children a node has.
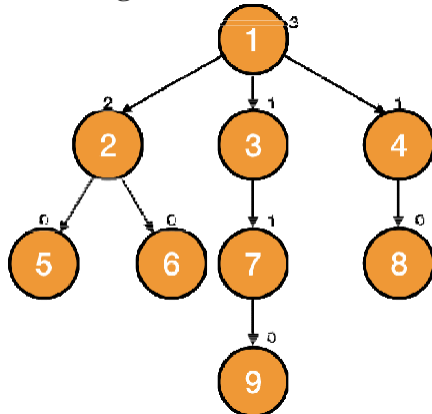
- **Tree Degree** → Tree degree is the maximum of the node degrees. So, the tree degree in the above picture is 3.

Till now, we have an idea of what a tree is and the terminologies we use with a tree. But we don't know yet what the specific properties of a tree are or which structure should qualify as a tree. So, let's see the properties of a tree.

Properties of a Tree

A tree must have some properties so that we can differentiate from other data structures. So, let's look at the properties of a tree.

The numbers of nodes in a tree must be a finite and nonempty set.

There must exist a path to every node of a tree i.e., every node must be connected to some other node.

There must not be any cycles in the tree. It means that the number of edges is one less than the number of nodes.
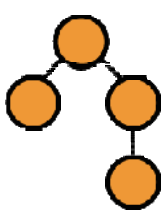
A tree
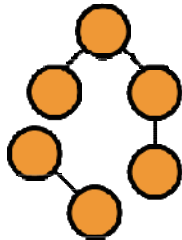
Not a tree
All nodes are not connected
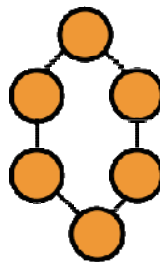
Not a tree
Cycle exists

Binary Trees

A binary tree is a tree in which every node has at most two children.
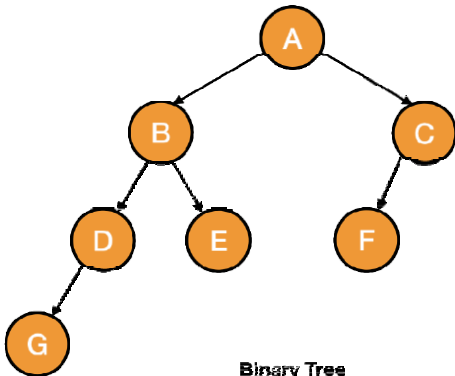
Binary Tree

As you can see in the picture given above, a node can have less than 2 children but not more than that.

We can also classify a binary tree into different categories. Let's have a look at them:

Full Binary Tree → A binary tree in which every node has 2 children except the leaves is known as a full binary tree.
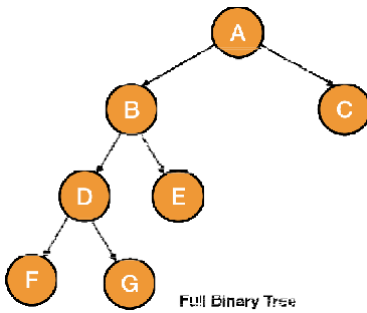
Full Binary Tree

Complete Binary Tree → A binary tree which is completely filled with a possible exception at the bottom level i.e., the last level may not be completely filled and the bottom level is filled from left to right.


Complete Binary Tree
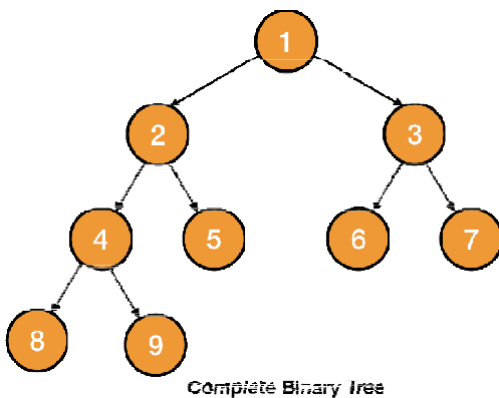
Let's look at this picture to understand the difference between a full and a complete binary tree.

Complete and Full

Full but NOT Complete

Complete but NOT Full

Neither Complete nor Full
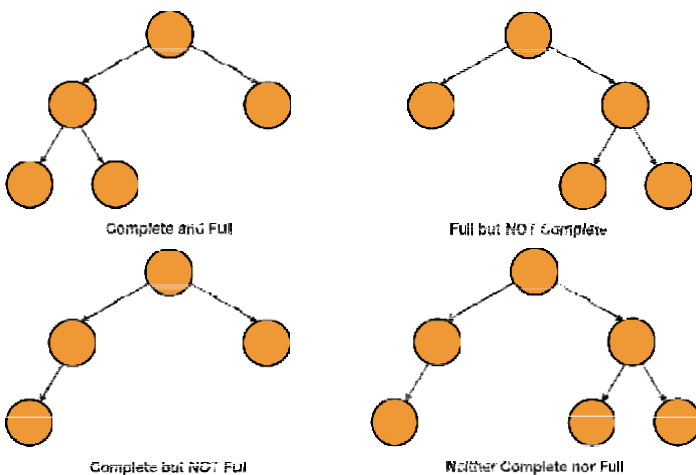
A complete binary tree also holds some important properties. So, let's look at them.

- The **parent of node i** is ⌊i2⌋⌊i2⌋. For example, the parent of node 4 is 2 and the parent of node 5 is also 2.
- The **left child of node i** is 2i2i.
- The **right child of node i** is 2i+12i+1
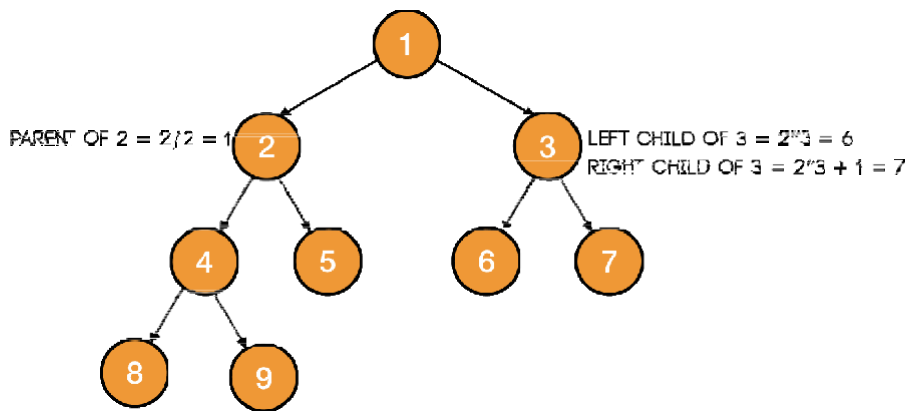
Perfect Binary Tree → In a perfect binary tree, each leaf is at the same level and all the interior nodes have two children.

Perfect Binary Tree
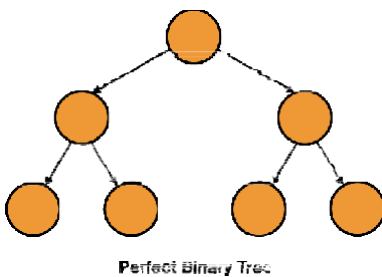
Thus, a perfect binary tree will have the maximum number of nodes for all alternative binary trees of the same height and it will be $2h+1-12h+1-1$ which we are going to prove next.

Maximum Number of Nodes in a Binary Tree

We know that the maximum number of nodes will be in a perfect binary tree. So, let's assume that the height of a perfect binary tree is hh.
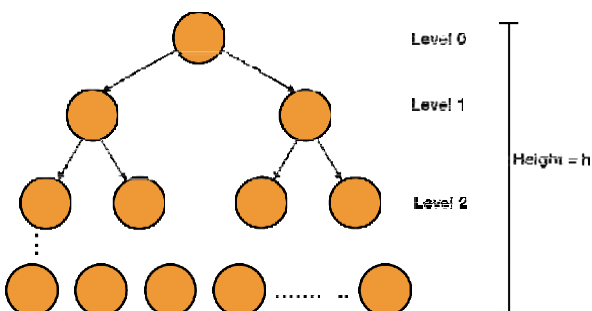
Number of nodes at level 0 = 20=120=1

Number of nodes at level 1 = 21=221=2

Similarly, the number of nodes at level h = 2h2h

Thus, the total number of nodes in the tree = 20+21+...+2h20+21+...+2h

The above sequence is a G.P. with common ratio 2 and first term 1 and total number of terms are h+1. So, the value of the summation will be 2h+1−12−1=2h+1−12h+1−12−1=2h+1−1.

Thus, the **total number of nodes in a perfect binary tree** = 2h+1−12h+1−1.

Height of a Perfect Binary Tree

We know that the number of nodes (n) for height (h) of a perfect binary tree = 2h+1−12h+1−1.

=>n=2∗2h−1=>n=2∗2h−1

or,2h=n+12or,2h=n+12

h=lgn+12h=lg n+12

Thus, the height of a perfect binary tree with n nodes = lgn+12lg n+12.
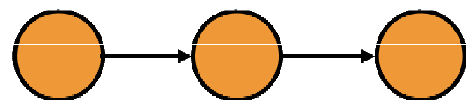
We know that the number of nodes at level i in a perfect binary tree = 2i2i. Thus, the **number of leaves** (nodes at level h) = 2h2h.

Thus, the total **number of non-leaf nodes** = 2h+1−1−2h=2h−12h+1−1−2h=2h−1 i.e., number of leaf nodes - 1.

Thus, the maximum number of nodes will be in a perfect binary tree and the minimum number of nodes will be in a tree in which nodes are linked just like a linked list.
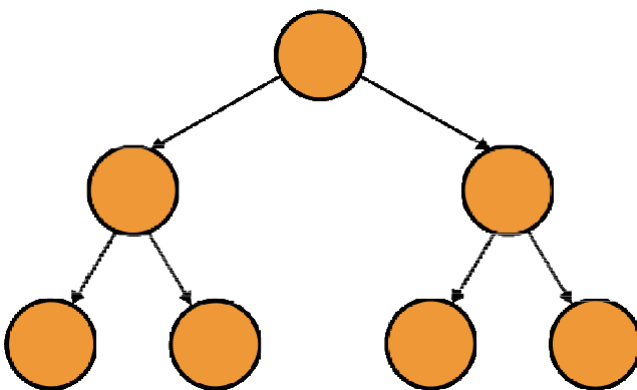

Maximum number of nodes for height 2   **Minimum number of nodes for height 2**

Array Representation of Binary Tree

In the previous chapter, we have already seen to make a node of a tree. We can easily use those nodes to make a linked representation of a binary tree. For now, let's discuss the array representation of a binary tree.

We start by numbering the nodes of the tree from 1 to n(number of nodes).
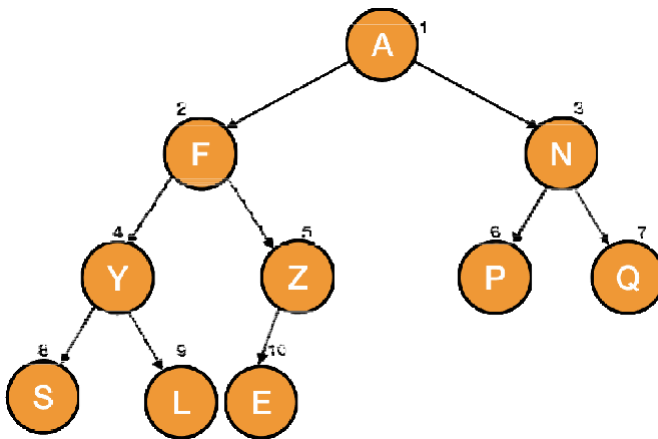
As you can see, we have numbered from top to bottom and left to right for the same level. Now, these numbers represent the indices of an array (starting from 1) as shown in the picture given below.

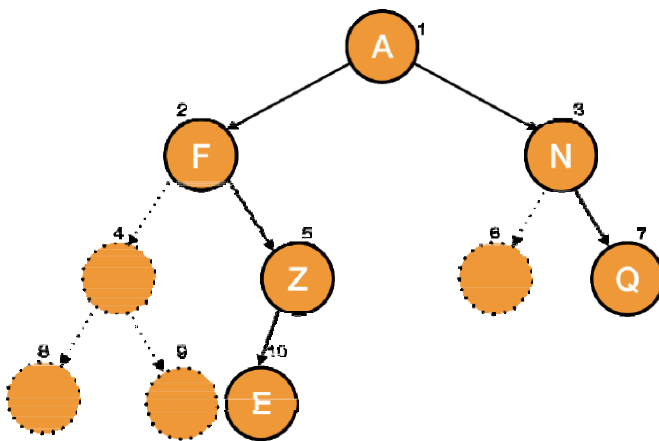**Array Representation of Binary Tree**

We can also get the parent, the right child and the left child using the properties of a complete binary tree we have discussed above i.e., for a node i, the parent is $[i2][i2]$, the left child is $2i2i$ and the right child is $2i+12i+1$.

LEFT CHILD OF 3 = 2*3 = 6
RIGHT CHILD OF 3 = 2*3 + 1 = 7
PARENT OF 2 = 2/2 = 1
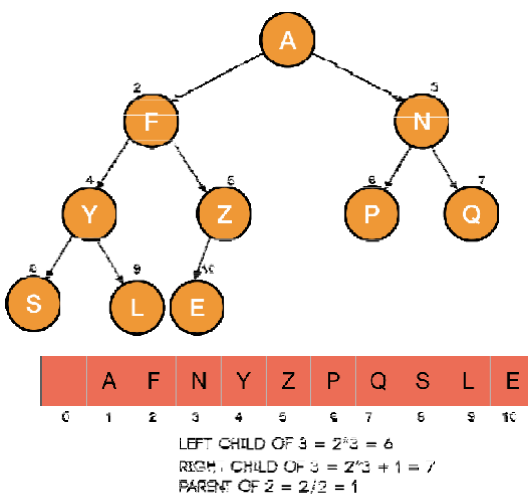
So, we represented a complete binary tree using an array and saw how to get the parent and children of any node. Let's discuss about doing the same for an incomplete binary tree.

Array Representation of Incomplete Binary Tree

To represent an incomplete binary tree with an array, we first assume that all the nodes are present to make it a complete binary tree and then number the nodes as shown in the picture given below.

Now according to these numbering, we fill up the array.

A	F	N		Z		Q			E
0	1	2	3	4	5	6	7	8	9	10

Coding a Binary Tree

For the linked representation, we can easily access the parent, right child and the left child with T.parent, T.right and T.left respectively.

So, we will first write explain the codes for the array representation and then write the full codes in C, Java and Python for both array and linked representation.

Let's start by writing the code to get the right child of a node. We will pass the index and the tree to the function - RIGHT_CHILD(index).

After this, we will check if there is a node at the index or not (if (T[index] != null)) and also if the index of the right child (2∗index+12∗index+1) lies in the size of the tree or not i.e., if (T[index] != null and (2*index + 1) <= T.size).

If the above condition is true, we will return the index of the right child i.e., return (2*index + 1).

RIGHT_CHILD(index)

if (T[index] != null and (2*index + 1) <= T.size)

return (2*index + 1)

else

return null

Similarly, we can get the left child.

LEFT_CHILD(index)

if (T[index] != null and (2*index) <= T.size)

return (2*index)

else

return null

Similarly, we can also write the code to get the parent.

PARENT(index)

if (T[index] != null and (floor(index/2)) =! null)

return floor(index/2)

else

return null

Code Using Array

```
'''

          D
         / \
        /   \
       /     \
      A       F
     / \     / \
    /   \   /   \
   E     B R     T
  / \     /     / \
 G   Q   V     J   L
'''


complete_node=15

tree=[None,'D','A','F','E','B','R','T','G','Q',None,None,'V',None,'J','L']

def get_right_child(index):
# node is not null
# and the result must lie within the number of nodes for a complete binary tree
if tree[index]!=None and ((2*index)+1)<=complete_node:
return (2*index)+1
# right child doesn't exist
```

```
return -1

def get_left_child(index):
    # node is not null
    # and the result must lie within the number of nodes for a complete binary tree
    if tree[index] != None and (2*index) <= complete_node:
        return 2*index
    # left child doesn't exist
    return -1

def get_parent(index):
    if tree[index] != None and index/2 != None:
        return index//2
    return -1
```
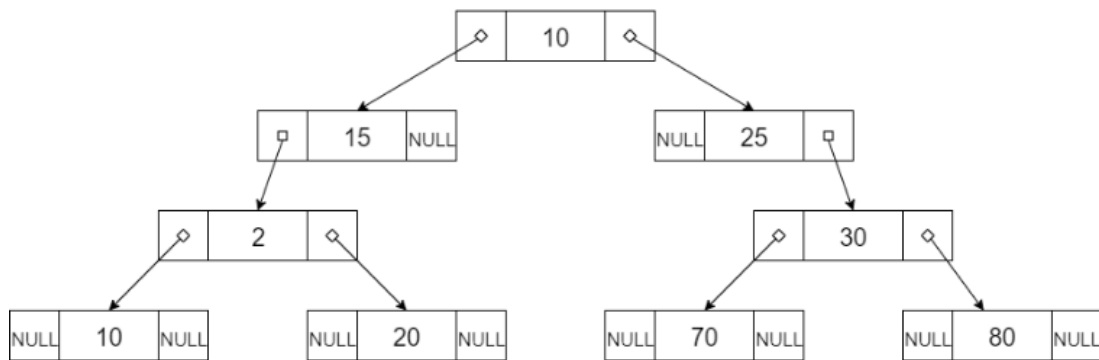
Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
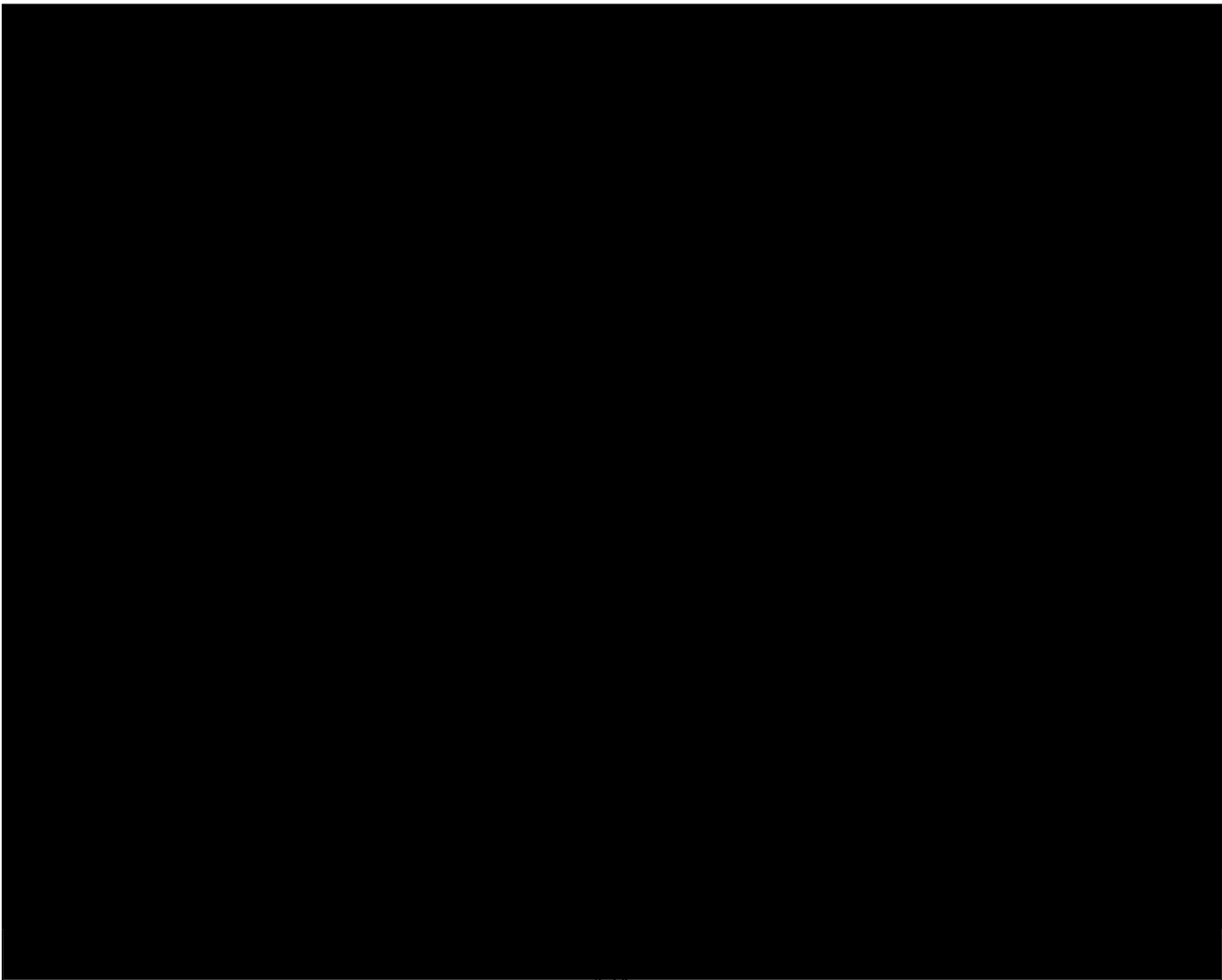In this linked list representation, a node has the following structure...
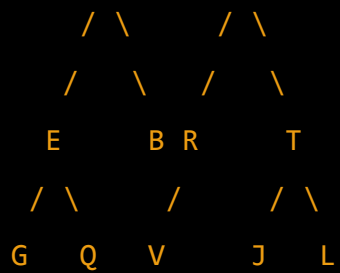
LINKED REPRESENTATION

Code Using Linked Representation

```
      / \       / \
     /   \     /   \
    E      B R       T
   / \       /     / \
  G   Q     V     J   L
'''


d=TreeNode('D')
a=TreeNode('A')
f=TreeNode('F')
e=TreeNode('E')
b=TreeNode('B')
r=TreeNode('R')
t1=TreeNode('T')
g=TreeNode('G')
q=TreeNode('Q')
v=TreeNode('V')
j=TreeNode('J')
l=TreeNode('L')


t=Tree(d)
```
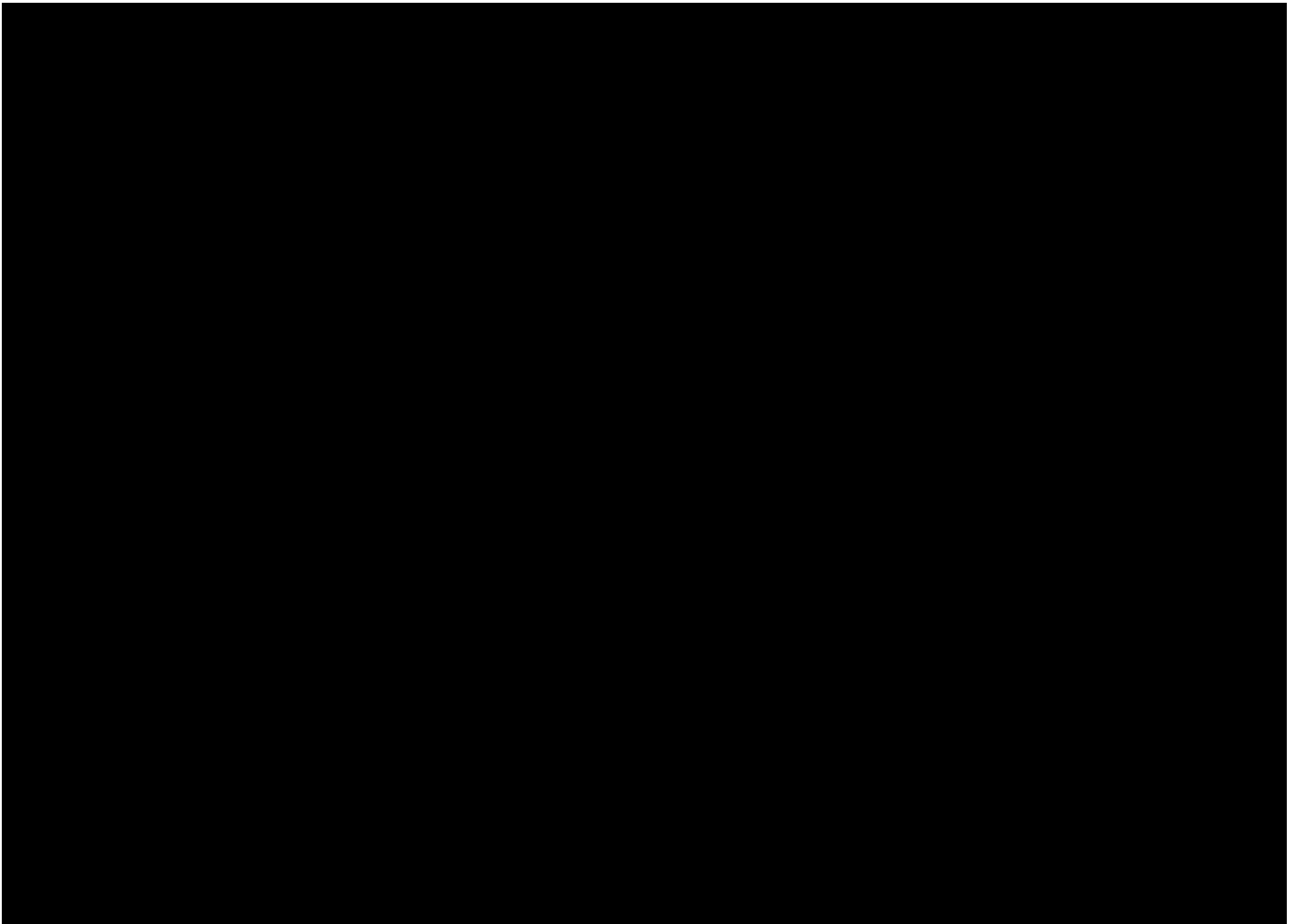
```
t.root.right=f
t.root.left=a


'''


     D
    / \
   /   \
```

Applications of Binary tree

Binary trees are used to represent a nonlinear data structure. There are various forms of Binary trees. Binary trees play a vital role in a software application. One of the most important applications of the Binary tree is in the searching algorithm.

A **general tree** is defined as a nonempty finite set T of elements called nodes such that:

- The tree contains the root element
- The remaining elements of the tree form an ordered collection of zeros and more disjoint trees T_1, T_2, T_3, T_4 …. T_n which are called subtrees.

Binary Tree Traversal

We are ready with a binary tree. Our next task would be to visit each node of it i.e., to traverse over the entire tree. In a linear data structure like linked list, it was a simple task, we just had to visit the next pointer of the node. But since a tree is a non-linear data structure, we follow different approaches. Generally, there are three types of traversals:

- Preorder Traversal

- Postorder Traversal
- Inorder Traversal

Basically, each of these traversals gives us a sequence in which we should visit the nodes. For example, in preorder traversal we first visit the root, then the left subtree and then the right subtree. Each traversal is useful in solving some specific problems. So, we choose the method of traversal accroding to the need of the problem we are going to solve. Let's discuss each of them one by one.

Preorder Traversal

In preorder traversal, we first visit the root of a tree, then its left subtree and after visiting the left subtree, the right subtree.

PREORDER(n)

 if(n != null)

   print(n.data) // visiting root

   PREORDER(n.left) // visiting left subtree
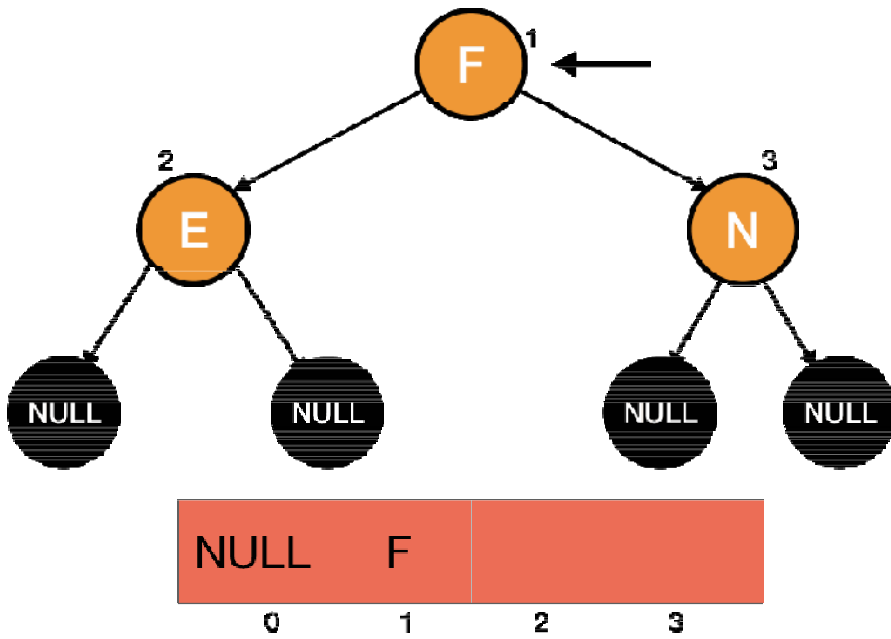
   PREORDER(n.right) // visiting right subtree

So, we are first checking if the node is null or not - if(n != null).

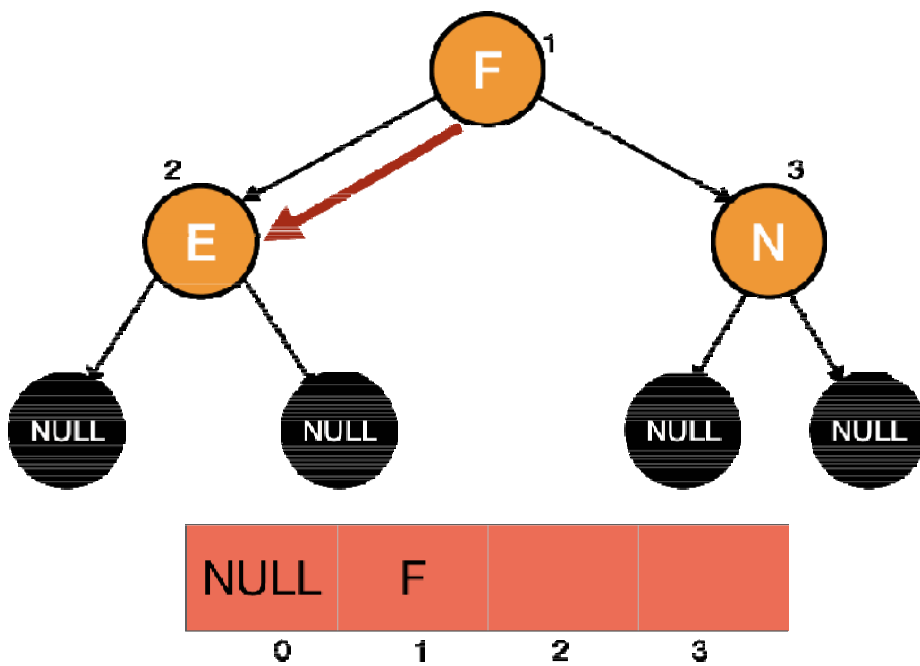After this, we are visiting the root i.e., printing its data - print(n.data).

Then we are visiting the left subtree - PREORDER(n.left).

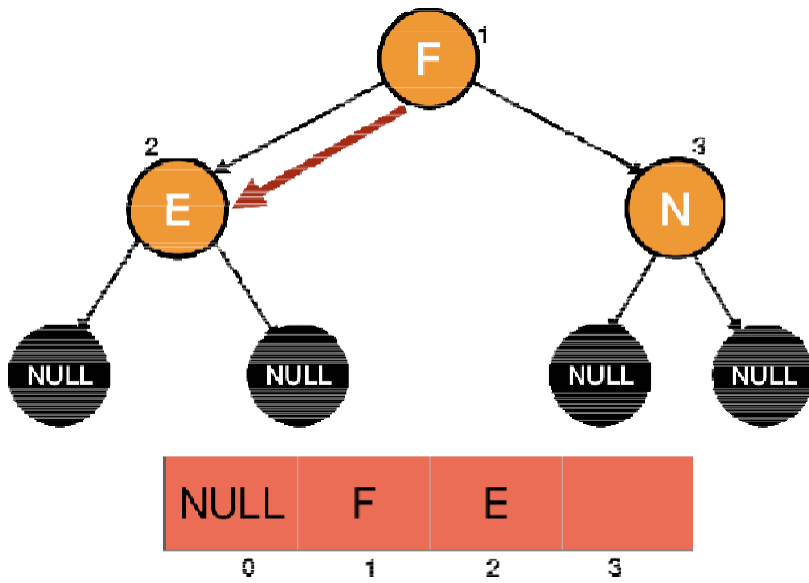At last, we are visiting the right subtree - PREORDER(n.right).

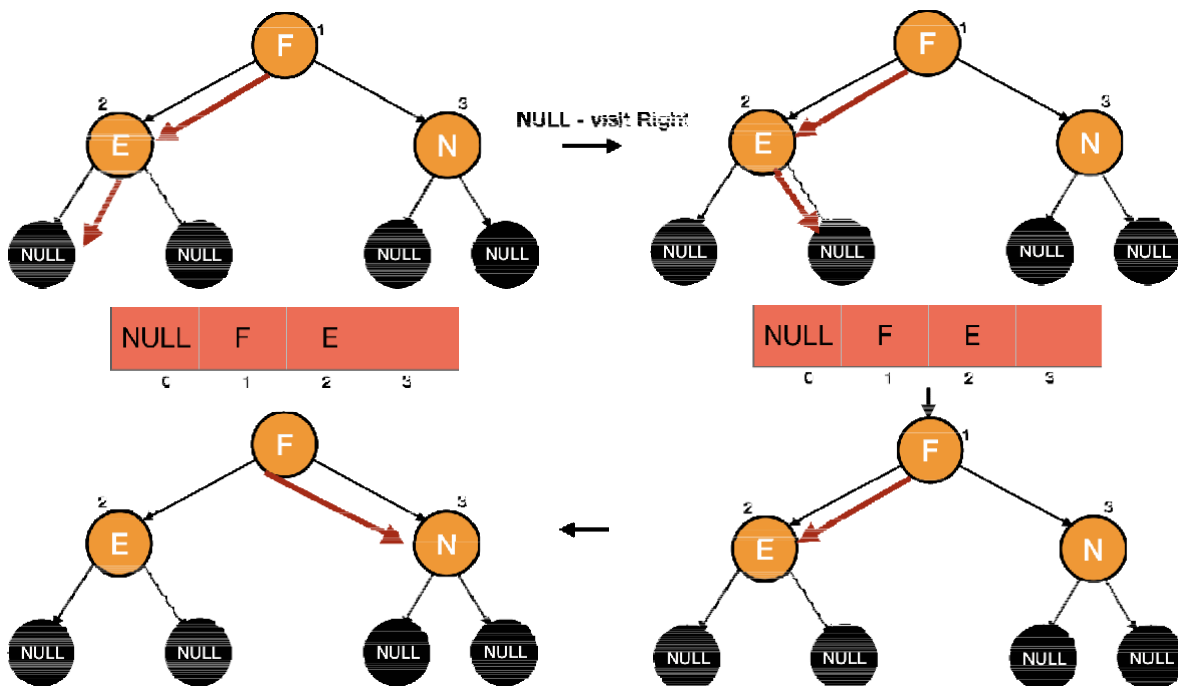So, we will first visit the root as shown in the picture given below.
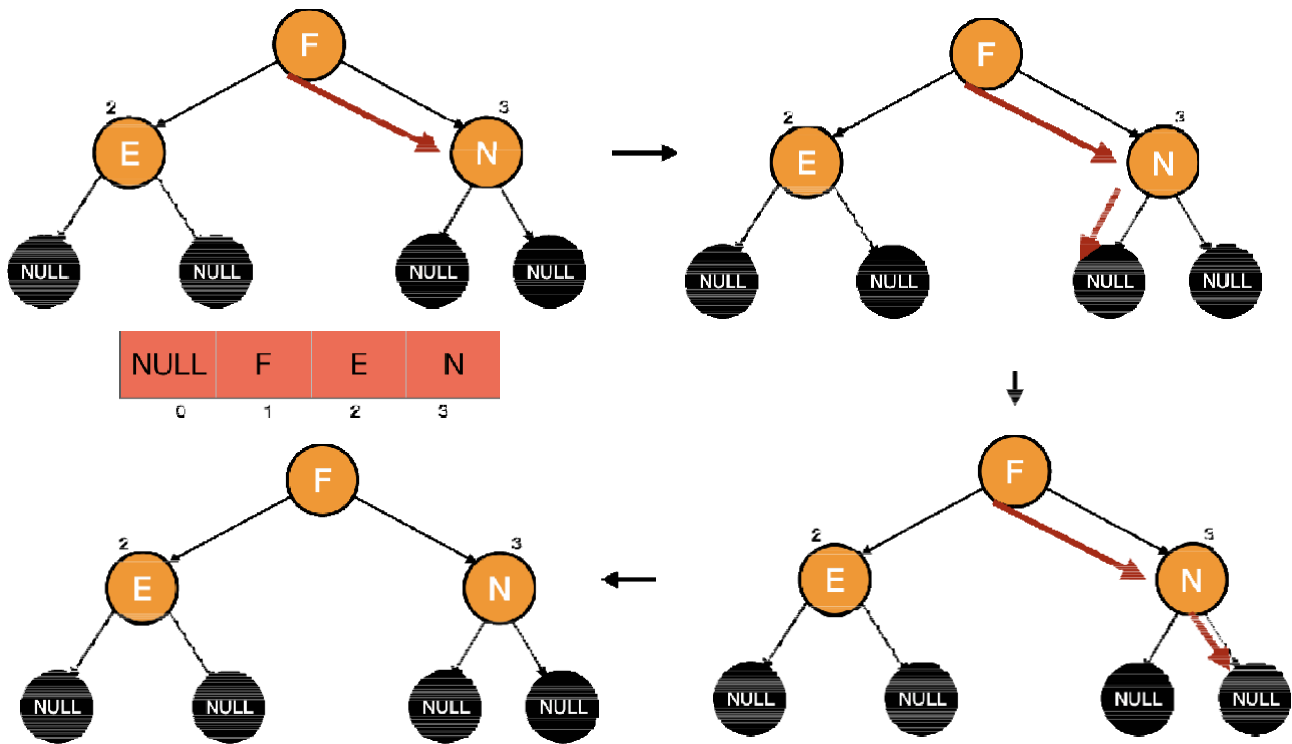
Then, we will visit the left subtree.

In this left subtree, again we will visit its root and then its left subtree.

At last, we will visit the right subtree.

Postorder Traversal

In postorder traversal, we first visit the left subtree, then the right subtree and at last, the root.
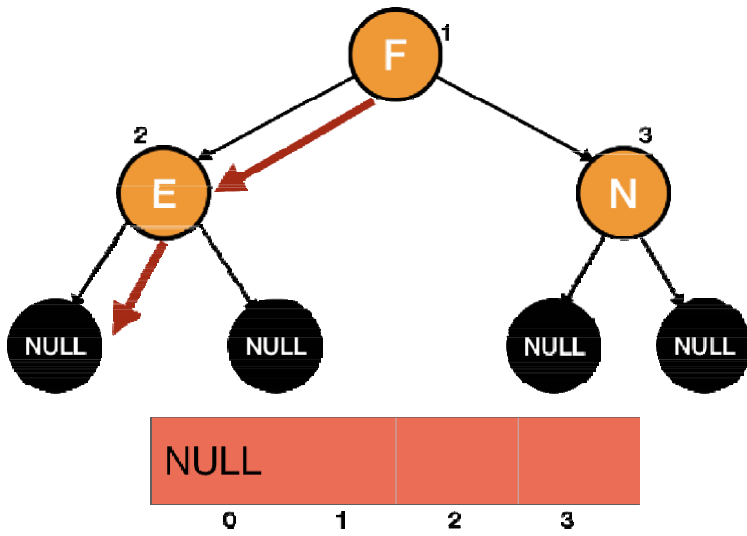
POSTORDER(n)

 if(n != null)

   PREORDER(n.left) // visiting left subtree
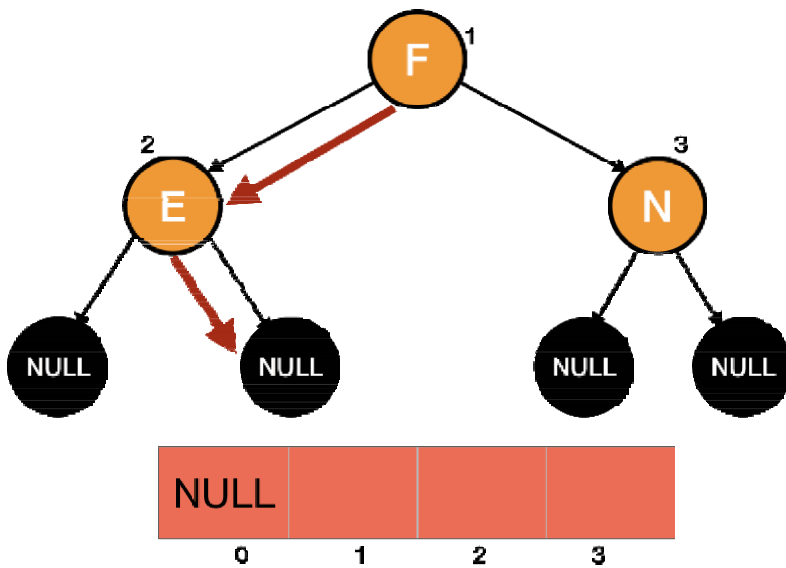
   PREORDER(n.right) // visiting right subtree

   print(n.data) // visiting root

We will first visit the left subtree.

When there is no left subtree, we will visit the right subtree.

Since the right subtree is null, we will visit the root.

NULL	E		
0	1	2	3
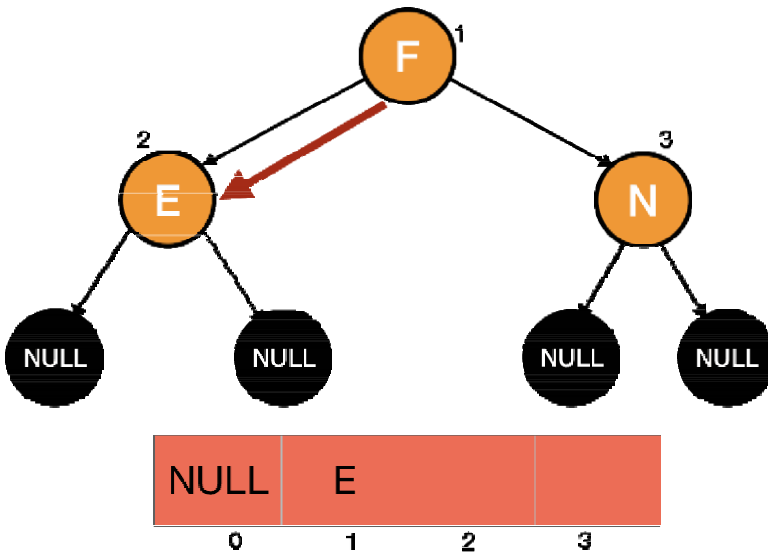
Similarly, we will visit every other node.

NULL	E	N	F
0	1	2	3
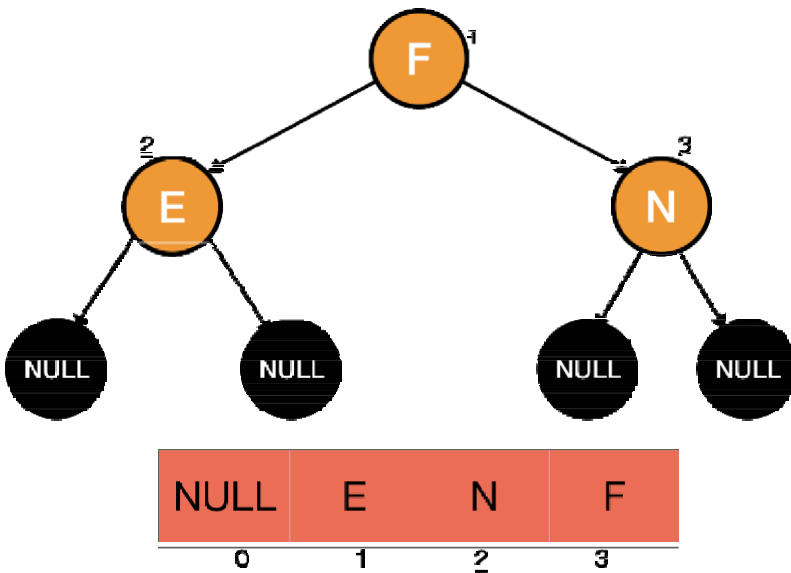
Inorder Traversal

In inorder traversal, we first visit the left subtree, then the root and lastly, the right subtree.

INORDER(n)

 if(n != null)

   INORDER(n.left)

   print(n.data)

INORDER(n.right)

NULL	E	F	N
0	1	2	3
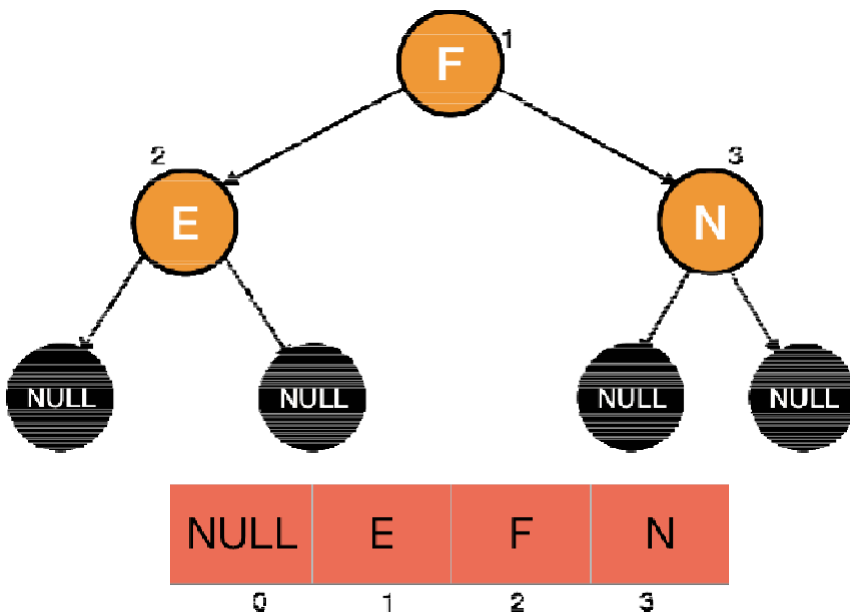
We can also see the inorder traversal as projection of the tree on an array as shown in the picture given below.

NULL	E	F	N
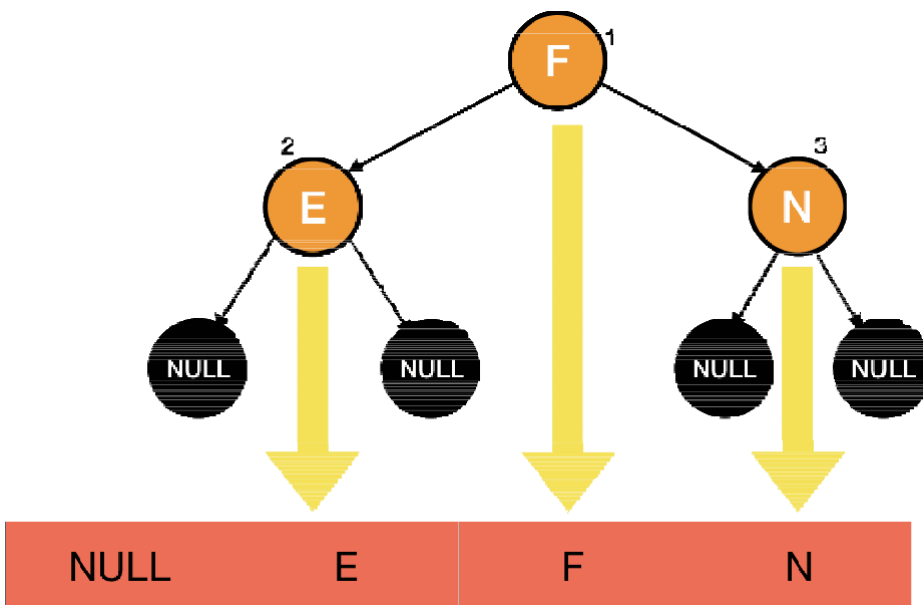
Binary Search Tree

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

A binary tree has the following time complexities...

1. **Search Operation - O(n)**
2. **Insertion Operation - O(1)**
3. **Deletion Operation - O(n)**

To enhance the performance of binary tree, we use a special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...
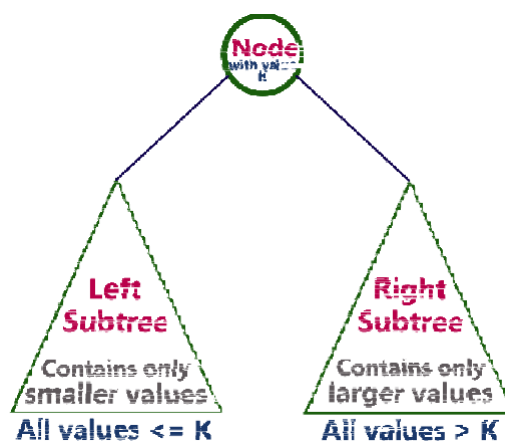
Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.
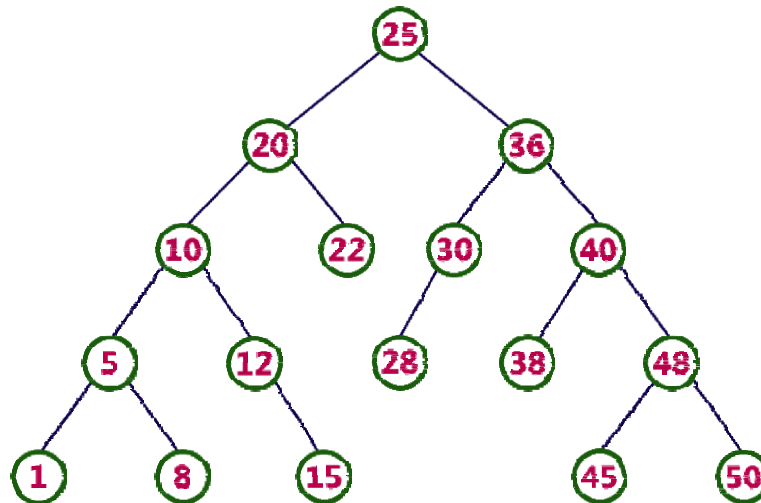

Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- **Step 1 -** Read the search element from the user.

- **Step 2 -** Compare the search element with the value of root node in the tree.

- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function

- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.

- **Step 5 -** If search element is smaller, then continue the search process in left subtree.

- **Step 6-** If search element is larger, then continue the search process in right subtree.

- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node

- **Step 8 -** If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1 -** Create a newNode with given value and set its **left** and **right** to **NULL**.

- **Step 2 -** Check whether tree is Empty.

- **Step 3 -** If the tree is **Empty**, then set **root** to **newNode**.

- **Step 4 -** If the tree is **Not Empty**, then check whether the value of newNode is **smaller** or **larger** than the node (here it is root node).

- **Step 5 -** If newNode is **smaller** than **or equal** to the node then move to its **left** child. If newNode is **larger** than the node then move to its **right** child.

- **Step 6-** Repeat the above steps until we reach to the **leaf** node (i.e., reaches to NULL).

- **Step 7 -** After reaching the leaf node, insert the newNode as **left child** if the newNode is **smaller or equal** to that leaf node or else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree includes following three cases...

- **Case 1:** Deleting a Leaf node (A node with no children)
- **Case 2:** Deleting a node with one child
- **Case 3:** Deleting a node with two children

Case 1: **Deleting a leaf node**

We use the following steps to delete a leaf node from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: **Deleting a node with one child**

We use the following steps to delete a node with one child from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has only one child then create a link between its parent node and child node.
- **Step 3 -** Delete the node using **free** function and terminate the function.

Case 3: **Deleting a node with two children**

We use the following steps to delete a node with two children from BST...

- **Step 1 - Find** the node to be deleted using **search operation**
- **Step 2 -** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3 - Swap** both **deleting node** and node which is found in the above step.
- **Step 4 -** Then check whether deleting node came to **case 1** or **case 2** or else goto step 2
- **Step 5 -** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6-** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7 -** Repeat the same process until the node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...


```
classNode:

def __init__(self,data):
```
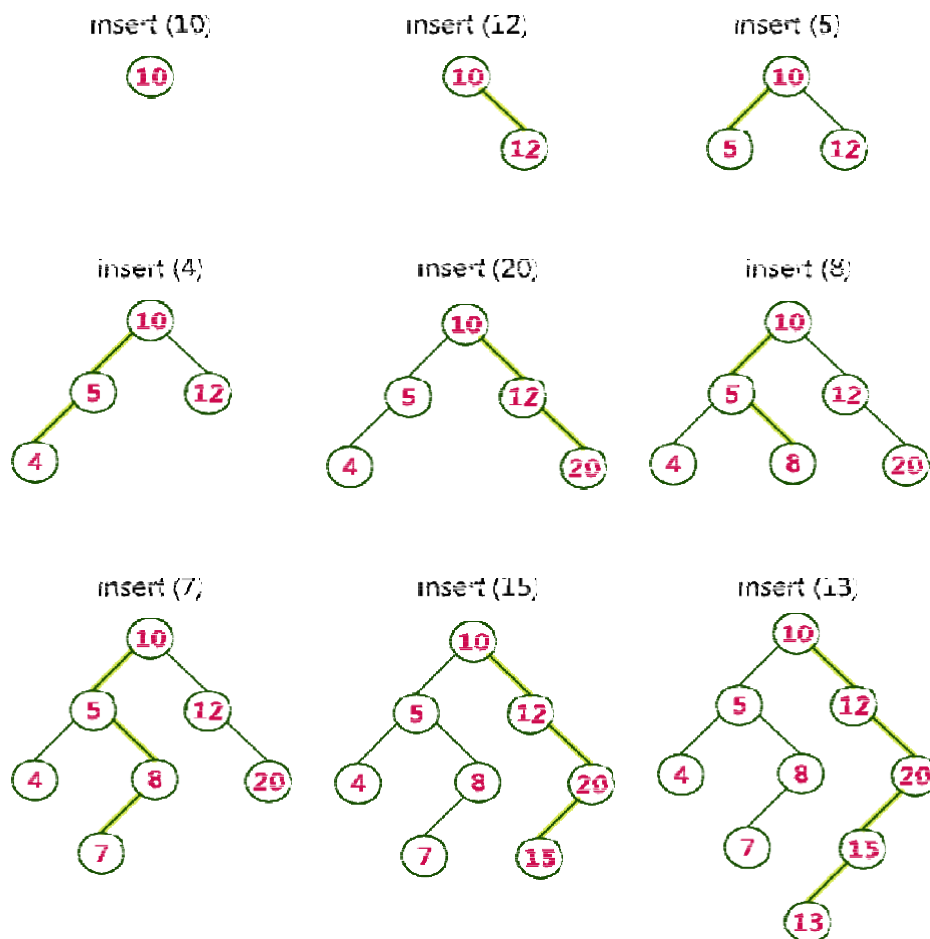
```python
        self.data=data
        self.right=None
        self.left=None
        self.parent=None

    class BinarySearchTree:
        def __init__(self):
            self.root=None

        def minimum(self,x):
            while x.left!=None:
                x=x.left
            return x

        def insert(self,n):
            y=None
            temp=self.root
            while temp!=None:
                y=temp
                if n.data<temp.data:
                    temp=temp.left
                else:
                    temp=temp.right

            n.parent=y

            if y==None:#newly added node is root
                self.root=n
            elif n.data<y.data:
                y.left=n
```

```python
else:
    y.right=n

def transplant(self,u,v):
    if u.parent==None:
        self.root=v
    elif u==u.parent.left:
        u.parent.left=v
    else:
        u.parent.right=v

    if v!=None:
        v.parent=u.parent

def delete(self,z):
    if z.left==None:
        self.transplant(z,z.right)

    elif z.right==None:
        self.transplant(z,z.left)

    else:
        y=self.minimum(z.right)#minimum element in right subtree
        if y.parent!=z:
            self.transplant(y,y.right)
            y.right=z.right
            y.right.parent=y

        self.transplant(z,y)
        y.left=z.left
```

```python
    y.left.parent=y

    def inorder(self,n):
    if n!=None:
    self.inorder(n.left)
    print(n.data)
    self.inorder(n.right)

    if __name__=='__main__':
    t=BinarySearchTree()

    a=Node(10)
    b=Node(20)
    c=Node(30)
    d=Node(100)
    e=Node(90)
    f=Node(40)
    g=Node(50)
    h=Node(60)
    i=Node(70)
    j=Node(80)
    k=Node(150)
    l=Node(110)
    m=Node(120)

    t.insert(a)
    t.insert(b)
    t.insert(c)
    t.insert(d)
    t.insert(e)
```

```
t.insert(f)

t.insert(g)

t.insert(h)

t.insert(i)

t.insert(j)

t.insert(k)

t.insert(l)

t.insert(m)


t.delete(a)

t.delete(m)


t.inorder(t.root)
```

AVL Tree

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example of AVL Tree
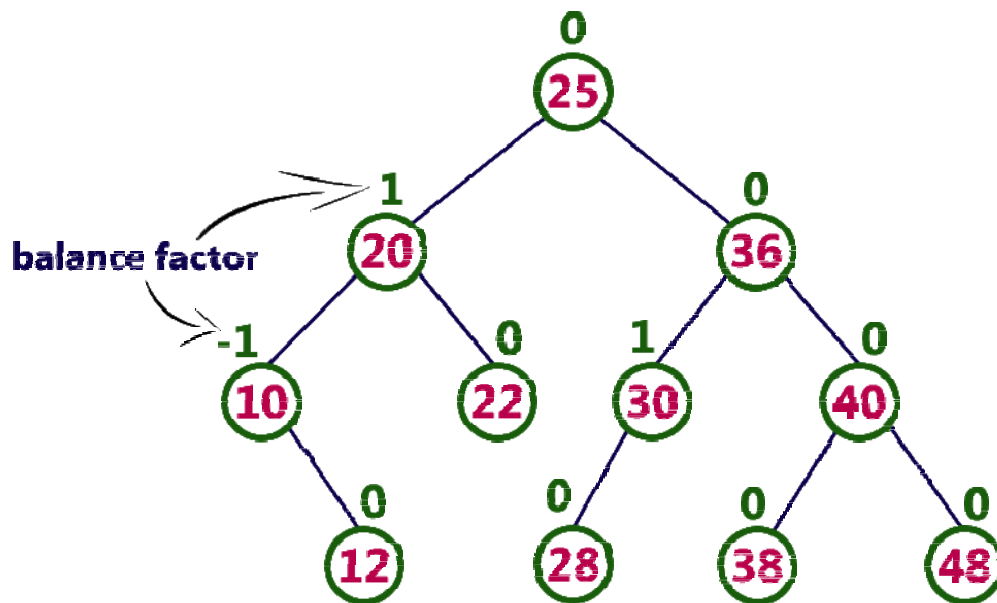
The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.
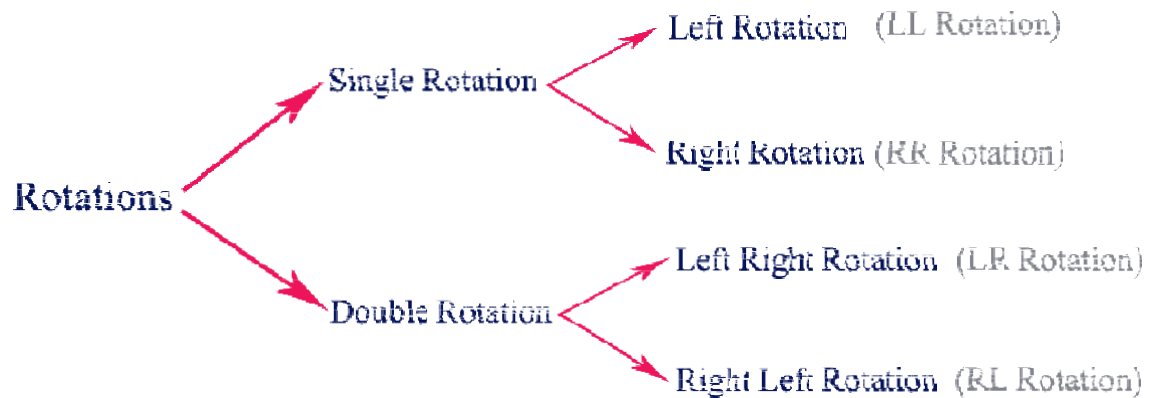
AVL Tree Rotations

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations   to   make   the   tree   balanced.

Rotation operations are used to make the tree balanced.

Rotation is the process of moving nodes either to left or to right to make the tree balanced.
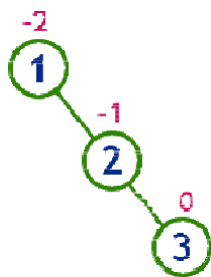
There are **four** rotations and they are classified into **two** types.
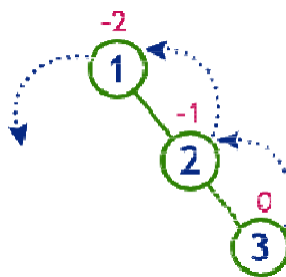

Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

Tree is imbalanced

To make balanced we use LL Rotation which moves nodes one position to left

After LL Rotation Tree is Balanced

Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...

Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...

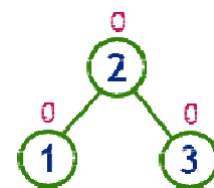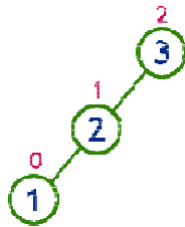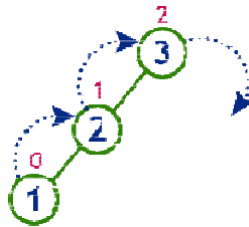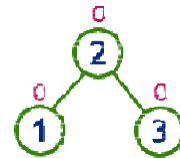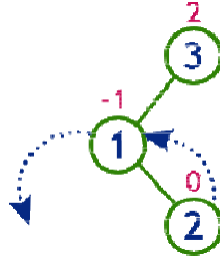insert 1, 3 and 2
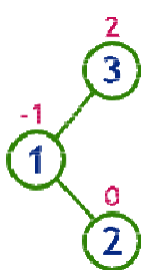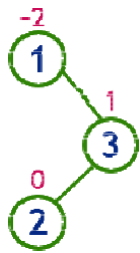
Tree is imbalanced
because node 1 has balance factor -2
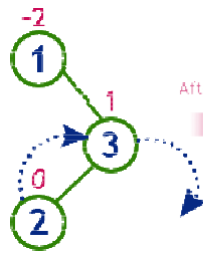
RR Rotation

After RR Rotation
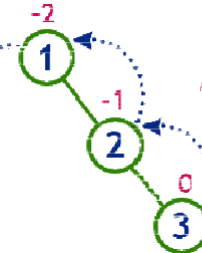
LL Rotation

After LL Rotation
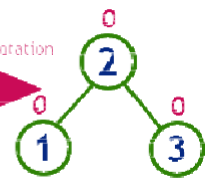
After RL Rotation
Tree is Balanced

Operations on an AVL Tree

The following operations are performed on AVL tree...

1. **Search**

2. **Insertion**

3. **Deletion**

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.

- **Step 6 -** If search element is larger, then continue the search process in right subtree.

- **Step 7 -** Repeat the same until we find the exact element or until the search element is compared with the leaf node.

- **Step 8 -** If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.

- **Step 9 -** If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.

- **Step 2 -** After insertion, check the **Balance Factor** of every node.

- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

- **Step 4 -** If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.

insert 1

Tree is balanced

insert 2

Tree is balanced

insert 3

Tree is imbalanced

LL Rotation

Tree is balanced

insert 4

Tree is balanced

insert 5

Tree is imbalanced

LL Rotation at 3

Tree is balanced

insert 6

Tree is imbalanced

LL Rotation at 2

Tree is balanced

insert 7

Tree is imbalanced

LL Rotation at 5

Tree is balanced

insert 8
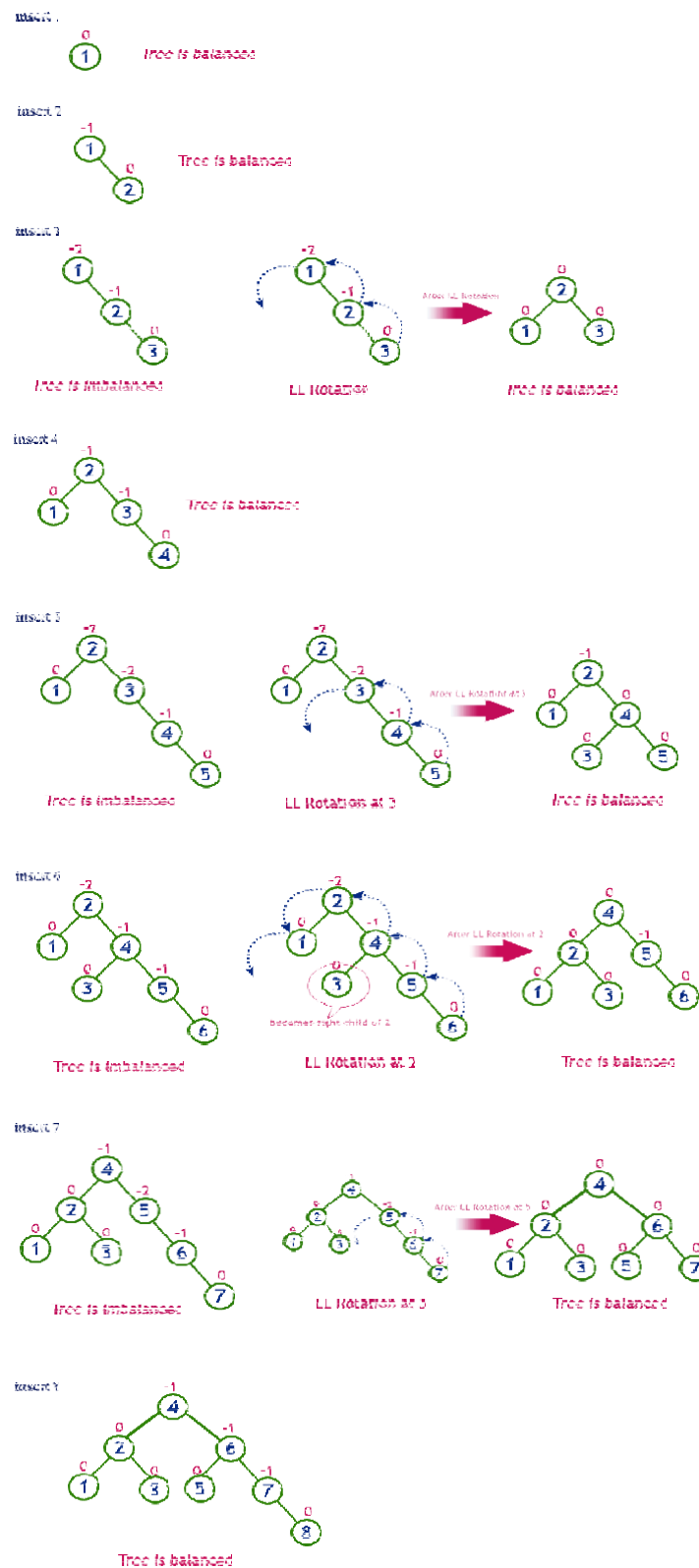
Tree is balanced

Deletion Operation in AVL Tree

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

B-tree

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.

It is also known as a height-balanced m-way tree.
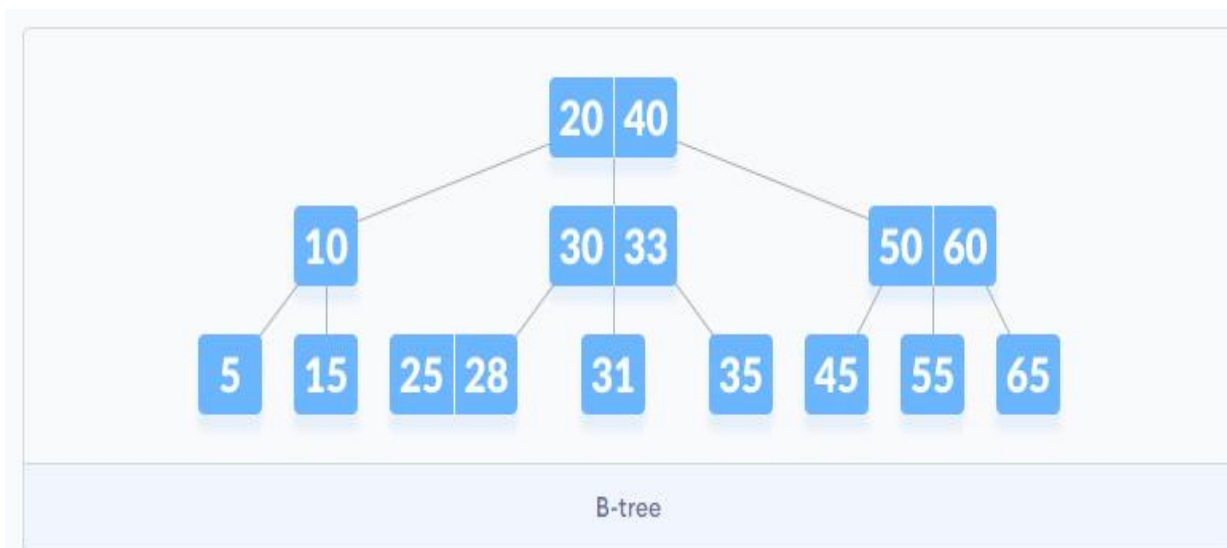
B-tree

Why do you need a B-tree data structure?

The need for B-tree arose with the rise in the need for lesser time in accessing physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk access.

Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large, and the access time increases.

However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

B-tree Properties

- For each node x, the keys are stored in increasing order.

- In each node, there is a boolean value x.leaf which is true if x is a leaf.

- If n is the order of the tree, each internal node can contain at most n - 1 keys along with a pointer to each child.

- Each node except root can have at most n children and at least n/2 children.

- All leaves have the same depth (i.e. height-h of the tree).

- The root has at least 2 children and contains a minimum of 1 key.

- If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $h \geq \log_t (n+1)/2$.

Operations on a B-tree

Searching an element in a B-tree

Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.

- Starting from the root node, compare k with the first key of the node.

- If k = the first key of the node, return the node and the index.

- If k.leaf = true, return NULL (i.e. not found).

- If k < the first key of the root node, search the left child of this key recursively.

- If there is more than one key in the current node and k > the first key, compare k with the next key in the node.

- If k < next key, search the left child of this key (ie. k lies in between the first and the second keys).

- Else, search the right child of the key.

- Repeat steps 1 to 4 until the leaf is reached.

Searching Example

Let us search key k = 17 in the tree below of degree 3.
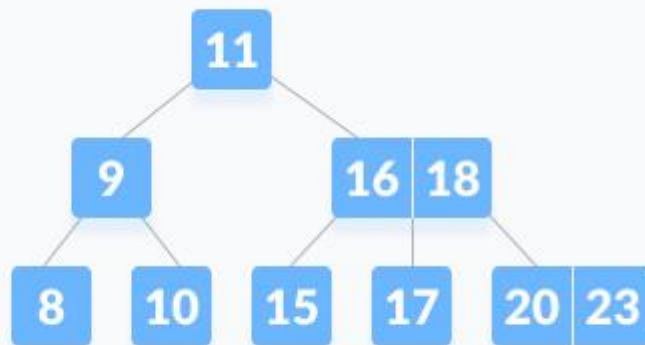
B-tree

k is not found in the root so, compare it with the root key.
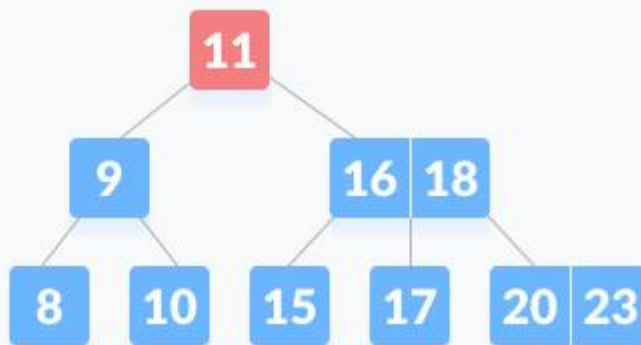
k is not found on the root node

Since k > 11, go to the right child of the root node
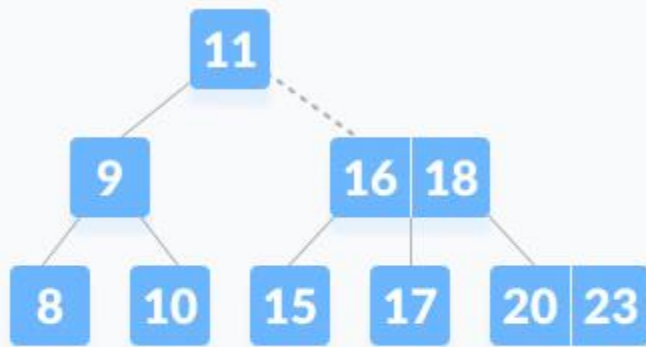
Go to the right subtree

Compare k with 16. Since k > 16, compare k with the next key 18.
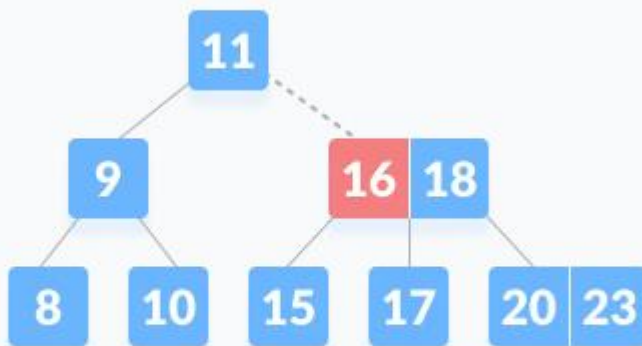
Compare with the keys from left to right

Since k < 18, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.
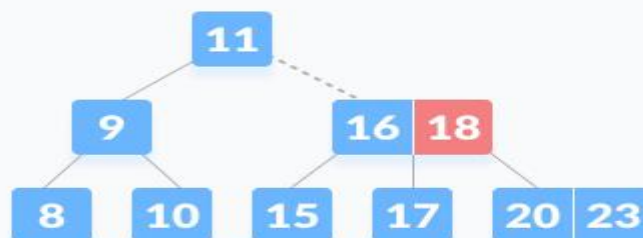
k lies in between 16 and 18

k is found.
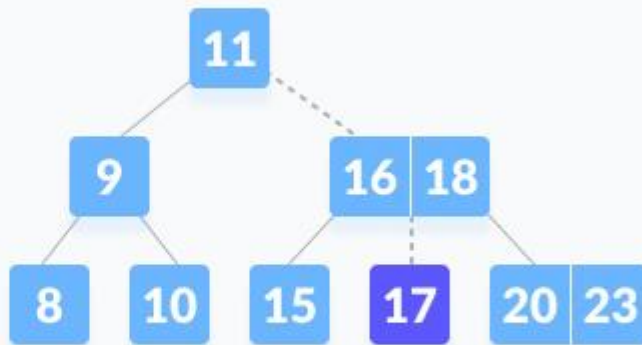
k is found

Algorithm for Searching an Element

```
BtreeSearch(x, k)
 i = 1
 while i ≤ n[x] and k ≥ keyi[x]            // n[x] means number of keys in x node
     do i = i + 1
 if i  n[x] and k = keyi[x]
     then return (x, i)
 if leaf [x]
     then return NIL
 else
     return BtreeSearch(ci[x], k)
```

B-tree operations code in Python

```python
# Searching a key on a B-tree in Python


# Create a node
class BTreeNode:
  def __init__(self, leaf=False):
    self.leaf = leaf
    self.keys = []
    self.child = []


# Tree
class BTree:
  def __init__(self, t):
```

```python
        self.root = BTreeNode(True)
        self.t = t

    # Insert node
    def insert(self, k):
        root = self.root
        if len(root.keys) == (2 * self.t) - 1:
            temp = BTreeNode()
            self.root = temp
            temp.child.insert(0, root)
            self.split_child(temp, 0)
            self.insert_non_full(temp, k)
        else:
            self.insert_non_full(root, k)

    # Insert nonfull
    def insert_non_full(self, x, k):
        i = len(x.keys) - 1
        if x.leaf:
            x.keys.append((None, None))
            while i >= 0 and k[0] < x.keys[i][0]:
                x.keys[i + 1] = x.keys[i]
                i -= 1
            x.keys[i + 1] = k
        else:
            while i >= 0 and k[0] < x.keys[i][0]:
                i -= 1
            i += 1
            if len(x.child[i].keys) == (2 * self.t) - 1:
                self.split_child(x, i)
                if k[0] > x.keys[i][0]:
                    i += 1
            self.insert_non_full(x.child[i], k)

    # Split the child
    def split_child(self, x, i):
        t = self.t
        y = x.child[i]
        z = BTreeNode(y.leaf)
        x.child.insert(i + 1, z)
        x.keys.insert(i, y.keys[t - 1])
        z.keys = y.keys[t: (2 * t) - 1]
        y.keys = y.keys[0: t - 1]
        if not y.leaf:
            z.child = y.child[t: 2 * t]
            y.child = y.child[0: t - 1]

# Print the tree
def print_tree(self, x, l=0):
    print("Level ", l, " ", len(x.keys), end=":")
    for i in x.keys:
        print(i, end=" ")
    print()
    l += 1
    if len(x.child) > 0:
```

```
        for i in x.child:
            self.print_tree(i, l)

  # Search key in the tree
  def search_key(self, k, x=None):
    if x is not None:
      i = 0
      while i < len(x.keys) and k > x.keys[i][0]:
        i += 1
      if i < len(x.keys) and k == x.keys[i][0]:
        return (x, i)
      elif x.leaf:
        return None
      else:
        return self.search_key(k, x.child[i])

    else:
      return self.search_key(k, self.root)


def main():
  B = BTree(3)

  for i in range(10):
    B.insert((i, 2 * i))

  B.print_tree(B.root)

  if B.search_key(8) is not None:
    print("\nFound")
  else:
    print("\nNot Found")


if __name__ == '__main__':
  main()
```

Searching Complexity on B Tree

Worst case Time complexity: $\Theta(\log n)$
Average case Time complexity: $\Theta(\log n)$
Best case Time complexity: $\Theta(\log n)$
Average case Space complexity: $\Theta(n)$
Worst case Space complexity: $\Theta(n)$

B Tree Applications
- databases and file systems
- to store blocks of data (secondary storage media)
- multilevel indexing

B+ Tree
A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.

An important concept to be understood before learning B+ tree is multilevel indexing. In multilevel indexing, the index of indices is created as in figure below. It makes accessing the data easier and faster.

Multilevel Indexing using B+ tree
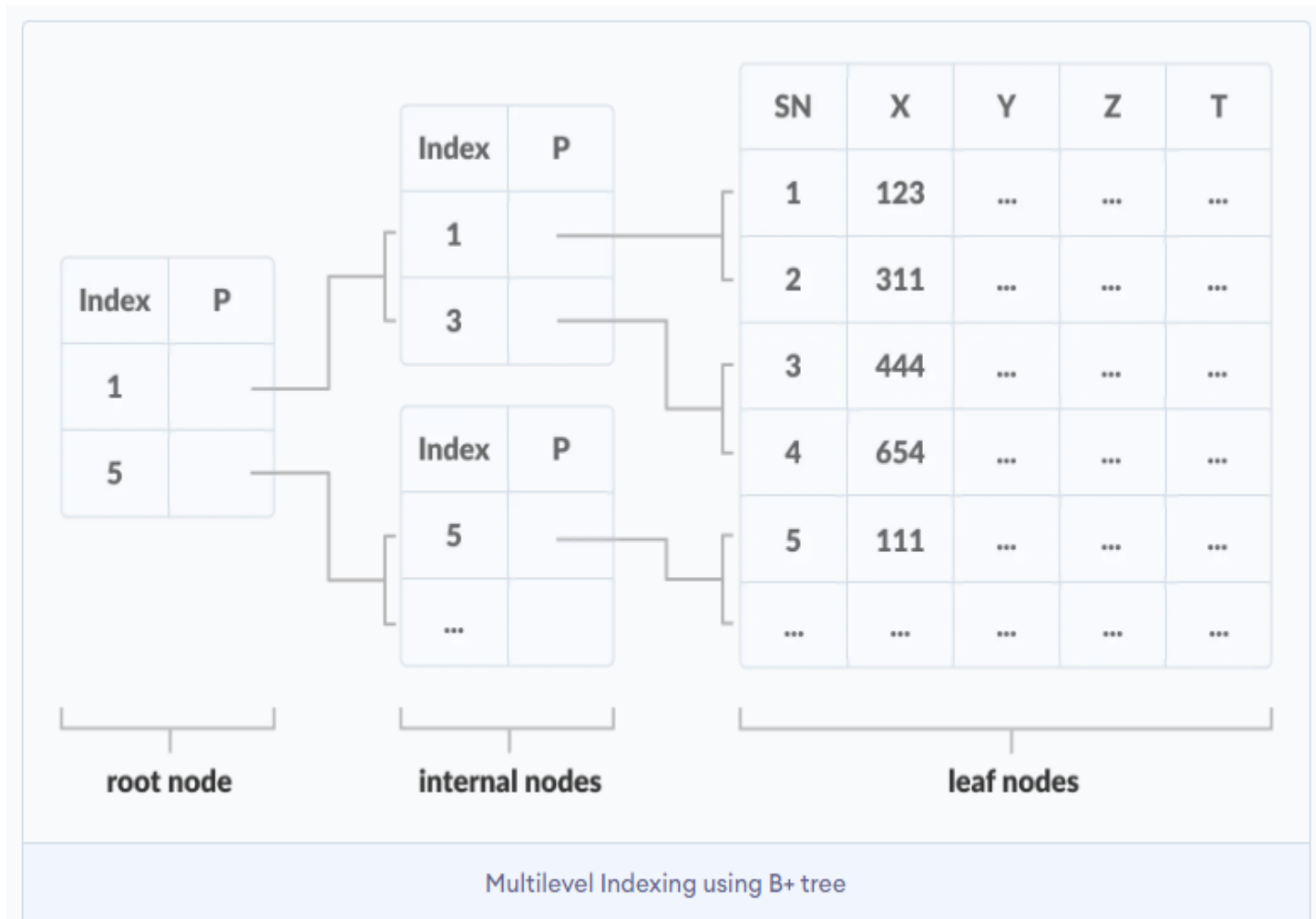
Properties of a B+ Tree

- All leaves are at the same level.

- The root has at least two children.

- Each node except root can have a maximum of m children and at least m/2 children.

- Each node can contain a maximum of m - 1 keys and a minimum of ⌈m/2⌉ - 1 keys.

Comparison between a B-tree and a B+ Tree
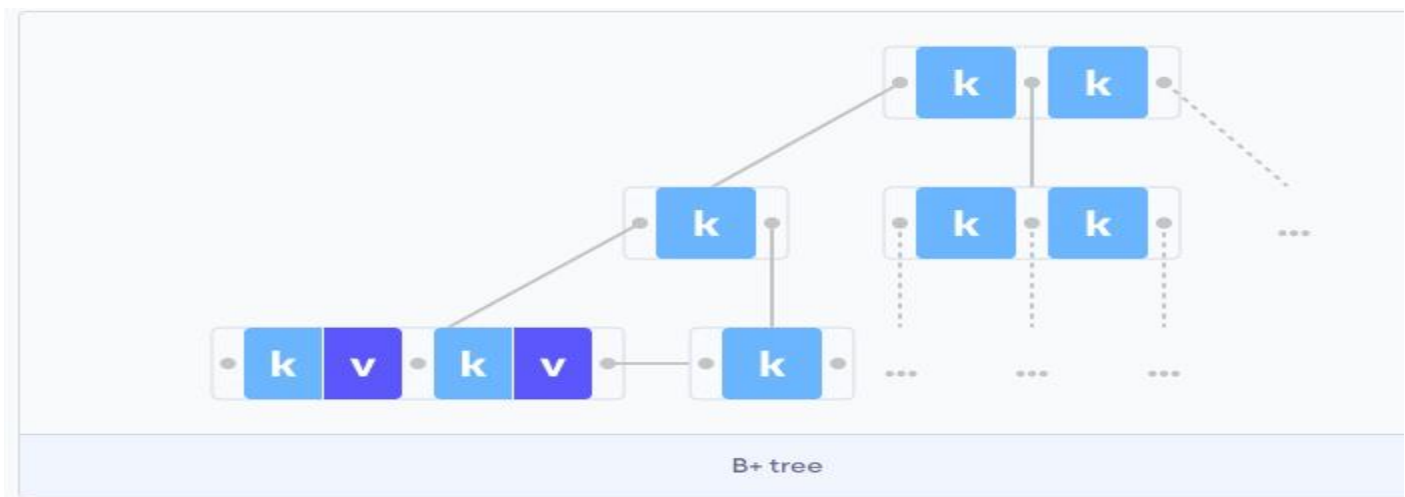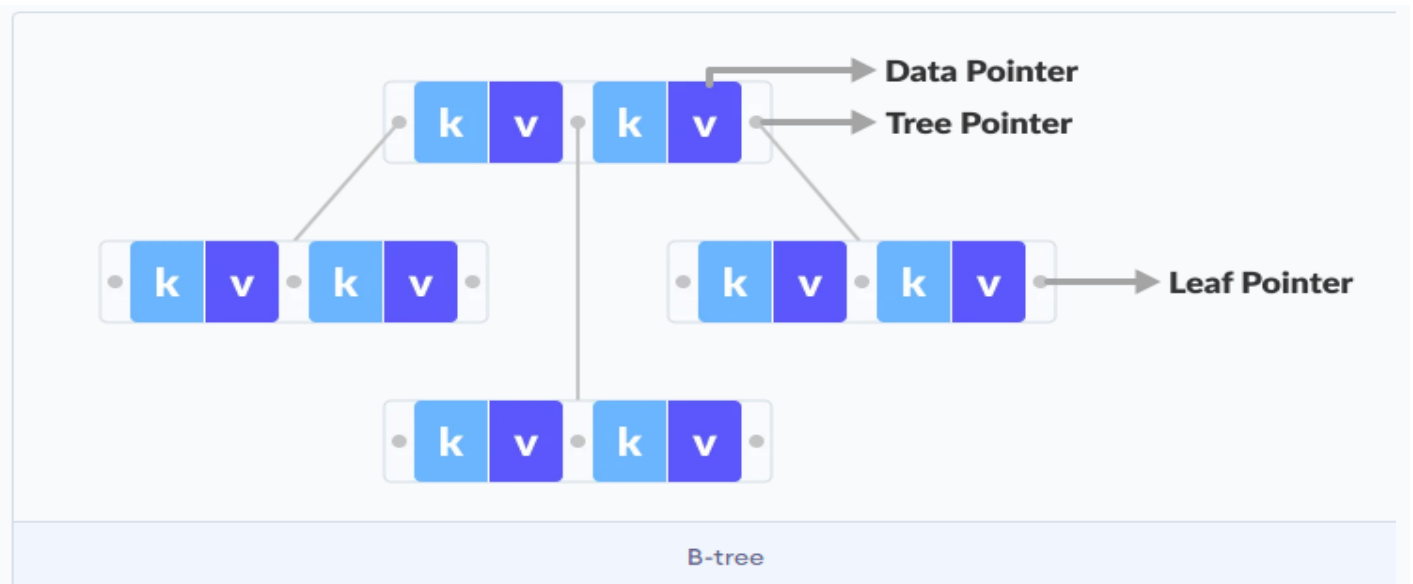
B-tree

B+ tree

The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.

The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.

Operations on a B+ tree are faster than on a B-tree.

Searching on a B+ Tree

The following steps are followed to search for data in a B+ Tree of order m. Let the data to be searched be k.

- Start from the root node. Compare k with the keys at the root node [k1, k2, k3,......km - 1].
- If k < k1, go to the left child of the root node.
- Else if k == k1, compare k2. If k < k2, k lies between k1 and k2. So, search in the left child of k2.
- If k > k2, go for k3, k4,...km-1 as in steps 2 and 3.
- Repeat the above steps until a leaf node is reached.
- If k exists in the leaf node, return true else return false.

Searching Example on a B+ Tree

Let us search k = 45 on the following B+ tree.

B+ tree

Compare k with the root node

k is not found at the root
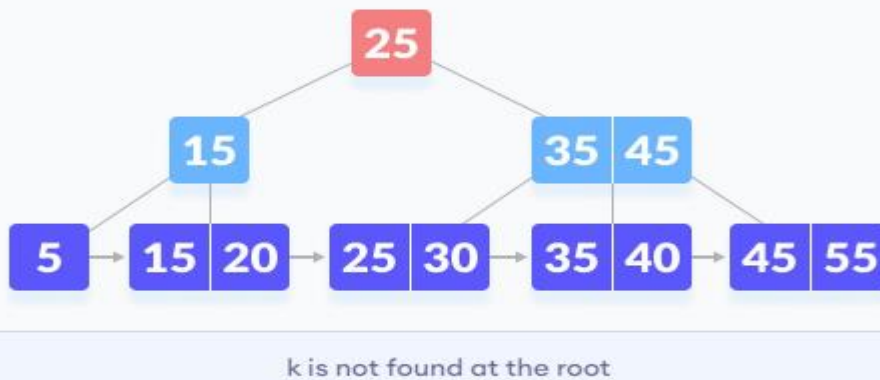
Since k > 25, go to the right child

Go to right of the root
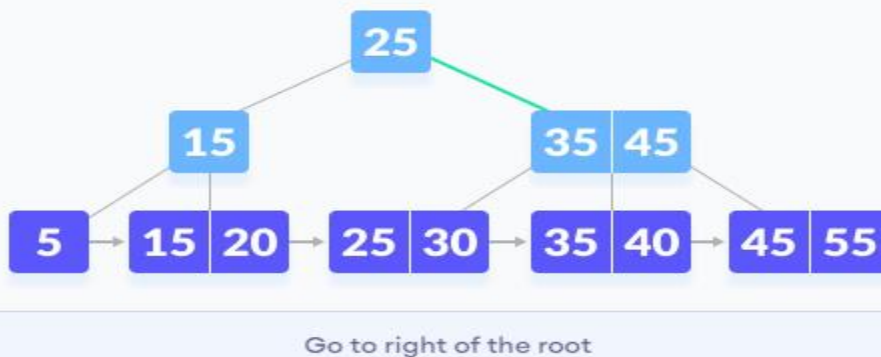
Compare k with 35. Since k > 30, compare k with 45.
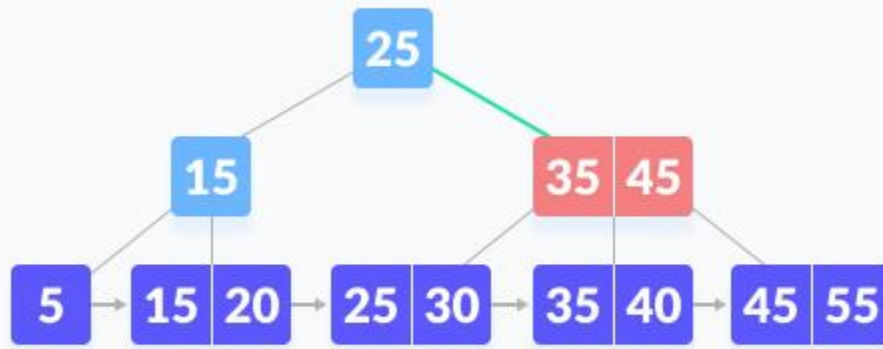
k not found

Since k ≥ 45, so go to the right child.
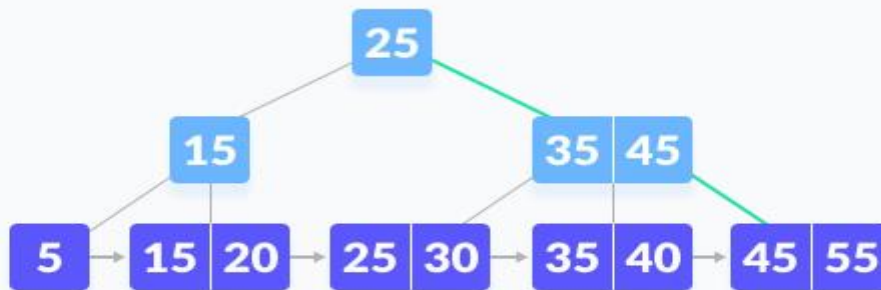
go to the right

k is found
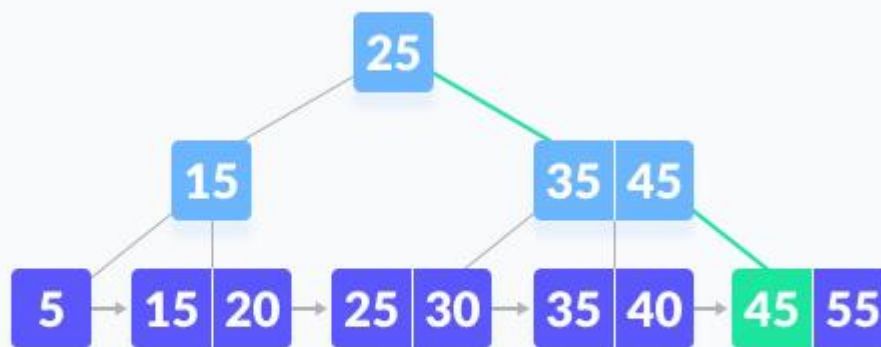
k is found

Python Example

```python
# B+ tee in python


import math

# Node creation
class Node:
    def __init__(self, order):
        self.order = order
        self.values = []
        self.keys = []
        self.nextKey = None
        self.parent = None
        self.check_leaf = False

    # Insert at the leaf
    def insert_at_leaf(self, leaf, value, key):
        if (self.values):
            temp1 = self.values
            for i in range(len(temp1)):
                if (value == temp1[i]):
                    self.keys[i].append(key)
                    break
                elif (value < temp1[i]):
                    self.values = self.values[:i] + [value] + self.values[i:]
                    self.keys = self.keys[:i] + [[key]] + self.keys[i:]
                    break
                elif (i + 1 == len(temp1)):
                    self.values.append(value)
                    self.keys.append([key])
                    break
        else:
            self.values = [value]
            self.keys = [[key]]


# B plus tree
class BplusTree:
    def __init__(self, order):
        self.root = Node(order)
        self.root.check_leaf = True

    # Insert operation
    def insert(self, value, key):
        value = str(value)
        old_node = self.search(value)
        old_node.insert_at_leaf(old_node, value, key)

        if (len(old_node.values) == old_node.order):
            node1 = Node(old_node.order)
            node1.check_leaf = True
            node1.parent = old_node.parent
            mid = int(math.ceil(old_node.order / 2)) - 1
            node1.values = old_node.values[mid + 1:]
```

```python
            node1.keys = old_node.keys[mid + 1:]
            node1.nextKey = old_node.nextKey
            old_node.values = old_node.values[:mid + 1]
            old_node.keys = old_node.keys[:mid + 1]
            old_node.nextKey = node1
            self.insert_in_parent(old_node, node1.values[0], node1)

    # Search operation for different operations
    def search(self, value):
        current_node = self.root
        while(current_node.check_leaf == False):
            temp2 = current_node.values
            for i in range(len(temp2)):
                if (value == temp2[i]):
                    current_node = current_node.keys[i + 1]
                    break
                elif (value < temp2[i]):
                    current_node = current_node.keys[i]
                    break
                elif (i + 1 == len(current_node.values)):
                    current_node = current_node.keys[i + 1]
                    break
        return current_node

    # Find the node
    def find(self, value, key):
        l = self.search(value)
        for i, item in enumerate(l.values):
            if item == value:
                if key in l.keys[i]:
                    return True
                else:
                    return False
        return False

    # Inserting at the parent
    def insert_in_parent(self, n, value, ndash):
        if (self.root == n):
            rootNode = Node(n.order)
            rootNode.values = [value]
            rootNode.keys = [n, ndash]
            self.root = rootNode
            n.parent = rootNode
            ndash.parent = rootNode
            return

        parentNode = n.parent
        temp3 = parentNode.keys
        for i in range(len(temp3)):
            if (temp3[i] == n):
                parentNode.values = parentNode.values[:i] + \
                    [value] + parentNode.values[i:]
                parentNode.keys = parentNode.keys[:i +
                                                  1] + [ndash] + parentNode.keys[i + 1:]
                if (len(parentNode.keys) > parentNode.order):
```

```python
                parentdash = Node(parentNode.order)
                parentdash.parent = parentNode.parent
                mid = int(math.ceil(parentNode.order / 2)) - 1
                parentdash.values = parentNode.values[mid + 1:]
                parentdash.keys = parentNode.keys[mid + 1:]
                value_ = parentNode.values[mid]
                if (mid == 0):
                    parentNode.values = parentNode.values[:mid + 1]
                else:
                    parentNode.values = parentNode.values[:mid]
                parentNode.keys = parentNode.keys[:mid + 1]
                for j in parentNode.keys:
                    j.parent = parentNode
                for j in parentdash.keys:
                    j.parent = parentdash
                self.insert_in_parent(parentNode, value_, parentdash)

    # Delete a node
    def delete(self, value, key):
        node_ = self.search(value)

        temp = 0
        for i, item in enumerate(node_.values):
            if item == value:
                temp = 1

                if key in node_.keys[i]:
                    if len(node_.keys[i]) > 1:
                        node_.keys[i].pop(node_.keys[i].index(key))
                    elif node_ == self.root:
                        node_.values.pop(i)
                        node_.keys.pop(i)
                    else:
                        node_.keys[i].pop(node_.keys[i].index(key))
                        del node_.keys[i]
                        node_.values.pop(node_.values.index(value))
                        self.deleteEntry(node_, value, key)
                else:
                    print("Value not in Key")
                    return
        if temp == 0:
            print("Value not in Tree")
            return

    # Delete an entry
    def deleteEntry(self, node_, value, key):

        if not node_.check_leaf:
            for i, item in enumerate(node_.keys):
                if item == key:
                    node_.keys.pop(i)
                    break
            for i, item in enumerate(node_.values):
                if item == value:
                    node_.values.pop(i)
```

```
                break

        if self.root == node_ and len(node_.keys) == 1:
            self.root = node_.keys[0]
            node_.keys[0].parent = None
            del node_
            return
        elif (len(node_.keys) < int(math.ceil(node_.order / 2)) and node_.check_leaf == False) or
(len(node_.values) < int(math.ceil((node_.order - 1) / 2)) and node_.check_leaf == True):

            is_predecessor = 0
            parentNode = node_.parent
            PrevNode = -1
            NextNode = -1
            PrevK = -1
            PostK = -1
            for i, item in enumerate(parentNode.keys):

                if item == node_:
                    if i > 0:
                        PrevNode = parentNode.keys[i - 1]
                        PrevK = parentNode.values[i - 1]

                    if i < len(parentNode.keys) - 1:
                        NextNode = parentNode.keys[i + 1]
                        PostK = parentNode.values[i]

            if PrevNode == -1:
                ndash = NextNode
                value_ = PostK
            elif NextNode == -1:
                is_predecessor = 1
                ndash = PrevNode
                value_ = PrevK
            else:
                if len(node_.values) + len(NextNode.values) < node_.order:
                    ndash = NextNode
                    value_ = PostK
                else:
                    is_predecessor = 1
                    ndash = PrevNode
                    value_ = PrevK

            if len(node_.values) + len(ndash.values) < node_.order:
                if is_predecessor == 0:
                    node_, ndash = ndash, node_
                ndash.keys += node_.keys
                if not node_.check_leaf:
                    ndash.values.append(value_)
                else:
                    ndash.nextKey = node_.nextKey
                ndash.values += node_.values

                if not ndash.check_leaf:
                    for j in ndash.keys:
```

```
                                j.parent = ndash

                    self.deleteEntry(node_.parent, value_, node_)
                    del node_
                else:
                    if is_predecessor == 1:
                        if not node_.check_leaf:
                            ndashpm = ndash.keys.pop(-1)
                            ndashkm_1 = ndash.values.pop(-1)
                            node_.keys = [ndashpm] + node_.keys
                            node_.values = [value_] + node_.values
                            parentNode = node_.parent
                            for i, item in enumerate(parentNode.values):
                                if item == value_:
                                    p.values[i] = ndashkm_1
                                    break
                        else:
                            ndashpm = ndash.keys.pop(-1)
                            ndashkm = ndash.values.pop(-1)
                            node_.keys = [ndashpm] + node_.keys
                            node_.values = [ndashkm] + node_.values
                            parentNode = node_.parent
                            for i, item in enumerate(p.values):
                                if item == value_:
                                    parentNode.values[i] = ndashkm
                                    break
                    else:
                        if not node_.check_leaf:
                            ndashp0 = ndash.keys.pop(0)
                            ndashk0 = ndash.values.pop(0)
                            node_.keys = node_.keys + [ndashp0]
                            node_.values = node_.values + [value_]
                            parentNode = node_.parent
                            for i, item in enumerate(parentNode.values):
                                if item == value_:
                                    parentNode.values[i] = ndashk0
                                    break
                        else:
                            ndashp0 = ndash.keys.pop(0)
                            ndashk0 = ndash.values.pop(0)
                            node_.keys = node_.keys + [ndashp0]
                            node_.values = node_.values + [ndashk0]
                            parentNode = node_.parent
                            for i, item in enumerate(parentNode.values):
                                if item == value_:
                                    parentNode.values[i] = ndash.values[0]
                                    break

                    if not ndash.check_leaf:
                        for j in ndash.keys:
                            j.parent = ndash
                    if not node_.check_leaf:
                        for j in node_.keys:
                            j.parent = node_
                    if not parentNode.check_leaf:
```

```python
                for j in parentNode.keys:
                    j.parent = parentNode


# Print the tree
def printTree(tree):
    lst = [tree.root]
    level = [0]
    leaf = None
    flag = 0
    lev_leaf = 0

    node1 = Node(str(level[0]) + str(tree.root.values))

    while (len(lst) != 0):
        x = lst.pop(0)
        lev = level.pop(0)
        if (x.check_leaf == False):
            for i, item in enumerate(x.keys):
                print(item.values)
        else:
            for i, item in enumerate(x.keys):
                print(item.values)
            if (flag == 0):
                lev_leaf = lev
                leaf = x
                flag = 1


record_len = 3
bplustree = BplusTree(record_len)
bplustree.insert('5', '33')
bplustree.insert('15', '21')
bplustree.insert('25', '31')
bplustree.insert('35', '41')
bplustree.insert('45', '10')

printTree(bplustree)

if(bplustree.find('5', '34')):
    print("Found")
else:
    print("Not found")
```

Search Complexity
Time Complexity
If linear search is implemented inside a node, then total complexity is Θ(logt n).
If binary search is used, then total complexity is Θ(log2t.logt n).

B+ Tree Applications
- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)
- Database indexing