COURSE MATERIAL

II Year B. Tech II- Semester

MECHANICAL ENGINEERING

DATA STRUCTURES USING PYTHON

R20A0311

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF MECHANICAL ENGINEERING

(Autonomous Institution-UGC, Govt. of India)
Secunderabad-500100, Telangana State, India.
www.mrcet.ac.in

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

DEPARTMENT OF MECHANICAL ENGINEERING

CONTENTS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

VISION

- To establish a pedestal for the integral innovation, team spirit, originality and competence in the students, expose them to face the global challenges and become technology leaders of Indian vision of modern society.

MISSION

- To become a model institution in the fields of Engineering, Technology and Management.
- To impart holistic education to the students to render them as industry ready engineers.
- To ensure synchronization of MRCET ideologies with challenging demands of International Pioneering Organizations.

QUALITY POLICY

- To implement best practices in Teaching and Learning process for both UG and PG courses meticulously.
- To provide state of art infrastructure and expertise to impart quality education.
- To groom the students to become intellectually creative and professionally competitive.
- To channelize the activities and tune them in heights of commitment and sincerity, the requisites to claim the never - ending ladder of **SUCCESS** year after year.

For more information: www.mrcet.ac.in

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

www.mrcet.ac.in

Department of Mechanical Engineering

VISION

To become an innovative knowledge center in mechanical engineering through state-of-the-art teaching-learning and research practices, promoting creative thinking professionals.

MISSION

The Department of Mechanical Engineering is dedicated for transforming the students into highly competent Mechanical engineers to meet the needs of the industry, in a changing and challenging technical environment, by strongly focusing in the fundamentals of engineering sciences for achieving excellent results in their professional pursuits.

Quality Policy

- ✓ To pursuit global Standards of excellence in all our endeavors namely teaching, research and continuing education and to remain accountable in our core and support functions, through processes of self-evaluation and continuous improvement.
- ✓ To create a midst of excellence for imparting state of art education, industry-oriented training research in the field of technical education.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

www.mrcet.ac.in

Department of Mechanical Engineering

PROGRAM OUTCOMES

Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and teamwork**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1 Ability to analyze, design and develop Mechanical systems to solve the Engineering problems by integrating thermal, design and manufacturing Domains.

PSO2 Ability to succeed in competitive examinations or to pursue higher studies or research.

PSO3 Ability to apply the learned Mechanical Engineering knowledge for the Development of society and self.

Program Educational Objectives (PEOs)

The Program Educational Objectives of the program offered by the department are broadly listed below:

PEO1: PREPARATION

To provide sound foundation in mathematical, scientific and engineering fundamentals necessary to analyze, formulate and solve engineering problems.

PEO2: CORE COMPETANCE

To provide thorough knowledge in Mechanical Engineering subjects including theoretical knowledge and practical training for preparing physical models pertaining to Thermodynamics, Hydraulics, Heat and Mass Transfer, Dynamics of Machinery, Jet Propulsion, Automobile Engineering, Element Analysis, Production Technology, Mechatronics etc.

PEO3: INVENTION, INNOVATION AND CREATIVITY

To make the students to design, experiment, analyze, interpret in the core field with the help of other inter disciplinary concepts wherever applicable.

PEO4: CAREER DEVELOPMENT

To inculcate the habit of lifelong learning for career development through successful completion of advanced degrees, professional development courses, industrial training etc.

PEO5: PROFESSIONALISM

To impart technical knowledge, ethical values for professional development of the student to solve complex problems and to work in multi-disciplinary ambience, whose solutions lead to significant societal benefits.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

www.mrcet.ac.in

Department of Mechanical Engineering

Blooms Taxonomy

Bloom's Taxonomy is a classification of the different objectives and skills that educators set for their students (learning objectives). The terminology has been updated to include the following six levels of learning. These 6 levels can be used to structure the learning objectives, lessons, and assessments of a course.

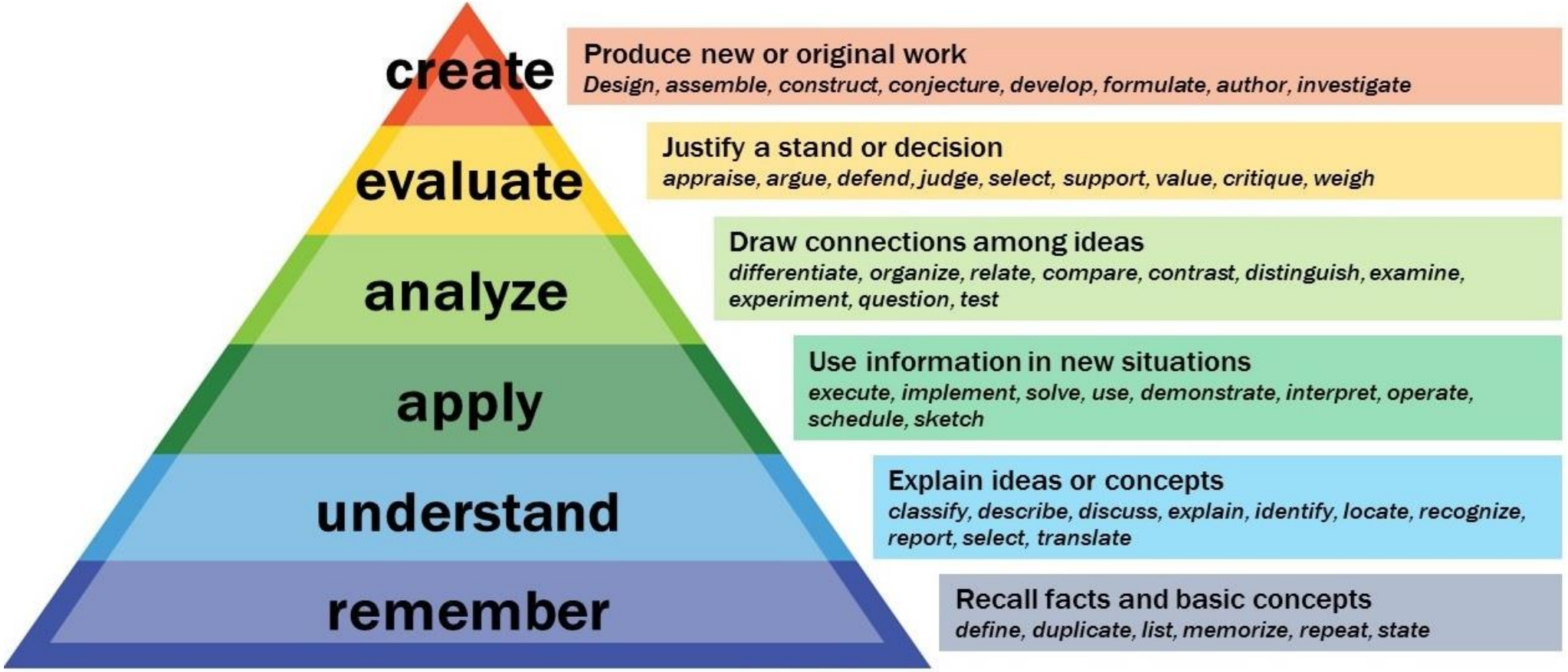
1. **Remembering**: Retrieving, recognizing, and recalling relevant knowledge from long- term memory.
2. **Understanding**: Constructing meaning from oral, written, and graphic messages through interpreting, exemplifying, classifying, summarizing, inferring, comparing, and explaining.
3. **Applying**: Carrying out or using a procedure for executing or implementing.
4. **Analyzing**: Breaking material into constituent parts, determining how the parts relate to one another and to an overall structure or purpose through differentiating, organizing, and attributing.
5. **Evaluating**: Making judgments based on criteria and standard through checking and critiquing.
6. **Creating**: Putting elements together to form a coherent or functional whole; reorganizing elements into a new pattern or structure through generating, planning, or producing.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

www.mrcet.ac.in

Department of Mechanical Engineering



MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

II Year B.Tech. ME- II Sem **L/T/P/C**
2/1/-/3

(R20A0311) DATA STRUCTURES USING PYTHON

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

Data Structures – Definition,Linear Data Structures,Non-Linear Data Structures

Python Specific Data Structures: List,Tuples, Set, Dictionaries, Comprehensions and its Types,Strings,slicing.

UNIT III

Arrays - Overview, Types of Arrays, Operations on Arrays, Arrays vs List.

Searching -Linear Search and Binary Search.

Sorting - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort.

UNIT IV

Linked Lists – Implementation of Singly Linked Lists, Doubly Linked Lists.

Stacks - Overview of Stack, Implementation of Stack (List & Linked list).

Queues:Overview of Queue, Implementation of Queue(List & Linked list).

UNIT V

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

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

TEXT BOOKS

1. Data structures and algorithms in python by Michael T. Goodrich
2. Data Structures and Algorithmic Thinking with Python by NarasimhaKarumanchi

REFERENCE BOOKS:

1. Hands-On Data Structures and Algorithms with Python: Write complex and powerful code using the latest features 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

UNIT 1

Oops Concepts

UNIT-I

Syllabus:

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.

Data Structure

Introduction

- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.
- Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.
- It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arrise the following problems:

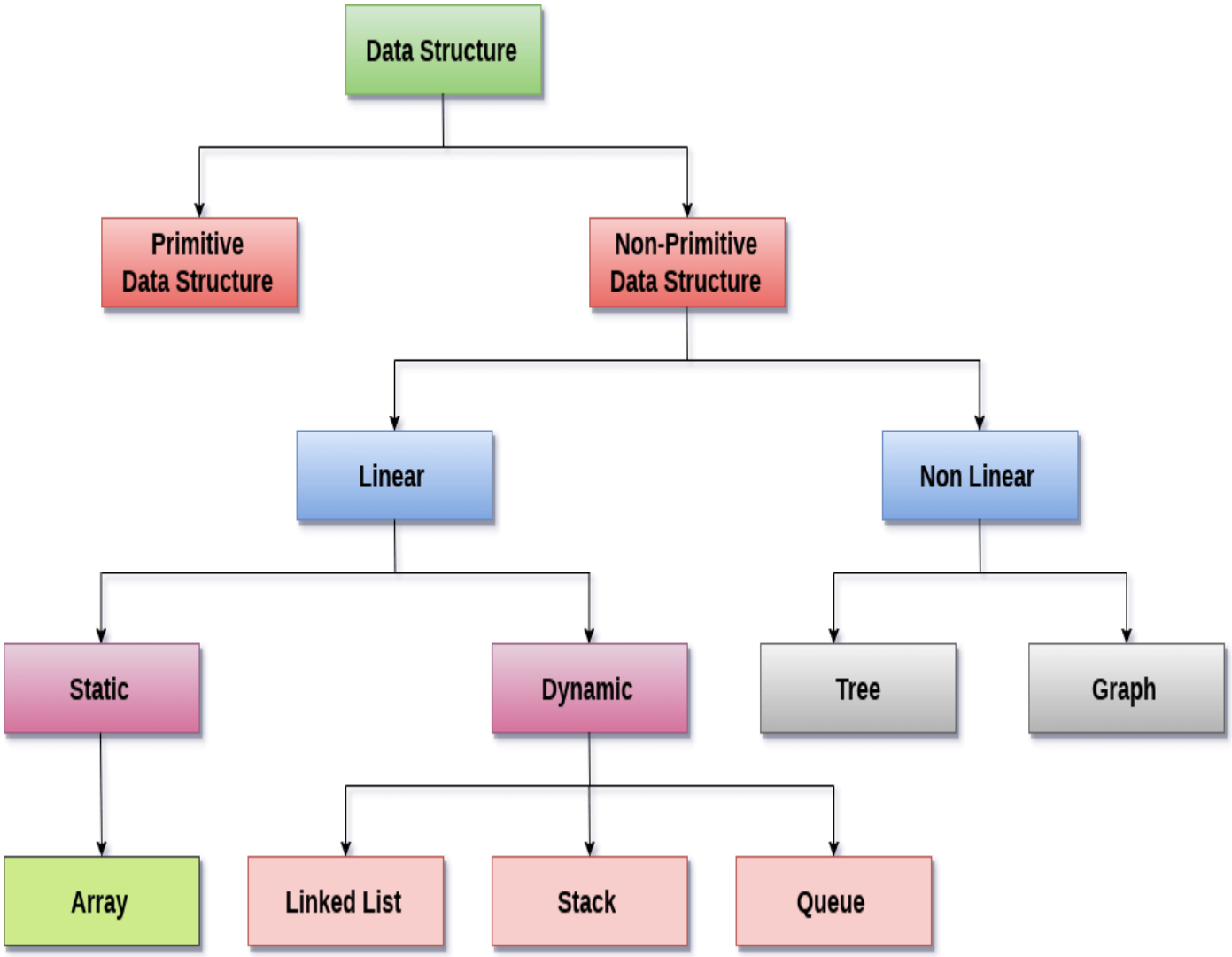
Processor speed: To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process.

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Data Structure Classification



Introduction

The term "Object-Oriented Programming" (OOP) was coined by Alan Kay around 1966 while he was at grad school. The language called **Simula** was the first programming language with the features of Object-oriented programming. It was developed in 1967 for making simulation programs, in which the most important information was called objects.

Though OOPs were in the market since the early 1960s it was in the 1990s that OOPs began to grow because of C++. After that, this technique of programming has been adapted by various programming languages including Python. Today its application is in almost every field such as Real-time systems, Artificial intelligence, and expert systems, Client-server systems, Object-oriented databases, and many more.

What Is Object-Oriented Programming

- Object-Oriented Programming(OOP), is all about creating "objects".
- An object is a group of interrelated variables(properties) and functions.
- These variables are often referred to as properties of the object and functions are referred to as the behavior of the objects.
- These objects provide a better and clear structure for the program.
- **For example**: A car can be an object. If we consider the car as an object then its properties would be – its color, its model, its price, its brand, etc. And its behavior/function would be acceleration, slowing down, gear change.
- Object-Oriented programming is famous because it implements the real-world entities like objects, hiding, inheritance, etc in programming.
- It makes visualization easier because it is close to real-world scenarios.

Difference Between OOP and POP

Procedure-oriented	Object-oriented
It is often known as POP (procedure-oriented programming).	It is often known as OOP (object-oriented programming).
It follows the top-bottom flow of execution.	It follows the bottom-top flow of execution.
Larger programs have divided into smaller modules called as functions.	The larger program has divided into objects.
The main focus is on solving the problem.	The main focus is on data security.
It doesn't support data abstraction.	It supports data abstraction using access specifiers that are public, protected, and private.
It doesn't support inheritance.	It supports the inheritance of four

Procedure-oriented	Object-oriented
	types.
Overloading is not supported.	It supports the overloading of function and also the operator.
There is no concept of friend function and virtual functions.	It has the concept of friend function and virtual functions.
Examples - C, FORTRAN	Examples - C++ , Java , VB.net, C#.net, Python, R Programming, etc.

OOP Concepts:

1.Class

2.Object

3.Encapsulation

4.Data Abstraction

5.Inheritance

6.Polymorthism

7.Data Binding

8.Message Passing

Class

- A class is a collection of objects.
  (or)
- Class is a blue print from which specific objects are created.
- Unlike the primitive data structures, classes are data structures that the user defines. They make the code more manageable.

Declaration of Class:

```
class class_name:
    Statement-1
    Statement-2
    ----------
    ----------
    Statement-n
```

Example:

```
class Car:
 pass
```

Note:

- We define a class with a keyword “class” following the class_name and colon.
- And we consider everything you write under this after using indentation as its body.

Objects and object instantiation

When we define a class only the description or a blueprint of the object is created. There is no memory allocation until we create its **object**. The **objector instance** contains real data or information.

Instantiation is nothing but creating a new object/instance of a class. Let’s create the object of the above class we defined-

```
obj1 = Car()
```

And it’s done! **Note** that you can change the object name according to your choice.

Try printing this object-

```
print(obj1)
```

```
<__main__.car object at 0x7fc5e677b6d8>
```

Since our class was empty, it returns the address where the object is stored i.e 0x7fc5e677b6d8

You also need to understand the class conductor before moving forward.

Constructors

Def:A constructor is a special type of method (function) which is used to initialize the instance members of the class.

- In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python.
- It is used to create an object.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Creating the Constructor in Python

- In Python, the method the **__init__()** simulates the constructor of the class.
- This method is called when the class is instantiated.
- It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.
- We can pass any number of arguments at the time of creating the class object, depending upon the **__init__()** definition.
- It is mostly used to initialize the class attributes.
- Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the **Employee** class attributes.

Example

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))


emp1 = Employee("John", 101)
emp2 = Employee("David", 102)

# accessing display() method to print employee 1 information
```

```
emp1.display()

# accessing display() method to print employee 2 information
emp2.display()
```

Output:

```
ID: 101
Name: John
ID: 102
Name: David
```

Python Non-Parameterized Constructor

- The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.

  Consider the following example.

Example

```
class Student:
   # Constructor - non parameterized
   def __init__(self):
      print("This is non parametrized constructor")
   def show(self,name):
      print("Hello",name)
student = Student()
student.show("John")
```

Python Parameterized Constructor

- The parameterized constructor has multiple parameters along with the **self**.

Consider the following example.

Example

```
class Student:
   # Constructor - parameterized
   def __init__(self, name):
      print("This is parametrized constructor")
      self.name = name
   def show(self):
      print("Hello",self.name)
student = Student("John")
student.show()
```

Output:

```
This is parametrized constructor
Hello John
```

Python Default Constructor

- When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor.
- It does not perform any task but initializes the objects.

Consider the following example.

Example

```
class Student:
    roll_num = 101
    name = "Joseph"

    def display(self):
        print(self.roll_num,self.name)

st = Student()
st.display()
```

Output:

```
101 Joseph
```

3. Class methods

- Methods are the functions that we use to describe the behavior of the objects.
- They are also defined inside a class.

Look at the following code-

```
class Car:
    car_type = "Sedan"

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage
```

```
    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"


    def max_speed(self, speed):
        return f"The {self.name} runs at the maximum speed of {speed}km/hr"
```

The methods defined inside a class other than the constructor method are known as the **instance** methods. Furthermore, we have two instance methods here- **description**() and **max_speed**(). Let's talk about them individually-

- **description**()- This method is returning a string with the description of the car such as the name and its mileage. This method has no additional parameter. This method is using the instance attributes.
- **max_speed**()- This method has one additional parameter and returning a string displaying the car name and its speed.

Notice that the additional parameter speed is not using the "self" keyword. Since speed is not an instance variable, we don't use the self keyword as its prefix. Let's create an object for the class described above.

```
obj2 = Car("Honda City",24.1)
print(obj2.description())
print(obj2.max_speed(150))
```

```
The Honda City car gives the mileage of 24.1km/l
The Honda City runs at the maximum speed of 150km/hr
```

- What we did is we created an object of class car and passed the required arguments. In order to access the instance methods we use object_name.method_name().
- The method description() didn't have any additional parameter so we did not pass any argument while calling it.
- The method max_speed() has one additional parameter so we passed one argument while calling it.

Note: Three important things to remember are-

1. We can create any number of objects of a class.
2. If the method requires n parameters and we do not pass the same number of arguments then an error will occur.
3. Order of the arguments matters.

Let’s look at this one by one-

Creating more than one object of a class

```
class Car:
    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

    def max_speed(self, speed):
            return f"The {self.name} runs at the maximum speed of {speed}km/hr"
    Honda = Car("Honda City",21.4)
    print(Honda.max_speed(150))

    Skoda = Car("Skoda Octavia",13)
    print(Skoda.max_speed(210))
```

```
The Honda City runs at the maximum speed of 150km/hr
The Skoda Octavia runs at the maximum speed of 210km/hr
```

1.Passing the wrong number of arguments.

```
class Car:

def __init__(self, name, mileage):
        self.name = name
          self.mileage = mileage
    Honda = Car("Honda City")
    print(Honda)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-19b2d6fccf19> in <module>()
----> 1 Honda = car("Honda City")
      2 print(Honda)

TypeError: __init__() missing 1 required positional argument: 'mileage'
```

Since we did not provide the second argument, we got this error.

Order of the arguments

```
class Car:

    def __init__(self, name, mileage):
        self.name = name
        self.mileage = mileage

    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"
Honda = Car(24.1,"Honda City")
print(Honda.description())
```

```
The 24.1 car gives the mileage of Honda Citykm/l
```

Different types of methods that can be defined in a Python class

In a Python class, we can define three types of methods:

1. Instance methods
2. Class methods
3. Static methods

1.Instance methods

- **Instance methods** are the most used methods in a Python class.
- These methods are only accessible through class objects.
- If we want to modify any class variable, this should be done inside an instance method.
- The first parameter in these methods is self. **self** is used to refer to the current class object's properties and attributes.
- **Example:**

```
class Cricket:
    teamName = None

    def setTeamName(self, name):
        self.teamName = name

    def getTeamName(self):
        return self.teamName

c = Cricket()
c.setTeamName('India')

print(c.getTeamName())
```

Explanation

- In **line 1**, we define our class.
- In **line 2**, we define a class variable and set it to None.
- In **lines 4 and 7**, we create two instance methods: setTeamName() and getTeamName().
- In **lines 11 and 12**, we use the class object to access the instance methods.

2.Class methods

- **Class methods** are usually used to access class variables.
- We can call these methods directly using the class name instead of creating an object of that class.
- To declare a class method, we need to use the @classmethod decorator.
- Also, as in the case of instance methods, self is the keyword used to access the class variables. In class methods, we use use the cls variable to refer to the class.

```
class Cricket:
  teamName = 'India'

  @classmethod
  def getTeamName(cls):
    return cls.teamName


print(Cricket.getTeamName())
```

Explanation

- In **line 1**, we create our class Cricket.
- In **line 2**, we define a class variable.
- In **line 4**, we use the decorator @classmethod to specify the below method as a class method.
- In **line 5**, we define our class method. (Note that we have used cls to access the class variable. You can give any name for this parameter, but as per the convention, the name of this parameter should be cls.
- In **line 8**, we call our class method by using the class name instead of creating an object of the class.

3.Static methods

- **Static methods** are usually used as a utility function or when we do not want an inherited class to modify a function definition.
- These methods do not have any relation to the class variables and instance variables; so, are not allowed to modify the class attributes inside a static method.
- To declare a static method, we need to use the `@staticmethod`. Again, we will be using the `cls` variable to refer to the class. These methods can be accessed using the class name as well as class objects.

Example:

```
class Cricket:
    teamName = 'India'

    @staticmethod
    def utility():
        print("This is a static method.")

c1 = Cricket()
c1.utility()

Cricket.utility()
```

Explanation

- The code is almost the same, but with a difference in **line 4**, where we used the decorator `@staticmethod` to specify the below method as a static method.
- Then, in **line 9**, we call our static method by using the class object and, in **line 11**, we call the static method using the class name.

Variables and Types

- Variables are nothing but reserved memory locations to store values.
- This means that when you create a variable you reserve some space in memory.

- Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

- Python variables do not need explicit declaration to reserve memory space.
- The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.
- The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example:

```
#!/usr/bin/python

counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result −

```
100
1000.0
John
```

Multiple Assignment

- Python allows you to assign a single value to several variables simultaneously. For example :a = b = c = 1
- Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

  For example : a,b,c = 1,2,"john"
- Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

- The data stored in memory can be of many types.

- For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

- Number data types store numeric values. Number objects are created when you assign a value to them.

  For example −

  ```
  var1 = 1
  var2 = 10
  ```

- You can also delete the reference to a number object by using the del statement. The syntax of the del statement is −

  ```
  el var1[,var2[,var3[....,varN]]]]
  ```

- You can delete a single object or multiple objects by using the del statement. For example : del var

  ```
  del var_a, var_b
  ```

Python supports four different numerical types −

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Examples

Here are some examples of numbers −

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j

-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

Python Strings

- Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.
- Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example −

```
#!/usr/bin/python

str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

This will produce the following result −

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

- Lists are the most versatile of Python's compound data types.
- A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C.
- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.
- The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

Example −

```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

This produce the following result −

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuples

- A tuple is another sequence data type that is similar to the list.
- A tuple consists of a number of values separated by commas.
- Unlike lists, tuples are enclosed within parentheses.

The main differences between lists and tuples are:

- Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated.
- Tuples can be thought of as **read-only** lists.

For example −

```
#!/usr/bin/python
```

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple             # Prints the complete tuple
print tuple[0]          # Prints first element of the tuple
print tuple[1:3]        # Prints elements of the tuple starting from 2nd till 3rd
print tuple[2:]         # Prints elements of the tuple starting from 3rd element
print tinytuple * 2     # Prints the contents of the tuple twice
print tuple + tinytuple # Prints concatenated tuples
```

This produce the following result −

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Dictionary

- A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

For example −

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
```

```
print dict['one']      # Prints value for 'one' key
print dict[2]          # Prints value for 2 key
print tinydict         # Prints complete dictionary
print tinydict.keys()  # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result −

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Data Type Conversion

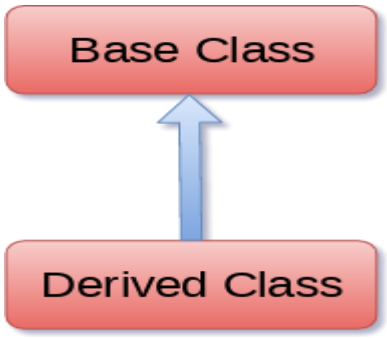
- Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.
- There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)**

	Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)** Converts s to a list.
10	**set(s)** Converts s to a set.
11	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)** Converts s to a frozen set. In Python, frozenset is the same as set except the frozensets are immutable which means that elements from the frozenset cannot be added or removed once created.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.
15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.

Inheritance

- Inheritance is an important aspect of the object-oriented paradigm.
- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class.
- In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.
- Consider the following syntax to inherit a base class into the derived class.



Syntax:

```
class derived-class(base class):
                    <class-suite>
```

- A class can inherit multiple classes by mentioning all of them inside the bracket.

Consider the following syntax.

Syntax:

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):

                    <class - suite>
```

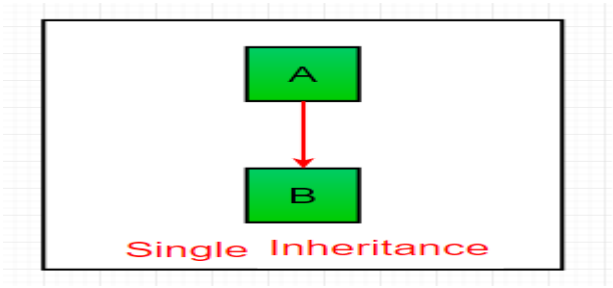
There are different types of inheritance:

1.Simple Inheritance
2.Multi-level Inheritance
3.Multiple Inheritance

4.Hierarchical Inheritance
5.Hybrid Inheritance

1.Simple Inheritance: The process of acquiring the properties of one base class into another single derived class is called “Simple Inheritance”.

i.e



Example1:

```
class Animal:
    def speak(self):
        print("Animal Speaking")

#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")

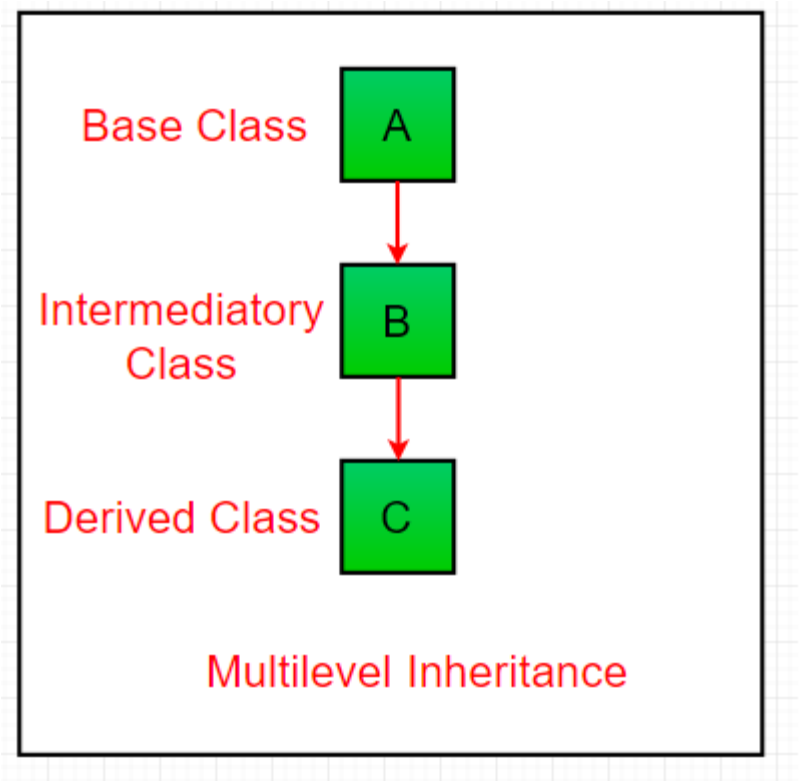
d = Dog()
d.bark()
d.speak()
```

Output:

```
dog barking
Animal Speaking
```

2.Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



The syntax of multi-level inheritance is given below.

Syntax:

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
.
.
```

Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
```

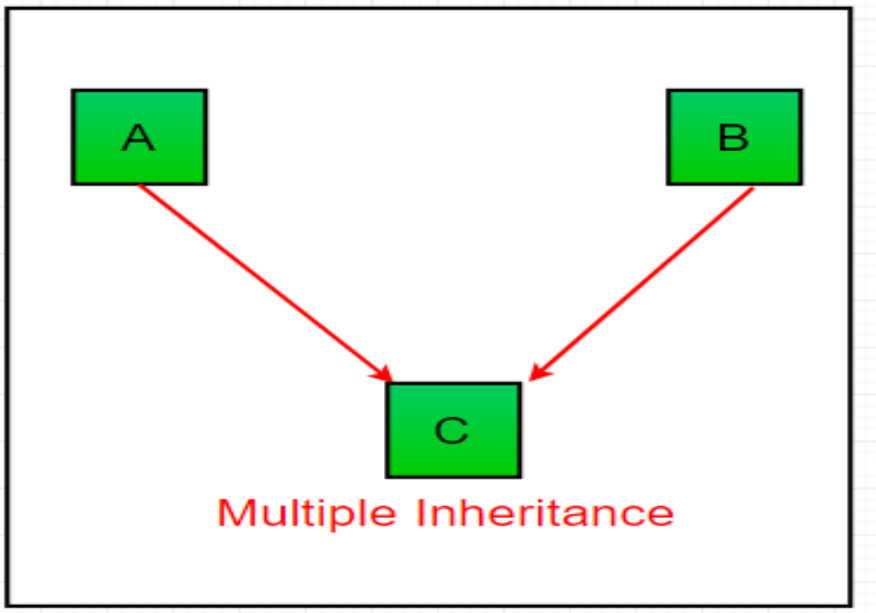
```
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

3.Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.



The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
```

```
class BaseN:
   <class-suite>

class Derived(Base1, Base2, ..... BaseN):
   <class-suite>
```

Example

```
class Calculation1:
   def Summation(self,a,b):
      return a+b;
class Calculation2:
   def Multiplication(self,a,b):
      return a*b;
class Derived(Calculation1,Calculation2):
   def Divide(self,a,b):
      return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

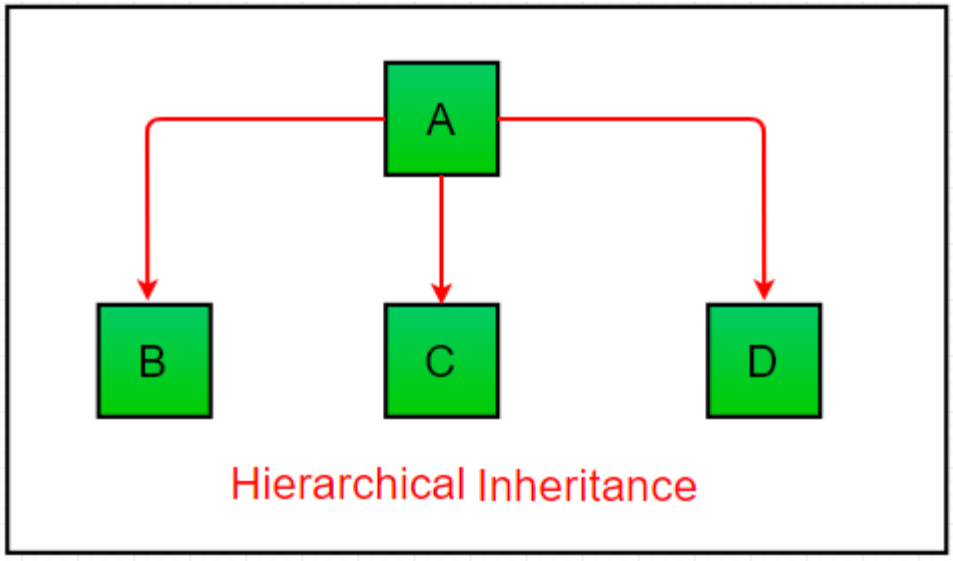
Output:

```
30
200
0.5
```

4.Hierarchical Inheritance

Def: The Procee of deriving multiple classes from a single base class is called “Hierarchical Inheritance”.

i.e



Example:

```
#Hierarchical Inheritance
  class A:
     a=None;
  class B(A):
     b=None;
     def __init__(self):
        self.a=10;
        self.b=20;
     def display_ab(self):
        print("The Variables in Class B is a={0} \t b={1}".format(self.a,self.b))
  class C(A):
     c=None;
     def __init__(self):
        self.a=30;
        self.c=40;
     def display_ac(self):
        print("The Variables in Class C is a={0} \t c={1}".format(self.a,self.c))
  class D(A):
     d=None;
     def __init__(self):
        self.a=50;
        self.d=60;
     def display_ad(self):
        print("The Variables in Class D is a={0} \t d={1}".format(self.a,self.d))

  b1=B()
  b1.display_ab();
  c1=C()
  c1.display_ac()
  d1=D()
  d1.display_ad()
```

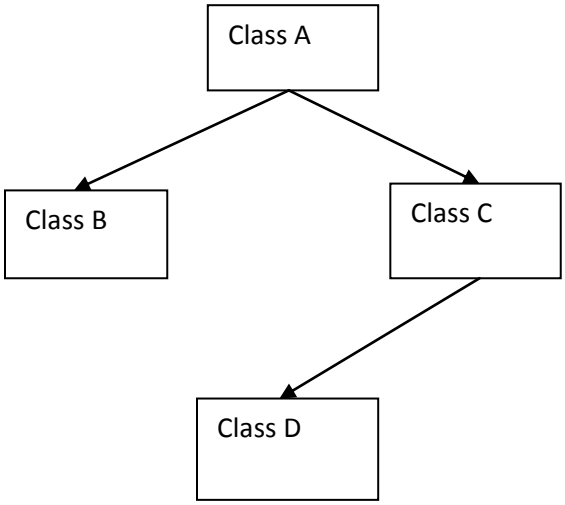
Output:

The Variables in Class B is a=10 b=20
The Variables in Class C is a=30 c=40
The Variables in Class D is a=50 d=60

5.Hybrid Inheritance

Def: The combination of any 2 types of inheritance like Multiple,Hierarchical and Hierarchical,Multilevel Inheritance.

i.e



Example:Write a Python Program to describe Hybrid Inheritance for above diagram.

Code:

```
class A:
   a=10;
class B(A):
   b=20;
   def display_ab(self):
      print("a=",self.a)
      print("b=",self.b)
class C(A):
   c=30;
class D(C):
   d=40;
   def display_acd(self):
      print("a=",self.a)
      print("c=",self.c)
      print("d=",self.d)

obj_b=B()
print("Class B Details are:")
```

```
obj_b.display_ab();
obj_d=D()
print("Class D Details are:")
obj_d.display_acd();
```

Output:

```
Class B Details are:
a= 10
b= 20
Class D Details are:
a= 10
c= 30
d= 40
```

Polymorphism in Python

Polymorphism: The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

Example of inbuilt polymorphic functions :

```
# Python program to demonstrate in-built poly-morphic functions

# len() being used for a string
        print(len("geeks"))

# len() being used for a list
        print(len([10, 20, 30]))
```

Output:

```
5
3
```

Examples of user-defined polymorphic functions :

```
# A simple Python function to demonstrate Polymorphism

def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Output:

```
5
9
```

Polymorphism with class methods:

The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")
    def language(self):
        print("Hindi is the most widely spoken language of India.")
    def type(self):
        print("India is a developing country.")
class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")
    def language(self):
        print("English is the primary language of USA.")
    def type(self):
        print("USA is a developed country.")
obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Output:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
```

India is a developing country.

Washington, D.C. is the capital of USA.

English is the primary language of USA.

USA is a developed country.

Polymorphism with Inheritance:

- In Python, Polymorphism lets us define 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.
- However, it is possible to modify a method in a child class that it has inherited from the parent class.
- This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class.
- In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

```
class Bird:
  def intro(self):
    print("There are many types of birds.")

  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

class ostrich(Bird):
  def flight(self):
    print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()
```

```
obj_ost.intro()
obj_ost.flight()
```

Output:

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Polymorphism with a Function and objects:

- It is also possible to create a function that can take any object, allowing for polymorphism. In this example, let's create a function called "func()" which will take an object which we will name "obj".
- Though we are using the name 'obj', any instantiated object will be able to be called into this function.
- Next, let's give the function something to do that uses the 'obj' object we passed to it. In this case, let's call the three methods, viz., capital(), language() and type(), each of which is defined in the two classes 'India' and 'USA'. Next, let's create instantiations of both the 'India' and 'USA' classes if we don't have them already.
- With those, we can call their action using the same func() function:

Example:

```
def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

Code: Implementing Polymorphism with a Function

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")
```

```
    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

Output:

New Delhi is the capital of India.

Hindi is the most widely spoken language of India.

India is a developing country.

Washington, D.C. is the capital of USA.

English is the primary language of USA.

USA is a developed country.

Abstraction in Python

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. User is familiar with that **"what function does"** but they don't know **"how it does."**

Example:

- In simple words, we all use the smartphone and very much familiar with its functions such as camera, voice-recorder, call-dialing, etc., but we don't know how these operations are happening in the background.
- Let's take another example - When we use the TV remote to increase the volume. We don't know how pressing a key increases the volume of the TV. We only know to press the "+" button to increase the volume.
- That is exactly the abstraction that works in the object-oriented concept

Why Abstraction is Important

- In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity.
- It also enhances the application efficiency. Next, we will learn how we can achieve abstraction using the Python program

Abstraction classes in Python

- In Python, abstraction can be achieved by using abstract classes and interfaces.
- A class that consists of one or more abstract method is called the abstract class.
- Abstract methods do not contain their implementation.
- Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass.
- Abstraction classes are meant to be the blueprint of the other class. An abstract class can be useful when we are designing large functions. An abstract class is also helpful to provide the standard interface for different implementations of components.
- Python provides the **abc** module to use the abstraction in the Python program.

Let's see the following syntax.

Syntax

```
from abc import ABC
class ClassName(ABC):
```

We import the ABC class from the **abc** module.

Abstract Base Classes

An abstract base class is the common application program of the interface for a set of subclasses. It can be used by the third-party, which will provide the implementations such as with plugins. It is also beneficial when we work with the large code-base hard to remember all the classes.

Working of the Abstract Classes

- Unlike the other high-level language, Python doesn't provide the abstract class itself.

- We need to import the abc module, which provides the base for defining Abstract Base classes (ABC). The ABC works by decorating methods of the base class as abstract.
- It registers concrete classes as the implementation of the abstract base. We use the ***@abstractmethod*** decorator to define an abstract method or if we don't provide the definition to the method, it automatically becomes the abstract method.
- Let's understand the following example.

Example -

```
# Python program demonstrate  abstract base class work
    from abc import ABC
    class Car(ABC):
       # abstract method
       def mileage(self):
          pass

    class Tesla(Car):
       def mileage(self):
          print("The mileage is 30kmph")
    class Suzuki(Car):
       def mileage(self):
          print("The mileage is 25kmph ")
    class Duster(Car):
        def mileage(self):
            print("The mileage is 24kmph ")

    class Renault(Car):
       def mileage(self):
             print("The mileage is 27kmph ")

    # Driver code
    t= Tesla ()
    t.mileage()

    r = Renault()
    r.mileage()

    s = Suzuki()
    s.mileage()
    d = Duster()
    d.mileage()
```

Output:

```
The mileage is 30kmph
```

```
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph
```

Explanation -

In the above code, we have imported the **abc module** to create the abstract base class. We created the Car class that inherited the ABC class and defined an abstract method named mileage(). We have then inherited the base class from the three different subclasses and implemented the abstract method differently. We created the objects to call the abstract method.

Let's understand another example.

Let's understand another example.

Example -

```
# Python program to define  abstract class

    from abc import ABC

    class Polygon(ABC):

      # abstract method
      def sides(self):
        pass

    class Triangle(Polygon):


      def sides(self):
        print("Triangle has 3 sides")

    class Pentagon(Polygon):


      def sides(self):
        print("Pentagon has 5 sides")

    class Hexagon(Polygon):

      def sides(self):
        print("Hexagon has 6 sides")

    class square(Polygon):
```

```
    def sides(self):
        print("I have 4 sides")

# Driver code
t = Triangle()
t.sides()

s = square()
s.sides()

p = Pentagon()
p.sides()

k = Hexagon()
K.sides()
```

Output:

```
Triangle has 3 sides
Square has 4 sides
Pentagon has 5 sides`
Hexagon has 6 sides
```

Explanation -

In the above code, we have defined the abstract base class named Polygon and we also defined the abstract method. This base class inherited by the various subclasses. We implemented the abstract method in each subclass. We created the object of the subclasses and invoke the **sides()** method. The hidden implementations for the **sides()** method inside the each subclass comes into play. The abstract method **sides()** method, defined in the abstract class, is never invoked.

Points to Remember

Below are the points which we should remember about the abstract base class in Python.

- An Abstract class can contain the both method normal and abstract method.
- An Abstract cannot be instantiated; we cannot create objects for the abstract class.
- Abstraction is essential to hide the core functionality from the users.

UNIT 2

Data Structures

UNIT II

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 Structures

- **Python** has been used worldwide for different fields such as making websites, artificial intelligence and much more.
- But to make all of this possible, **data** plays a very important role which means that this data should be stored efficiently and the access to it must be timely.
- So how do We achieve this? We use something called Data Structures.

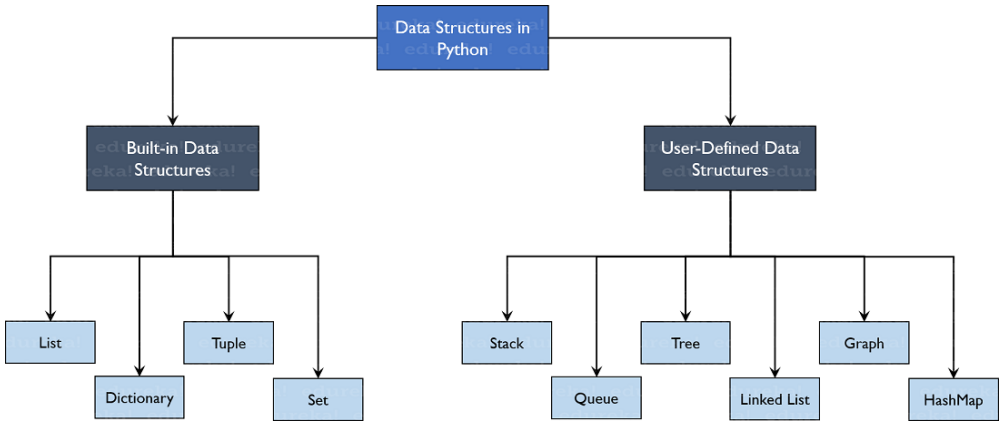
What is a Data Structure

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 us to store collections of data, relate them and perform operations on them accordingly.

Definition: Data structures are "containers" that organize and group data according to type.

Types of Data Structures in Python

Python has **implicit** support for Data Structures which enable us to store and access data. These structures are called List, Dictionary, Tuple and Set.



Linear Data Structures

Def: The data structure where data items are organized sequentially or linearly is called a linear data structure.

- In the linear data structure, data items are organized linearly, one after another and only one data item can be reached.

Example of linear data structures:

- Array: series of a collection of elements(array elements) all of which are the same type, and has fixed size.
- Linked List: data structure consisting of a collection of nodes that represent a sequence. Linked lists are useful for dynamic memory allocation.
- Queue: data structure where one end is used to insert data (enqueue) and the other end is used to remove data (dequeue). Queue order is FIFO(First In First Out).
- Stack: a data structure that follows a particular order for performing operations. The order is LIFO(Last In First Out) and it is a dynamic and constantly changing object.

Non-linear Data Structures

Def:Opposite of linear data structure, is a non-linear data structure,i.e data items are not arranged in a sequential structure and are connected to several other data items to represent a specific relationship.

Example of non-linear data structures:

- Tree: a data structure that may have multiple relations among its nodes that simulates a hierarchical tree structure.
- Heap: sometimes called as the binary heap is a binary tree data structure that satisfies the heap property.
- Hash Table: or hash map is a data structure that map keys to values and using a hash function it computes an index from which value can be found.
- Graph: set of items connected by edges.

Built-in Data Structures

As the name suggests, these Data Structures are built-in with Python which makes programming easier and helps programmers use them to obtain solutions faster. Let's discuss each of them in detail.

Lists

- Lists are used to store data of different data types in a sequential manner.
- There are addresses assigned to every element of the list, which is called as 'Index'. The index value starts from 0 and goes on until the last element called the '**Positive index'**.
- There is also **negative indexing** which starts from -1 enabling you to access elements from the last to first.

Let us now understand lists better with the help of an example program.

Creating a list

- To create a list, use the square brackets and add elements into it accordingly.
- If you do not pass any elements inside the square brackets, you get an empty list as the output.

Example

```
my_list = [] #create empty list
print(my_list)
my_list = [1, 2, 3, 'example', 3.132] #creating list with data
print(my_list)
```

Output:

[]
[1, 2, 3, 'example', 3.132]

Adding Elements

Adding the elements in the list can be achieved using the append(), extend() and insert() functions.

- The append() function adds all the elements passed to it as a single element.
- The extend() function adds the elements one-by-one into the list.
- The insert() function adds the element passed to the index value and increase the size of the list too.

Example:

```
my_list = [1, 2, 3]
print(my_list)
my_list.append([555, 12]) #add as a single element
print(my_list)
my_list.extend([234, 'more_example']) #add as different elements
print(my_list)
my_list.insert(1, 'insert_example') #add element i
print(my_list)
```

Output:

[1, 2, 3]
[1, 2, 3, [555, 12]]
[1, 2, 3, [555, 12], 234, ‘more_example’]
[1, ‘insert_example’, 2, 3, [555, 12], 234, ‘more_example’]

Deleting Elements

- To delete elements, use the *del* keyword which is built-in into Python but this does not return anything back to us.
- If you want the element back, you use the pop() function which takes the index value.
- To remove an element by its value, you use the remove() function.
- To delete all elements from the List we can use the clear() function.

Example:

```
my_list = [1, 2, 3, 'example', 3.132, 10, 30]
del my_list[5] #delete element at index 5
print(my_list)
my_list.remove('example') #remove element with value
print(my_list)
a = my_list.pop(1) #pop element from list
print('Popped Element: ', a, ' List remaining: ', my_list)
my_list.clear() #empty the list
print(my_list)
```

Output:
[1, 2, 3, ‘example’, 3.132, 30]
[1, 2, 3, 3.132, 30]
Popped Element: 2 List remaining: [1, 3, 3.132, 30]
[]

Accessing Elements

- Accessing elements is the same as accessing Strings in Python.
- We pass the index values and hence can obtain the values as needed.

Example:

```
my_list = [1, 2, 3, 'example', 3.132, 10, 30]
for element in my_list: #access elements one by one
    print(element)
```

- ```
  print(my_list) #access all elements
  print(my_list[3]) #access index 3 element
  print(my_list[0:2]) #access elements from 0 to 1 and exclude 2
  print(my_list[::-1]) #access elements in reverse
  ```

Output:
1
2
3
example
3.132
10
30
[1, 2, 3, ‘example’, 3.132, 10, 30]
example
[1, 2]
[30, 10, 3.132, ‘example’, 3, 2, 1]

Other Functions

You have several other functions that can be used when working with lists.

- The len() function returns to us the length of the list.
- The index() function finds the index value of value passed where it has been encountered the first time.
- The count() function finds the count of the value passed to it.
- The sorted() and sort() functions do the same thing, that is to sort the values of the list. The sorted() has a return type whereas the sort() modifies the original list.
  ```

Example:

```
my_list = [1, 2, 3, 10, 30, 10]
print(len(my_list)) #find length of list
print(my_list.index(10)) #find index of element that occurs first
print(my_list.count(10)) #find count of the element
print(sorted(my_list)) #print sorted list but not change original
my_list.sort(reverse=True) #sort original list
print(my_list)
```

Output:

```
6
3
2
[1, 2, 3, 10, 10, 30]
[30, 10, 10, 3, 2, 1]
```

Slice Methods:

- The slice() function returns a slice object that can use used to slice strings, lists, tuple etc.

The syntax of slice() is:

```
slice(start, stop, step)
```

slice() Parameters

slice() can take three parameters:

start (optional) - Starting integer where the slicing of the object starts. Default to None if not provided.

stop - Integer until which the slicing takes place. The slicing stops at index

'stop -1' (last element).

step (optional) - Integer value which determines the increment between each index for slicing. Defaults to None if not provided.

Example 1: Create a slice object for slicing

```
# contains indices (0, 1, 2)
result1 = slice(3)
print(result1)
# contains indices (1, 3)
result2 = slice(1, 5, 2)
print(slice(1, 5, 2))
```

Output

```
slice(None, 3, None)
slice(1, 5, 2)
```

Here, result1 and result2 are slice objects.

Example 2: Get substring using slice object

```
# Program to get a substring from the given string
py_string = 'Python'
# stop = 3
# contains 0, 1 and 2 indices
slice_object = slice(3)
print(py_string[slice_object])  # Pyt
# start = 1, stop = 6, step = 2
# contains 1, 3 and 5 indices
slice_object = slice(1, 6, 2)
print(py_string[slice_object])   # yhn
```

Output

```
Pyt
yhn
```

Example 3: Get substring using negative index

```
py_string = 'Python'
# start = -1, stop = -4, step = -1
# contains indices -1, -2 and -3
slice_object = slice(-1, -4, -1)
print(py_string[slice_object])   # noh
```

Output

```
noh
```

Example 4: Get sublist and sub-tuple

```
py_list = ['P', 'y', 't', 'h', 'o', 'n']
py_tuple = ('P', 'y', 't', 'h', 'o', 'n')
# contains indices 0, 1 and 2
slice_object = slice(3)
print(py_list[slice_object]) # ['P', 'y', 't']

# contains indices 1 and 3
slice_object = slice(1, 5, 2)
print(py_tuple[slice_object]) # ('y', 'h')
```

Output

```
['P', 'y', 't']
('y', 'h')
```

Example 5: Get sublist and sub-tuple using negative index

```
py_list = ['P', 'y', 't', 'h', 'o', 'n']
py_tuple = ('P', 'y', 't', 'h', 'o', 'n')
# contains indices -1, -2 and -3
slice_object = slice(-1, -4, -1)
print(py_list[slice_object])  # ['n', 'o', 'h']
# contains indices -1 and -3
slice_object = slice(-1, -5, -2)
print(py_tuple[slice_object]) # ('n', 'h')
```

Output

```
['n', 'o', 'h']
('n', 'h')
```

Example 6: Using Indexing Syntax for Slicing

The slice object can be substituted with the indexing syntax in Python.

We can alternately use the following syntax for slicing:

obj[start:stop:step]

For example,

```
py_string = 'Python'
# contains indices 0, 1 and 2
print(py_string[0:3])  # Pyt
# contains indices 1 and 3
print(py_string[1:5:2]) # yh
```

Output

Pyt

Yh

Tuples in Python

- A Tuple is a collection of Python objects separated by commas.
- In someways a tuple is similar to a list in terms of indexing, nested objects and repetition but a tuple is **"immutable"** unlike lists which are mutable.

Creating Tuples

```
# An empty tuple
empty_tuple = (,)
print (empty_tuple)
```

Output:

()

 # Creating non-empty tuples

```
 # One way of creation
tup = 'python', 'geeks'
print(tup)
# Another for doing the same
tup = ('python', 'geeks')
print(tup)
```

Output

('python', 'geeks')

('python', 'geeks')

Concatenation of Tuples

```
# Code for concatenating 2 tuples

tuple1 = (0, 1, 2, 3)

tuple2 = ('python', 'geek')

# Concatenating above two

print(tuple1 + tuple2)
```

Output:

(0, 1, 2, 3, 'python', 'geek')

Nesting of Tuples

```
# Code for creating nested tuples

tuple1 = (0, 1, 2, 3)

tuple2 = ('python', 'geek')

tuple3 = (tuple1, tuple2)

print(tuple3)
```

Output :

((0, 1, 2, 3), ('python', 'geek'))

Repetition in Tuples

```
# Code to create a tuple with repetition

tuple3 = ('python',)*3

print(tuple3)
```

Output

('python', 'python', 'python')

Note: Try the above without a comma and check. We will get tuple3 as a string ‘pythonpythonpython’.

Immutable Tuples

```
#code to test that tuples are immutable
    tuple1 = (0, 1, 2, 3)
    tuple1[0] = 4
    print(tuple1)
```

Output

```
Traceback (most recent call last):
  File "e0eaddff843a8695575daec34506f126.py", line 3, in
    tuple1[0]=4
TypeError: 'tuple' object does not support item assignment
```

Slicing in Tuples

```
    # code to test slicing
    tuple1 = (0 ,1, 2, 3)
    print(tuple1[1:])
    print(tuple1[::-1])
    print(tuple1[2:4])
```

Output

```
(1, 2, 3)
(3, 2, 1, 0)
(2, 3)
```

Deleting a Tuple

```
    # Code for deleting a tuple
    tuple3 = ( 0, 1)
    del tuple3
    print(tuple3)
```

Error:

```
Traceback (most recent call last):
  File "d92694727db1dc9118a5250bf04dafbd.py", line 6, in <module>
    print(tuple3)
NameError: name 'tuple3' is not defined
```

Output:

```
(0, 1)
```

Finding Length of a Tuple

```
# Code for printing the length of a tuple
tuple2 = ('python', 'geek')
print(len(tuple2))
```

Output

```
2
```

Converting list to a Tuple

```
# Code for converting a list and a string into a tuple
list1 = [0, 1, 2]
print(tuple(list1))
print(tuple('python')) # string 'python'
```

Output

```
(0, 1, 2)
('p', 'y', 't', 'h', 'o', 'n')
```

Sets in Python

- Sets are a collection of unordered elements that are unique.
- Meaning that even if the data is repeated more than one time, it would be entered into the set only once.

Creating a set

Sets are created using the "flower braces" but instead of adding key-value pairs, we just pass values to it.

```
my_set = {1, 2, 3, 4, 5, 5, 5} #create set
print(my_set)
```

Output:

{1, 2, 3, 4, 5}

Adding elements

To add elements, you use the add() function and pass the value to it.

```
my_set = {1, 2, 3}

my_set.add(4) #add element to set

print(my_set)
```

Output:

{1, 2, 3, 4}

Operations in sets

The different operations on set such as union, intersection and so on are shown below.

```
my_set = {1, 2, 3, 4}

my_set_2 = {3, 4, 5, 6}

print(my_set.union(my_set_2),          '----------', my_set | my_set_2)

print(my_set.intersection(my_set_2),    '----------', my_set & my_set_2)

print(my_set.difference(my_set_2),      '----------', my_set - my_set_2)

print(my_set.symmetric_difference(my_set_2), '----------', my_set ^ my_set_2)

my_set.clear()

print(my_set)
```

- The union() function combines the data present in both sets.
- The intersection() function finds the data present in both sets only.
- The difference() function deletes the data present in both and outputs data present only in the set passed.
- The symmetric_difference() does the same as the difference() function but outputs the data which is remaining in both sets.

Output:

{1, 2, 3, 4, 5, 6} ———- {1, 2, 3, 4, 5, 6}

{3, 4} ———- {3, 4}

{1, 2} ———- {1, 2}

{1, 2, 5, 6} ———- {1, 2, 5, 6}

Dictionary

- Dictionaries are used to store **key-value** pairs.
- To understand better, think of a phone directory where hundreds and thousands of names and their corresponding numbers have been added.
- Now the constant values here are Name and the Phone Numbers which are called as the keys.
- And the various names and phone numbers are the values that have been fed to the keys.
- If you access the values of the keys, you will obtain all the names and phone numbers. So that is what a key-value pair is.
- And in Python, this structure is stored using Dictionaries. Let us understand this better with an example program.

Creating a Dictionary

Dictionaries can be created using the "flower braces" or using the dict() function. You need to add the key-value pairs whenever you work with dictionaries.

```
my_dict = {} #empty dictionary

print(my_dict)

my_dict = {1: 'Python', 2: 'Java'} #dictionary with elements

print(my_dict)
```

Output:
{}
{1: ‘Python’, 2: ‘Java’}

Changing and Adding key, value pairs

- To change the values of the dictionary, you need to do that using the keys.
- So, you firstly access the key and then change the value accordingly.
- To add values, you simply just add another key-value pair as shown below.

```
my_dict = {'First': 'Python', 'Second': 'Java'}

print(my_dict)

my_dict['Second'] = 'C++' #changing element

print(my_dict)

my_dict['Third'] = 'Ruby' #adding key-value pair

print(my_dict)
```

Output:
{‘First’: ‘Python’, ‘Second’: ‘Java’}
{‘First’: ‘Python’, ‘Second’: ‘C++’}
{‘First’: ‘Python’, ‘Second’: ‘C++’, ‘Third’: ‘Ruby’}

Deleting key, value pairs

- To delete the values, you use the pop() function which returns the value that has been deleted.
- To retrieve the key-value pair, we use the popitem() function which returns a tuple of the key and value.
- To clear the entire dictionary, we use the clear() function.

```
my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}

a = my_dict.pop('Third') #pop element

print('Value:', a)

print('Dictionary:', my_dict)

b = my_dict.popitem() #pop the key-value pair

print('Key, value pair:', b)

print('Dictionary', my_dict)
```

Output:

Value: Ruby
Dictionary: {‘First’: ‘Python’, ‘Second’: ‘Java’}

Key, value pair: (‘Second’, ‘Java’)
Dictionary {‘First’: ‘Python’}

{}

Accessing Elements

- We can access elements using the keys only.
- We can use either the get() function or just pass the key values and you will be retrieving the values.

```
my_dict = {'First': 'Python', 'Second': 'Java'}

print(my_dict['First']) #access elements using keys

print(my_dict.get('Second'))
```

Output:
Python
Java

Other Functions

- **We have different functions which return to us the keys or the values of the key-value pair accordingly to the keys(), values(), items() functions accordingly**.

```
my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}

print(my_dict.keys()) #get keys

print(my_dict.values()) #get values

print(my_dict.items()) #get key-value pairs

print(my_dict.get('First'))
```

Output:

dict_keys([‘First’, ‘Second’, ‘Third’])

dict_values([‘Python’, ‘Java’, ‘Ruby’])

dict_items([(‘First’, ‘Python’), (‘Second’, ‘Java’), (‘Third’, ‘Ruby’)])

Python

Sequences in Python

Sequences are the essential building block of python programming and are used on a daily basis by python developers.

There are seven types of sequences in Python.

- Unicode string
- strings
- Lists
- Tuples
- Byte arrays
- Buffers
- Xrange objects

Out of these seven, three are the most popular. These three are: –

- Lists
- Tuples
- Strings

Some common operations for the sequence type object can work on both mutable and immutable sequences. Some of the operations are as follows –

Sr.No.	Operation/Functions & Description
1	**x in seq** True, when x is found in the sequence seq, otherwise False
2	**x not in seq** False, when x is found in the sequence seq, otherwise True
3	**x + y** Concatenate two sequences x and y
4	**x * n or n * x** Add sequence x with itself n times
5	**seq[i]** ith item of the sequence.
6	**seq[i:j]** Slice sequence from index i to j
7	**seq[i:j:k]** Slice sequence from index i to j with step k
8	**len(seq)** Length or number of elements in the sequence
9	**min(seq)** Minimum element in the sequence
10	**max(seq)** Maximum element in the sequence
11	**seq.index(x[, i[, j]])**

	Index of the first occurrence of x (in the index range i and j)
12	**seq.count(x)** Count total number of elements in the sequence
13	**seq.append(x)** Add x at the end of the sequence
14	**seq.clear()** Clear the contents of the sequence
15	**seq.insert(i, x)** Insert x at the position i
16	**seq.pop([i])** Return the item at position i, and also remove it from sequence. Default is last element.
17	**seq.remove(x)** Remove first occurrence of item x
18	**seq.reverse()** Reverse the list

Example Code

```
myList1 = [10, 20, 30, 40, 50]
myList2 = [56, 42, 79, 42, 85, 96, 23]

if 30 in myList1:
   print('30 is present')

if 120 not in myList1:
   print('120 is not present')

print(myList1 + myList2) #Concatinate lists
print(myList1 * 3) #Add myList1 three times with itself
print(max(myList2))
print(myList2.count(42)) #42 has two times in the list
```

```
print(myList2[2:7])
print(myList2[2:7:2])

myList1.append(60)
print(myList1)

myList2.insert(5, 17)
print(myList2)

myList2.pop(3)
print(myList2)
myList1.reverse()
print(myList1)

myList1.clear()
print(myList1)
```

Output

```
30 is present
120 is not present
[10, 20, 30, 40, 50, 56, 42, 79, 42, 85, 96, 23]
[10, 20, 30, 40, 50, 10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
96
2
[79, 42, 85, 96, 23]
[79, 85, 23]
[10, 20, 30, 40, 50, 60]
[56, 42, 79, 42, 85, 17, 96, 23]
[56, 42, 79, 85, 17, 96, 23]
[60, 50, 40, 30, 20, 10]
[]
```

Comprehensions in Python

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined.

Python supports the following 3 types of comprehensions:

- List Comprehensions
- Dictionary Comprehensions
- Set Comprehensions
- Generator Comprehensions

List Comprehensions:

- List Comprehensions provide an elegant way to create new lists.
- The following is the basic structure of a list comprehension:

Syntax:

output_list = [output_exp for var in input_list if (var satisfies this condition)]

Note: List comprehension may or may not contain an if condition. List comprehensions can contain multiple for (nested list comprehensions).

Example #1: Suppose we want to create an output list which contains only the even numbers which are present in the input list. Let's see how to do this using for loops and list comprehension and decide which method suits better.

 # Constructing output list WITHOUT Using List comprehensions

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
output_list = []
# Using loop for constructing output list
for var in input_list:
   if var % 2 == 0:
      output_list.append(var)
print("Output List using for loop:", output_list)
```

Output:

Output List using for loop: [2, 4, 4, 6]

 # Using List comprehensions for constructing output list

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
Output_list = [var for var in input_list if var % 2 == 0]
print("Output List using list comprehensions:",
                        Output_list)
```

Output:

Output List using list comprehensions: [2, 4, 4, 6]

Example #2: Suppose we want to create an output list which contains squares of all the numbers from 1 to 9. Let's see how to do this using for loops and list comprehension.

 # Constructing output list using for loop

```
output_list = []
for var in range(1, 10):
   output_list.append(var ** 2)
```

```
print("Output List using for loop:", output_list)
```

Output:

Output List using for loop: [1, 4, 9, 16, 25, 36, 49, 64, 81]

 # Constructing output list using list comprehension

```
list_using_comp = [var**2 for var in range(1,10)]
print("Output List using list comprehension:",list_using_comp)
```

Output:

Output List using list comprehension: [1, 4, 9, 16, 25, 36, 49, 64, 81]

Dictionary Comprehensions:

- Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions.
- The basic structure of a dictionary comprehension looks like below.

Syntax:

output_dict = {key:value for (key, value) in iterable if (key, value satisfy this condition)}

Example #1: Suppose we want to create an output dictionary which contains only the odd numbers that are present in the input list as keys and their cubes as values. Let's see how to do this using for loops and dictionary comprehension.

```
input_list = [1, 2, 3, 4, 5, 6, 7]

output_dict = {}
```

 # Using loop for constructing output dictionary

```
output_dict = {}
for val in input_list:
   if val % 2 != 0:
      output_dict[val] = val**3
print("Output Dictionary using for loop:", output_dict )
```

Output:

Output Dictionary using for loop: {1: 1, 3: 27, 5: 125, 7: 343}

 # Using Dictionary comprehensions for constructing output dictionary

```
input_list = [1,2,3,4,5,6,7]
dict_using_comp = {val:val ** 3 for val in input_list if val % 2 != 0}
print("Output Dictionary using dictionary comprehensions:",
                                dict_using_comp)
```

Output:

Output Dictionary using dictionary comprehensions: {1: 1, 3: 27, 5: 125, 7: 343}

Example #2: Given two lists containing the names of states and their corresponding capitals, construct a dictionary which maps the states with their respective capitals. Let's see how to do this using for loops and dictionary comprehension.

```
state = ['Gujarat', 'Maharashtra', 'Rajasthan']
capital = ['Gandhinagar', 'Mumbai', 'Jaipur']
output_dict = {}

# Using loop for constructing output dictionary
for (key, value) in zip(state, capital):
    output_dict[key] = value
print("Output Dictionary using for loop:",output_dict)
```

Output:

```
Output Dictionary using for loop: {'Gujarat': 'Gandhinagar',
                                   'Maharashtra': 'Mumbai',
                                   'Rajasthan': 'Jaipur'}
```

 # Using Dictionary comprehensions for constructing output dictionary

```
state = ['Gujarat', 'Maharashtra', 'Rajasthan']
capital = ['Gandhinagar', 'Mumbai', 'Jaipur']
dict_using_comp = {key:value for (key, value) in zip(state, capital)}
print("Output Dictionary using dictionary comprehensions:",
                                   dict_using_comp)
```

Output:

```
Output Dictionary using dictionary comprehensions: {'Rajasthan': 'Jaipur',
                                                    'Maharashtra': 'Mumbai',
                                                    'Gujarat': 'Gandhinagar'}
```

Set Comprehensions:

- Set comprehensions are pretty similar to list comprehensions.
- The only difference between them is that set comprehensions use curly brackets { }.

Let's look at the following example to understand set comprehensions.

Example #1 : Suppose we want to create an output set which contains only the even numbers that are present in the input list. Note that set will discard all the duplicate values. Let's see how we can do this using for loops and set comprehension.

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
output_set = set()

# Using loop for constructing output set
for var in input_list:
    if var % 2 == 0:
        output_set.add(var)
print("Output Set using for loop:", output_set)
```

Output:

Output Set using for loop: {2, 4, 6}

 # Using Set comprehensions for constructing output set

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
set_using_comp = {var for var in input_list if var % 2 == 0}
print("Output Set using set comprehensions:", set_using_comp)
```

Output:

Output Set using set comprehensions: {2, 4, 6}

Generator Comprehensions

- The syntax of Generator Expression is similar to List Comprehension except it uses parentheses ( ) instead of square brackets [ ].
- A **generator** is a special kind of iterator, which stores the instructions for how to *generate* each of its members, in order, along with its current state of iterations.
- It generates each member, one at a time, only as it is requested via iteration.
- Generators are special iterators in Python which returns the generator object.
- The point of using it, is to generate a sequence of items without having to store them in memory and this is why we can use Generator only once.

Example: Here, we have created a List *num_cube_lc* using List Comprehension and Generator Expression is defined as *num_cube_generator*.

```
#Generator Expression
num_cube_lc=[n**3 for n in range(1,11) if n%2==0]    #List Comprehension
num_cube_generator=(num**3 for num in range(1,11) if num%2==0)  #Generator Expression
print(f"List Comprehension = {num_cube_lc}")
print(f"Generator Expression = {num_cube_generator}")
#sum(num_cube_generator)
print(f"Sum = {sum(num_cube_generator)}")
```

Note: Recall that a list readily stores all of its members; you can access any of its contents via indexing. A generator, on the other hand, *does not store any items*. Instead, it stores the instructions for generating each of its members, and stores its iteration state; this means that the generator will know if it has generated its second member, and will thus generate its third member the next time it is iterated on.

The whole point of this is that you can use a generator to produce a long sequence of items, without having to store them all in memory.

The `range` generator

An extremely popular built-in generator is `range`, which, given the values:

- 'start' (inclusive, default=0)
- 'stop' (exclusive)
- 'step' (default=1)

will generate the corresponding sequence of integers (from start to stop, using the step size) upon iteration

Strings and Slicing

Strings

- In Python, string is a contiguous sequence of Unicode Characters.
- In the computer, everything is stored in binary format, i.e. 0(zero)'s and 1(one)'s. The elementary storage unit is called a bit, which can store either zero or one.
- To represent various letters, symbols and numbers in different languages, we need 16 bits.
- Each different sequence of the 16 bits represent one symbol like A or B or C etc.

- This system of representing the various existing letters, numbers and symbols (+ _ - $ % @ ! etc) is called UNICODE.
- Strings are represented with prefixing and suffixing the characters with quotation marks (either single quotes (') or double quotes (")).
- An individual character within a string is accessed using an index. The index should always be an integer (positive or negative).
- The index starts from 0 to n-1, where n is the number of characters in the string.
- The contents of the string cannot be changed after it is created.
- The return value of the Python input() statement is, by default, a string.
- A string of length 1 can be treated as a character.
- [Hint: Some of the words which appear in violet are links. Click to know more about them.]

- Take string as input from the console using input() function, print the given input string as shown in the example.
- We have 5 types of operations which can be performed on strings in Python

1. Indexing
2. Slicing
3. Concatenation
4. Repetition
5. Membership

1.Indexing

- We can access a String by using its index. The index is enclosed in square brackets [] in Python. Index value always starts with 0.
- We have two types of indexing in Python. If the index starts from the first character of a string, then it is Positive Index and it starts with 0.
- If the index starts from the last character of a String, then it is negative Index and its starts with -1
- Positive indexing helps in accessing the string from the beginning.
- Negative indexing helps in accessing the string from the end.

Let us consider the below example:

```
a = "HELLO"
print(a[0]) # prints the 0th index value
  'H'
```

Here at '0'th index 'H' is available, so we traverse from left to right.

```
print(a[-1]) # prints -1 index value
  'O'
```

Here at '-1' index 'O' is available to traverse from right to left.

2.Slicing

- Python provides many ways to extract a part of a string using a concept called “Slicing”.
- Slicing makes it possible to access parts (segments) of the string by providing a startIndex and an endIndex.

The syntax for using the slice operation on a string is:

[startIndex : endIndex : step]#where is the string variable#startIndex, endIndex and step are all optional

Example: In the below code: slice[start : stop : step] operator extracts sub string from the string.

- A segment of a string is called a slice.
- The indexes start and stop should be valid.
- step is the increment or decrement . A positive step will travel left to right and increments the index by step.
- A negative step will travel from right to left and decrements the index by step.
- If we are not providing starting of index position in Slice [ ] operator then interpreter takes starting index zero(0) as default.
- If end index is not specified for slice [ ] operator then Interpreter takes the end of String as default stop index.

Example1:

```
lang = "Python"
print(lang[0:]) # means it prints 0th index position to ending index of String.
Python
print(lang[:6]) # means it prints 0th index to (n-1) i.e. 6-1 = 5th position of String.
Python
```

Example2:

```
a = "HELLO"
Print(a[0:4]) # index starts from 0 and ends before 4 i.e.3.
HELL
Print(a[ :3]) # prints 0th index to 2nd index(3 - 1)
HEL
Print(a[0: ]) # Starts at 0 prints upto last index.
HELLO
```

Note:In the following example start and stop are not given, so they will be defaulted to beginning and end of the string.

- The step is positive indicating left to right traversal and increment is 1.

Example:

```
a = "Python"
print (a[::1])
```

Output:

Python

Note:

- In the following example start and stop are not given, so they will be defaulted to end and beginning of the string.
- The step is negative indicating right to left traversal and decrement is 1.

Example:

```
a = "Python"
print(a[::-1])
```

Output:

nohtyP

Let us take another example:

Example:

```
a = "Python"
print(a[-1::-3])
```

Output:

'nt'

Note: Here we are indicating start as -1 and stop not specified means traverse from right to left till start of the string and step is -3 means decrement is done by 3 from right to left of a string.

Example:

```
a = "Python"
print(a[4:1:-1])
```

The result is

Output:

'oht'

Note: Here we are indicating start as 4 and stop as 1 and index as -1. So it will start index 4(o) and will traverse right to left till index 2(t).

Example:

```
a = "Python"
print(a[2:5:-1])
```

Output:

" (Null string)

Note: Here we are indicating start as 2 and stop as 5 (which implies that the direction as left to right, but the step is -1 which means the direction is right to left.

Because this is not possible so it returns a null string

3.Cancatination

- String Concatenation is the technique of **combining two strings**. String Concatenation can be done using many ways.
  We can perform string concatenation using following ways:

1. Using + operator
2. Using join() method
3. Using % operator
4. Using format() function
5. Using , (comma)

Using + Operator

It's very easy to use + operator for string concatenation. This operator can be used to add multiple strings together. However, the arguments must be a string.

Note: Strings are immutable, therefore, whenever it is concatenated, it is assigned to a new variable.

Example:

```
  # Python program to demonstrate string concatenation

 var1 = "Hello"
var2 = "World"

# join() method is used to combine the strings
print("".join([var1, var2]))

# join() method is used here to combine
# the string with a separator Space(" ")
```

```
var3 = " ".join([var1, var2])

print(var3)
```

Output

Hello World

Here, the variable var1 stores the string “Hello ” and variable var2 stores the string “World”. The + Operator combines the string that is stored in the var1 and var2 and stores in another variable var3.

Using join() Method

The join() method is a string method and returns a string in which the elements of sequence have been joined by str separator.

Example:

```
# Python program to demonstrate string concatenation

var1 = "Hello"
var2 = "World"

# % Operator is used here to combine the string
print("% s % s" % (var1, var2))
```

Output

HelloWorld

Hello World

In the above example, the variable var1 stores the string “Hello” and variable var2 stores the string “World”. The join() method combines the string that is stored in the var1 and var2. The join method accepts only the list as it’s argument and list size can be anything. We can store the combined string in another variable var3 which is separated by space.

Using % Operator

We can use % operator for string formatting, it can also be used for string concatenation. It’s useful when we want to concatenate strings and perform simple formatting.

Example:

```
# Python program to demonstrate string concatenation
        var1 = "Hello"
        var2 = "World"
```

```
# % Operator is used here to combine the string
print("% s % s" % (var1, var2))
```

Output

Hello World

Here, the % Operator combine the string that is stored in the var1 and var2. The %s denotes string data type. The value in both the variable is passed to the string %s and becomes “Hello World”.

Using format() function

str.format() is one of the string formatting methods in Python, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

Example:

```
# Python program to demonstrate string concatenation

 var1 = "Hello"
var2 = "World"

# format function is used here to
# combine the string
print("{} {}".format(var1, var2))

# store the result in another variable
var3 = "{} {}".format(var1, var2)
 print(var3)
```

Output

Hello World

Hello World

Here, the format() function combines the string that is stored in the var1 and var2 and stores in another variable var3. The curly braces {} are used to set the position of strings. The first variable stores in the first curly braces and second variable stores in the second curly braces. Finally it prints the value “Hello World”.

Using , (comma)

- “,” is a great alternative to string concatenation using “+”. when you want to include a single whitespace.

Example:

```
# Python program to demonstrate string concatenation
```

```
var1 = "Hello"
var2 = "World"

# , to combine data types with a single whitespace.
print(var1, var2)
```

Output

Hello World

Use , when you want to combine data types with a single whitespace in between.

4.Repetition

- Sometimes we need to repeat the string in the program, and we can do this easily by using the **repetition operator in Python**.
- The repetition operator is denoted by a '*' symbol and is useful for repeating strings to a certain length.

Example:

```
str = 'Python program'

print(str*3)
```

Output

Python programPython programPython program

- Similarly, it is also possible to repeat any part of the string by slicing:

Example

```
str = 'Python program'

print(str[7:9]*3) #Repeats the seventh and eighth character three times
```

Output

prprpr

5. Membership

- Membership operators are operators used to validate the membership of a value. It tests for membership in a sequence, such as strings, lists, or tuples.

in operator: The 'in' operator is used to check if a value exists in a sequence or not.

- Evaluates to true if it finds a variable in the specified sequence and false otherwise.

```
# Python program to illustrate  Finding common member in list  using 'in' operator
list1=[1,2,3,4,5]
list2=[6,7,8,9]
for item in list1:
    if item in list2:
```

```
        print("overlapping")
    else:
       print("not overlapping")
```

Output:

```
not overlapping
```

'not in' operator- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Example:

```
# Python program to illustrate not 'in' operator
    x = 24
    y = 20
    list = [10, 20, 30, 40, 50 ];

    if ( x not in list ):
      print("x is NOT present in given list")
    else:
      print("x is  present in given list")

    if ( y in list ):
      print("y is present in given list")
    else:
      print("y is NOT present in given list")
```

UNIT 3

Arrays, Searching, Sorting

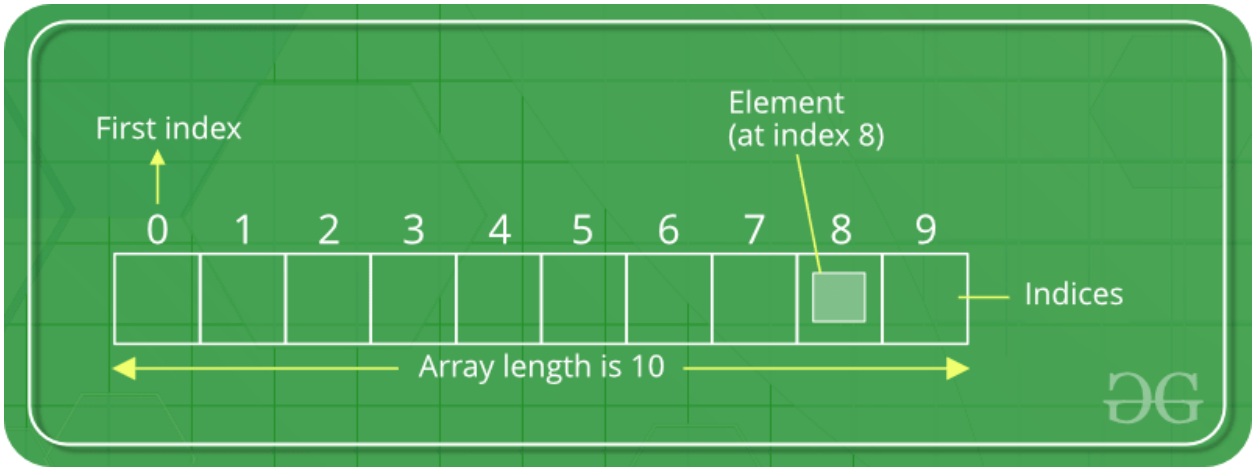
UNIT -III

Arrays - Overview, Types of Arrays, Operations on Arrays, Arrays vs List.
Searching - Linear Search and Binary Search.
Sorting - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort

Arrays in Python

Overview:

- An array is a collection of items stored at contiguous memory locations.
- The idea is to store multiple items of the same type together.
- This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). For simplicity, we can think of an array a fleet of stairs where on each step is placed a value (let's say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on.
- Array can be handled in Python by a module named **array**. They can be useful when we have to manipulate only a specific data type values.
- A user can treat lists as arrays. However, user cannot constraint the type of elements stored in a list. If you create arrays using the **array** module, all elements of the array must be of the same type.



Creating a Array

Array in Python can be created by importing array module. **array(*data_type, value_list*)** is used to create an array with data type and value list specified in its arguments.

```
# Python program to demonstrate Creation of Array

# importing "array" for array creations
import array as arr

# creating an array with integer type
a = arr.array('i', [1, 2, 3])

# printing original array
print ("The new created array is : ", end =" ")
for i in range (0, 3):
    print (a[i], end =" ")
print()

# creating an array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

# printing original array
print ("The new created array is : ", end =" ")
for i in range (0, 3):
    print (b[i], end =" ")
```

Output :

The new created array is : 1 2 3

The new created array is : 2.5 3.2 3.3

Some of the data types are mentioned below which will help in creating an array of different data types.

Type Code	C Type	Python Type	Minimum Size In Bytes
'b'	signed cahar	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Array operations

Some of the basic operations supported by an array are as follows:

- **Traverse** - It prints all the elements one by one.
- **Insertion** - It adds an element at the given index.
- **Deletion** - It deletes an element at the given index.
- **Search** - It searches an element using the given index or by the value.
- **Update** - It updates an element at the given index.

1.Traverse(Accessing array elements): We can access the array elements using the respective indices of those elements.

Example:

```
import array as arr
a = arr.array('i', [2, 4, 6, 8])
print("First element:", a[0])
print("Second element:", a[1])
print("Second last element:", a[-1])
```

Output:

```
First element: 2
Second element: 4
Second last element: 8
```

Explanation: In the above example, we have imported an array, defined a variable named as "a" that holds the elements of an array and print the elements by accessing elements through indices of an array.

2.Insertion(Adding Elements to a Array)

1.insert(): Insert is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

2.append(): It is also used to add the value mentioned in its arguments at the end of the array.

Example:

```
# Python program to demonstrate  Adding Elements to a Array


# importing "array" for array creations
import array as arr

# array with int type
a = arr.array('i', [1, 2, 3])

print ("Array before insertion : ", end =" ")
for i in range (0, 3):
   print (a[i], end =" ")
print()
```

```
# inserting array using
# insert() function
a.insert(1, 4)

print ("Array after insertion : ", end =" ")
for i in (a):
    print (i, end =" ")
print()

# array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

print ("Array before insertion : ", end =" ")
for i in range (0, 3):
    print (b[i], end =" ")
print()

# adding an element using append()
b.append(4.4)

print ("Array after insertion : ", end =" ")
for i in (b):
    print (i, end =" ")
print()
```

Output :

Array before insertion : 1 2 3

Array after insertion : 1 4 2 3

Array before insertion : 2.5 3.2 3.3

Array after insertion : 2.5 3.2 3.3 4.4

3.Deletion(Removing Elements from the Array)

Elements can be removed from the array by using

1. remove(): It removes one element at a time, to remove range of elements, iterator is used.

2. pop(): it is also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the pop() method.

Note – Remove method in List will only remove the first occurrence of the searched element.

Example:

Python program to demonstrate Removal of elements in a Array

```
# importing "array" for array operations
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 5])

# printing original array
print ("The new created array is : ", end ="")
for i in range (0, 5):
    print (arr[i], end =" ")

# using pop() to remove element at 2nd position
print ("The popped element is : ", end ="")
print (arr.pop(2))

# printing array after popping
print ("The array after popping is : ", end ="")
for i in range (0, 4):
    print (arr[i], end =" ")

# using remove() to remove 1st occurrence of 1
arr.remove(1)

# printing array after removing
print ("The array after removing is : ", end ="")
for i in range (0, 3):
    print (arr[i], end =" ")
```

Output:

The new created array is : 1 2 3 1 5

The popped element is : 3

The array after popping is : 1 2 1 5

The array after removing is : 2 1 5

4.Search(Searching element in a Array)

In order to search an element in the array we use a python

1. index():This function returns the index of the first occurrence of value mentioned in arguments.

Example:

Python code to demonstrate searching an element in array

```
# importing array module
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print ("The new created array is : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")


# using index() to print index of 1st occurrenece of 2
print ("The index of 1st occurrence of 2 is : ", end ="")
print (arr.index(2))

# using index() to print index of 1st occurrenece of 1
print ("The index of 1st occurrence of 1 is : ", end ="")
print (arr.index(1))
```

Output:

The new created array is : 1 2 3 1 2 5

The index of 1st occurrence of 2 is : 1

The index of 1st occurrence of 1 is : 0

5.Updating Elements in a Array

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

Example:

Python code to demonstrate how to update an element in array

```
# importing array module
import array

# initializing array with array values
# initializes array with signed integers
arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print ("Array before updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")

print ("\r")

# updating a element in a array
arr[2] = 6
print("Array after updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
print()

# updating a element in a array
arr[4] = 8
print("Array after updation : ", end ="")
for i in range (0, 6):
```

```
print (arr[i], end =" ")
```

Output:

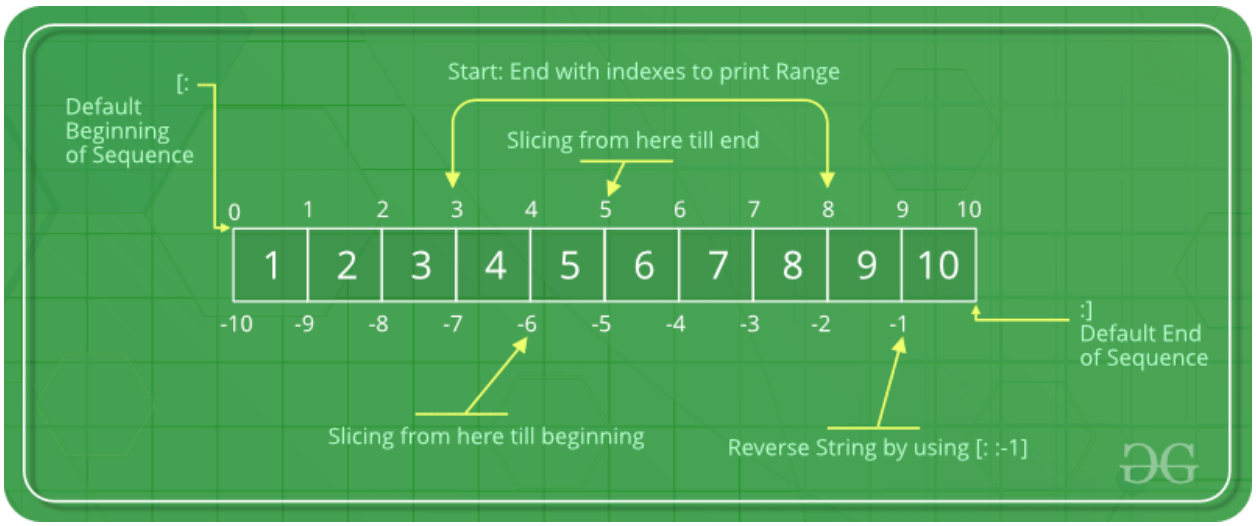
Array before updation : 1 2 3 1 2 5

Array after updation : 1 2 6 1 2 5

Array after updation : 1 2 6 1 8 5

Slicing of a Array

- In Python array, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use Slice operation.
- Slice operation is performed on array with the use of colon(:). To print elements from beginning to a range use [:Index], to print elements from end use [:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print whole List with the use of slicing operation, use [:].
- Further, to print whole array in reverse order, use [::-1].



Example:

```
# Python program to demonstrate slicing of elements in a Array

# importing array module
```

```
import array as arr

# creating a list
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

a = arr.array('i', l)
print("Initial Array: ")
for i in (a):
    print(i, end =" ")

# Print elements of a range
# using Slice operation
Sliced_array = a[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_array)

# Print elements from a
# pre-defined point to end
Sliced_array = a[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_array)

# Printing elements from
# beginning till end
Sliced_array = a[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_array)
```

Output

Initial Array:

1 2 3 4 5 6 7 8 9 10

Slicing elements in a range 3-8:

array('i', [4, 5, 6, 7, 8])

Elements sliced from 5th element till the end:

array('i', [6, 7, 8, 9, 10])

Printing all elements using slice operation:

array('i', [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

Types of Arrays

There are different types of Arrays:

1.One Dimensional Arrays(1D Arrays)

2.Two Dimensional Arrays(2D Arrays)

3. Three Dimensional Arrays(3D Arrays)

1.One Dimentional Array: The array which consists the values in single dimension is called “One Dimentional Array”.

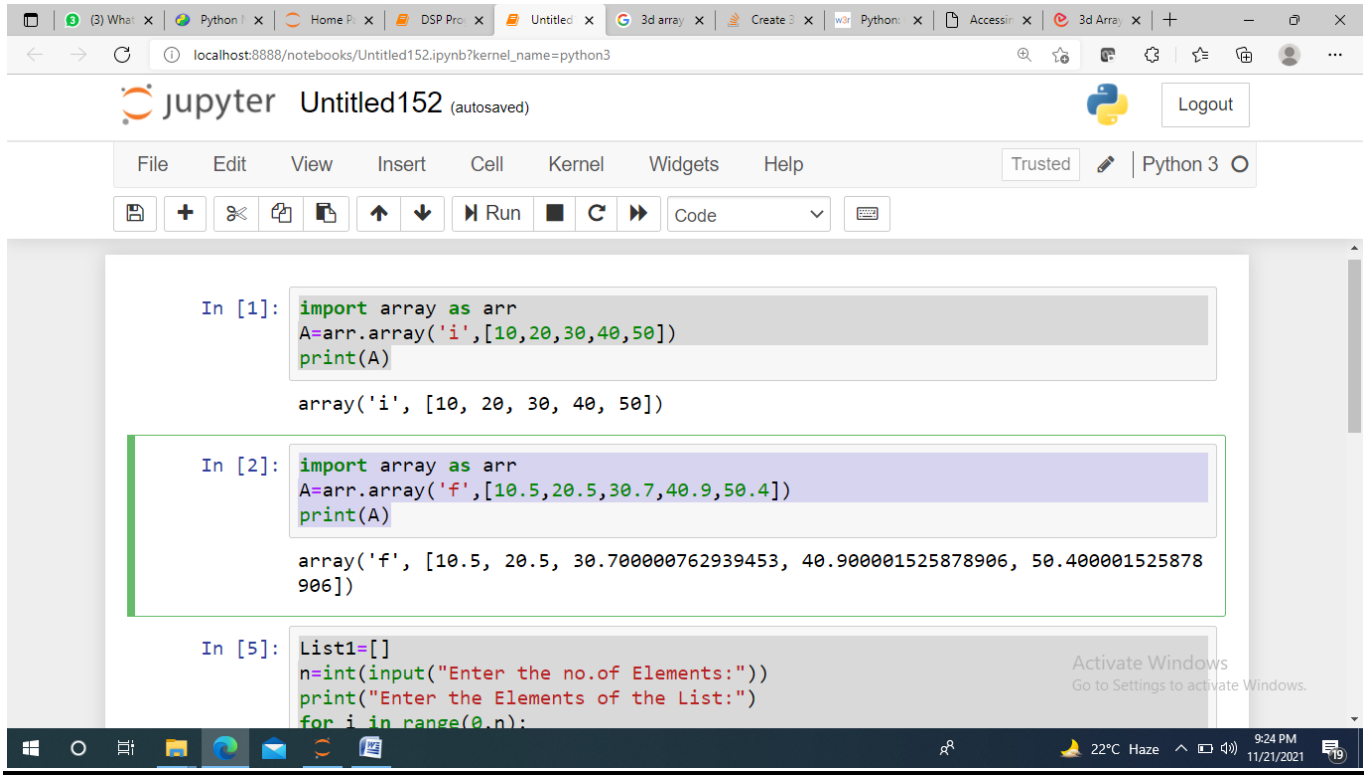
Syntax to create 1D Array:

```
Import array as arr
Array_name=arr.array([datatype,[List of values])
```

Ex:

```
#Creating Integer array
    import array as arr
    A=arr.array('i',[10,20,30,40,50])
    print(A)
#Creating Integer array
    import array as arr
    A=arr.array('f',[10.5,20.5,30.7,40.9,50.4])
    print(A)
```

Output:



2.Two-Dimensional Array (2-D Array):A 2D array **is an array of arrays that can be represented in matrix form like rows and columns.**

- In this array, the position of data elements is defined with two indices instead of a single index.

Syntax

Array_name = [rows][columns] # declaration of 2D array
Arr-name = [[m1, m2, m3, … . m_n], [n1, n2, n3, … .. n_n]]

Where **m** is the row and **n** is the column of the table.

Access Two-Dimensional Array

In Python we can access elements of a two-dimensional array using two indices. The first index refers to the indexing of the list and the second index refers to the position of the elements. If we define only one index with an array name, it returns all the elements of 2-dimensional stored in the array

2dSimple.py

Student_dt = [[72, 85, 87, 90, 69], [80, 87, 65, 89, 85], [96, 91, 70, 78, 97], [90, 93, 91, 90 , 94], [57, 89, 82, 69, 60]]

```
#print(student_dt[])
print(Student_dt[1]) # print all elements of index 1
print(Student_dt[0]) # print all elements of index 0
print(Student_dt[2]) # print all elements of index 2
print(Student_dt[3][4]) # it defines the 3rd index and 4 position of the data element.
```

Output:

```
[80, 87, 65, 89, 85]
[72, 85, 87, 90, 69]
[96, 91, 70, 78, 97]
94
```

Explanation: In the above example, we passed 1, 0, and 2 as parameters into 2D array that prints the entire row of the defined index. And we have also passed **student_dt[3][4]** that represents the 3rd index and 4th position of a 2-dimensional array of elements to print a particular element.

Traversing the element in 2D (two dimensional)

Program.py

```
# write a program to traverse every element of the two-
dimensional array in Python.
Student_dt = [ [72, 85, 87, 90, 69], [80, 87, 65, 89, 85], [96, 91, 70, 78, 97], [90, 93,
 91, 90, 94], [57, 89, 82, 69, 60] ]
# Use for loop to print the entire elements of the two dimensional array.
for x in Student_dt:  # outer loop
    for i in x:  # inner loop
        print(i, end = " ") # print the elements
    print()
```

Output:

```
Traverse each element in 2 Dimensional Array
72 85 87 90 69
80 87 65 89 85
96 91 70 78 97
90 93 91 90 94
57 89 82 69 60
```

Insert elements in a 2D (Two Dimensional) Array

- We can insert elements into a 2 D array using the **insert()** function that specifies the element' index number and location to be inserted.

Insert.py

```
# Write a program to insert the element into the 2D (two dimensional) array of Pyth
on.
from array import * # import all package related to the array.
arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]  # initialize the array elements.
print("Before inserting the array elements: ")
print(arr1) # print the arr1 elements.
# Use the insert() function to insert the element that contains two parameters.
arr1.insert(1, [5, 6, 7, 8])  # first parameter defines the index no., and second param
eter defines the elements
print("After inserting the array elements ")
for i in arr1: # Outer loop
    for j in i: # inner loop
        print(j, end = " ") # print inserted elements.
    print()
```

Output:

```
Before inserting the array elements:
[[1, 2, 3, 4], [8, 9, 10, 12]]
After inserting the array elements
1 2 3 4
5 6 7 8
8 9 10 12
```

Update elements in a 2 -D (Two Dimensional) Array

- In a 2D array, the existing value of the array can be updated with a new value. In this method, we can change the particular value as well as the entire index of the array. Let's understand with an example of a 2D array, as shown below.

Create a program to update the existing value of a 2D array in Python.

Update.py

```
from array import * # import all package related to the array.
arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]  # initialize the array elements.
print("Before inserting the array elements: ")
print(arr1) # print the arr1 elements.


arr1[0] = [2, 2, 3, 3] # update the value of the index 0
arr1[1][2] = 99 # define the index [1] and position [2] of the array element to update
 the value.
print("After inserting the array elements ")
for i in arr1: # Outer loop
    for j in i: # inner loop
        print(j, end = " ") # print inserted elements.
    print()
```

Output:

```
Before inserting the array elements:
[[1, 2, 3, 4], [8, 9, 10, 12]]
After inserting the array elements
2 2 3 3
8 9 99 12
```

Delete values from a 2D (two Dimensional) array in Python

- In a 2- D array, we can remove the particular element or entire index of the array using **del()** function in Python.

Delete.py

```
from array import * # import all package related to the array.
```

```
arr1 = [[1, 2, 3, 4], [8, 9, 10, 12]]  # initialize the array elements.
print("Before Deleting the array elements: ")
print(arr1) # print the arr1 elements.

del(arr1[0][2]) # delete the particular element of the array.
del(arr1[1]) # delete the index 1 of the 2-D array.

print("After Deleting the array elements ")
for i in arr1: # Outer loop
    for j in i: # inner loop
        print(j, end = " ") # print inserted elements.
    print()
```

Output:

```
Before Deleting the array elements:
[[1, 2, 3, 4], [8, 9, 10, 12]]
After Deleting the array elements
1 2 4
```

Size of a 2D array

- A **len**() function is used to get the length of a two-dimensional array. In other words, we can say that a **len**() function determines the total index available in 2-dimensional arrays.
- The len() function to get the size of a 2-dimensional array in Python.

Size.py

```
array_size = [[1, 3, 2],[2,5,7,9], [2,4,5,6]] # It has 3 index

print("The size of two dimensional array is : ")
print(len(array_size)) # it returns 3

array_def = [[1, 3, 2], [2, 4, 5, 6]] # It has 2 index
print("The size of two dimensional array is : ")
print(len(array_def)) # it returns 2
```

Output:

```
The size of two dimensional array is :
3
The size of two dimensional array is :
2
```

Write a program to print the sum of the 2-dimensional arrays in Python.

Matrix.py

```
def two_d_matrix(m, n): # define the function

    Outp = []  # initially output matrix is empty
    for i in range(m): # iterate to the end of rows
        row = []
        for j in range(n): # j iterate to the end of column
            num = int(input(f "Enter the matrix [{0}][{j}]"))
            row.append(num) # add the user element to the end of the row
        Outp.append(row) # append the row to the output matrix
    return Outp

def sum(A, B): # define sum() function to add the matrix.
    output = [] # initially, it is empty.
    print("Sum of the matrix is :")
    for i in range(len(A)): # no. of rows
        row = []
        for j in range(len(A[0])): # no. of columns

            row.append(A[i][j] + B[i][j]) # add matrix A and B
        output.append(row)
    return output    # return the sum of both matrix

m = int(input("Enter the value of m or Row\n")) # take the rows
n = int(input("Enter the value of n or columns\n")) # take the columns

print("Enter the First matrix ") # print the first matrix
A = two_d_matrix(m, n) # call the matrix function
print("display the first (A) matrix")
print(A) # print the matrix

print("Enter the Second (B) matrix ")
```

```
B = two_d_matrix(m, n) # call the matrix function
print("display the Second (B) matrix")
print(B) # print the B matrix

s= sum(A, B) # call the sum function
print(s) # print the sum of A and B matrix.
```

Output:

```
Enter the value of m or Row :3
Enter the value of n or columns :3
Enter the First matrix
Enter the matrix [0][0]2
Enter the matrix [0][1]4
Enter the matrix [0][2]1
Enter the matrix [0][0]2
Enter the matrix [0][1]5
Enter the matrix [0][2]6
Enter the matrix [0][0]3
Enter the matrix [0][1]4
Enter the matrix [0][2]0
display the first (A) matrix
[[2, 4, 1], [2, 5, 6], [3, 4, 0]]
Enter the Second (B) matrix
Enter the matrix [0][0]3
Enter the matrix [0][1]1
Enter the matrix [0][2]0
Enter the matrix [0][0]2
Enter the matrix [0][1]4
Enter the matrix [0][2]3
Enter the matrix [0][0]2
Enter the matrix [0][1]1
Enter the matrix [0][2]7
display the Second (B) matrix
[[3, 1, 0], [2, 4, 3], [2, 1, 7]]
Sum of the matrix is :
[[5, 5, 1], [4, 9, 9], [5, 5, 7]]
```

3.Three Dimensional Arrays:The array which consists multiple dimensions is called "3-D Array".

Example:

```
array = [[ ['*' for col in range(6)] for col in range(4)] for row in range(3)]

print(array)
```

Output:

```
[[['*', '*', '*', '*', '*', '*'], ['*', '*', '*', '*', '*', '*'], ['*', '*', '*', '*', '*', '*'], ['*', '*', '*', '*', '*', '*']],
[['*', '*', '*', '*', '*', '*'], ['*', '*', '*', '*', '*', '*'], ['*', '*', '*', '*', '*'
, '*'], ['*', '*', '*', '*', '*', '*']], [['*', '*', '*', '*', '*', '*'], ['*', '*', '*', '*', '*', '*'], ['*', '*', '*', '*', '*',
'*'], ['*', '*', '*', '*', '*', '*']]]
```

List Vs Array

List	Array
Can consist of elements belonging to different data types	Only consists of elements belonging to the same data type
No need to explicitly import a module for declaration	Need to explicitly import a module for declaration
Cannot directly handle arithmetic operations	Can directly handle arithmetic operations
Can be nested to contain different type of elements	Must contain either all nested elements of same size
Preferred for shorter sequence of data items	Preferred for longer sequence of data items
Greater flexibility allows easy modification (addition, deletion) of data	Less flexibility since addition, deletion has to be done element wise
The entire list can be printed without any explicit looping	A loop has to be formed to print or access the components of array
Consume larger memory for easy addition of elements	Comparatively more compact in memory size

Extra Content about Types of Arrays

Python: Convert a 1D array to a 2D Numpy array or Matrix

- We will discuss how to convert a 1D Numpy Array to a 2D numpy array or Matrix using reshape() function.

- We will also discuss how to construct the 2D array row wise and column wise, from a 1D array.

- Suppose we have a 1D numpy array of size 10,

```
# create 1D numpy array from a list
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print('1D Numpy array:')
print(arr)
```

Output:

```
[0 1 2 3 4 5 6 7 8 9]
```

Reshape 1D array to 2D array or Matrix

First, import the numpy module,

```
import numpy as np
```

- Now to convert the shape of numpy array, we can use the reshape() function of the numpy module,

numpy.reshape()

numpy.reshape(arr, newshape, order='C')

Accepts following arguments,

- a: Array to be reshaped, it can be a numpy array of any shape or a list or list of lists.
- newshape: New shape either be a tuple or an int.
- order: The order in which items from the input array will be used.

It returns a new view object (if possible, otherwise returns a copy) of new shape.

Let's use this to convert our 1D numpy array to 2D numpy array,

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# Convert 1D array to a 2D numpy array of 2 rows and 3 columns
arr_2d = np.reshape(arr, (2, 5))

print(arr_2d)
```

Output:

```
[[0 1 2 3 4]
[5 6 7 8 9]]
```

- We passed the 1D array as the first argument and the new shape i.e. a tuple (2, 5) as the second argument. It returned a 2D view of the passed array.
- An important point here is that the new shape of the array must be compatible with the original shape of the input array, otherwise it will raise the ValueError. For example, if we try to reshape out 1D numpy array of 10 elements to a 2D array of size 2X3, then it will raise error,

```
# Converting 1D array to a 2D numpy array of incompatible shape will cause error
arr_2d = np.reshape(arr, (2, 3))
```

Error:

ValueError: cannot reshape array of size 10 into shape (2,3)

It raised the error because 1D array of size 10 can only be reshaped to a 2D array of size 2X5 or 5X2. But in the above example, we tried to convert it into a shape which is incompatible with its size.

Reshaped 2D array is a view of 1D array

If possible then reshape() function returns a view of the original array and any modification in the view object will affect the original input array too. For example,

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
arr_2d = np.reshape(arr, (2, 5))
# Modify the 2D numpy array (View object)
arr_2d[0][0] = 22
print('1D Numpy array:')
print(arr)
print('2D Numpy array:')
print(arr_2d)
```

Output:

```
1D Numpy array:
[22 1 2 3 4 5 6 7 8 9]
2D Numpy array:
```

```
[[22 1 2 3 4]
[ 5 6 7 8 9]]
```

Convert a 1D Numpy array to a 3D Numpy array using numpy.reshape()

Suppose we have a 1D numpy array of 12 elements,

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

Now let's convert this 1D array to a 3D numpy array i.e.

```
# Convert 1D Numpy array to a 3D array with 2 matrices of shape 2X3
arr_3d = np.reshape(arr, (2, 2, 3))
print('3D Numpy array:')
print(arr_3d)
```

Output:

```
3D Numpy array:
[[[ 1 2 3]
[ 4 5 6]]
[[ 7 8 9]
[10 11 12]]]
```

We passed the 1D array as the first argument and the new shape i.e. a tuple (2, 2, 3) as the second argument. It returned a 3D view of the passed array.

Convert 1D Numpy array to a 2D numpy array along the column

In the previous example, when we converted a 1D array to a 2D array or matrix, then the items from input array will be read row wise i.e.

- 1st row of 2D array was created from items at index 0 to 2 in input array
- 2nd row of 2D array was created from items at index 3 to 5 in input array
- 3rd row of 2D array was created from items at index 6 to 8 in input array

Now suppose we want to construct the matrix / 2d array column wise. For that we can pass the order parameter as 'F' in the reshape() function i.e.

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# column wise conversion of 1D numpy array to 2D Numpy array
arr_2d = np.reshape(arr, (2, 5), order='F')
print('2D Numpy array:')
print(arr_2d)
```

Output:

```
2D Numpy array:
[[0 2 4 6 8]
[1 3 5 7 9]]
```

It converted the 1D array to a 2D matrix and this matrix was created column wise i.e.

- 1st column of 2D array was created from items at index 0 to 2 in input array
- 2nd column of 2D array was created from items at index 3 to 5 in input array

- 3rd column of 2D array was created from items at index 6 to 8 in input array

Convert 2D Array to 1D Array as copy

If possible then numpy.reshape() returns a view of the original array. Now suppose we want to create a 2D copy of the 1D numpy array then use the copy() function along with the reshape() function,

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

arr_2d = np.reshape(arr, (2, 5)).copy()

# Modify the 2D numpy array and it will not affect original 1D array
arr_2d[0][0] = 22

print('1D Numpy array:')

print(arr)

print('2D Numpy array:')
print(arr_2d)
```

Output:

```
1D Numpy array:
[0 1 2 3 4 5 6 7 8 9]
2D Numpy array:
[[22 1 2 3 4]
[ 5 6 7 8 9]]
```

It created a 2D copy of the 1D array. Any changes made in this 2D array will not affect the original array.

The complete example is as follows,

```
import numpy as np

def main():

print('*** Convert a 1D array to a 2D Numpy array ***')

# create 1D numpy array from a list

arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

print('1D Numpy array:')

print(arr)

# Convert 1D array to a 2D numpy array of 2 rows and 3 columns

arr_2d = np.reshape(arr, (2, 5))

print('2D Numpy array:')

print(arr_2d)

print('Shape of 2D array must be compatible to 1D array')

# Converting 1D array to a 2D numpy array of incompatible shape will cause error

#arr_2d = np.reshape(arr, (2, 3))

#ValueError: cannot reshape array of size 10 into shape (2,3)

print('Reshaped 2D array is a view of 1D array')

# Modify the 2D numpy array (View object)
```

```
arr_2d[0][0] = 22
print('1D Numpy array:')
print(arr)
print('2D Numpy array:')
print(arr_2d)
print('Convert a 1D Numpy array to a 3D Numpy array using numpy.reshape()')
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
print('1D Numpy array:')
print(arr)
# Convert 1D Numpy array to a 3D array with 2 matrices of shape 2X3
arr_3d = np.reshape(arr, (2, 2, 3))
print('3D Numpy array:')
print(arr_3d)
print('*** Convert 1D Numpy array to 2D numpy array along the column ***')
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print('1D Numpy array:')
print(arr)
# column wise conversion of 1D numpy array to 2D Numpy array
arr_2d = np.reshape(arr, (2, 5), order='F')
print('2D Numpy array:')
print(arr_2d)
```

```
print('*** Convert 2D aray to 1D array as copy ***')
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print('1D Numpy array:')
print(arr)
arr_2d = np.reshape(arr, (2, 5)).copy()
print('2D Numpy array:')
print(arr_2d)
# Modify the 2D numpy array and it will not affect original 1D array
arr_2d[0][0] = 22
print('1D Numpy array:')
print(arr)
print('2D Numpy array:')
print(arr_2d)
if __name__ == '__main__':
main()
```

Output

```
*** Convert a 1D array to a 2D Numpy array ***
1D Numpy array:
[0 1 2 3 4 5 6 7 8 9]
2D Numpy array:
[[0 1 2 3 4]
```

[5 6 7 8 9]]

Shape of 2D array must be compatible to 1D array

Reshaped 2D array **is** a view of 1D array

1D Numpy array:

[22 1 2 3 4 5 6 7 8 9]

2D Numpy array:

[[22 1 2 3 4]

[5 6 7 8 9]]

Convert a 1D Numpy array to a 3D Numpy array using numpy.reshape()

1D Numpy array:

[1 2 3 4 5 6 7 8 9 10 11 12]

3D Numpy array:

[[[1 2 3]

[4 5 6]]

[[7 8 9]

[10 11 12]]]

*** Convert 1D Numpy array to 2D numpy array along the column ***

1D Numpy array:

[0 1 2 3 4 5 6 7 8 9]

2D Numpy array:

[[0 2 4 6 8]

[1 3 5 7 9]]

*** Convert 2D aray to 1D array **as** copy ***

1D Numpy array:

[0 1 2 3 4 5 6 7 8 9]

2D Numpy array:

[[0 1 2 3 4]

[5 6 7 8 9]]

1D Numpy array:

[0 1 2 3 4 5 6 7 8 9]

2D Numpy array:

[[22 1 2 3 4]

[5 6 7 8 9]]

UNIT 4

Linked Lists, Stacks, Queues

UNIT IV

Linked Lists – Implementation ofSingly 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

Introduction:

Linked List:

- Linked lists are one of the most commonly used data structures in any programming language.
- Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.
- Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.

Linked Lists vs Arrays:

- A linked list is a dynamic data structure which means that the memory reserved for the link list can be increased or reduced at runtime.
    - No memory is allocated for a linked list data structure in advance.
    - Whenever a new item is required to be added to the linked, the memory for the new node is created at run time.
    - On the other hand, in case of the array, memory has to be allocated in advance for a specific number of items.
    - In cases where sufficient items are not available to fill all array index, memory space is wasted.

- Since arrays require contiguous memory locations, it is very difficult to remove or insert an item in an array since the memory locations of a large number of items have to be updated.

    1. On the other hand, linked list items are not stored in a contiguous memory location, therefore you can easily update linked lists.
    2. Owing to its flexibility, a linked list is more suitable for implementing data structures like stacks, queues, and lists.

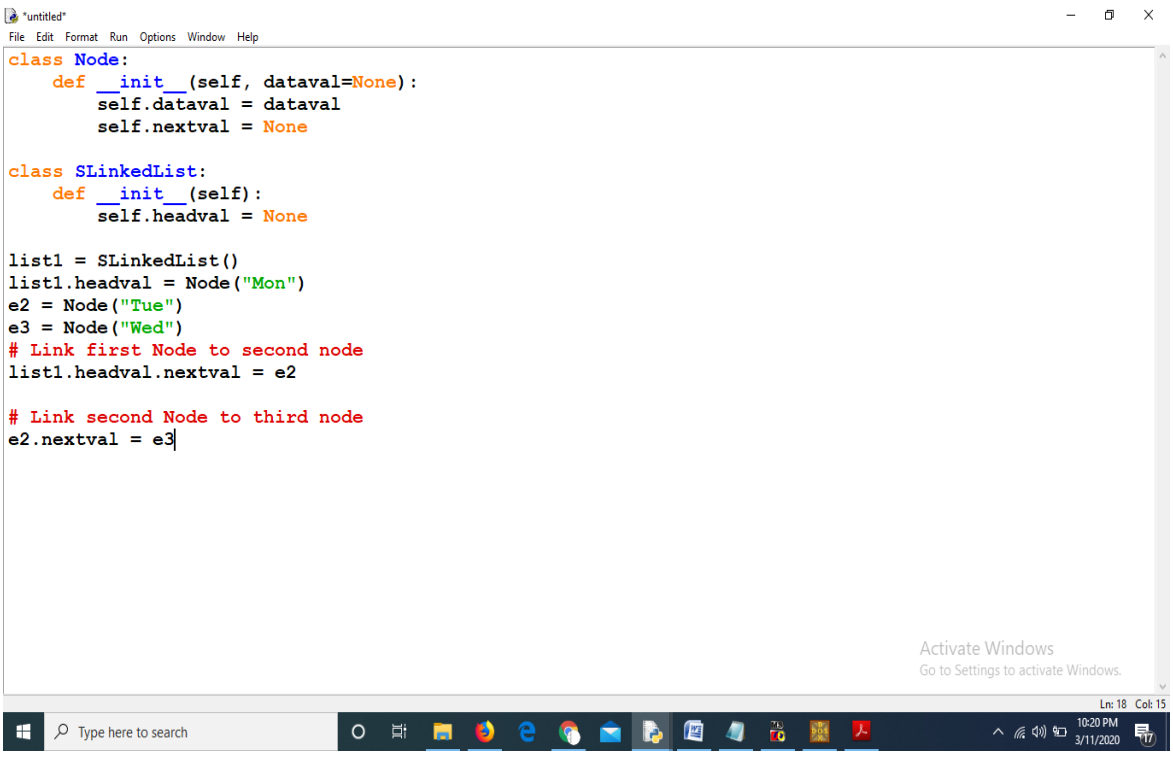
    There are different types of Linked lists.They are:

    - Single Linked List
    - Doubly Linked List
    - Circular Linked List

Creation of Linked list

A linked list is created by using the node class . We create a Node object and create another class to use this code object. We pass the appropriate values thorugh the node object to point the to the next data elements.

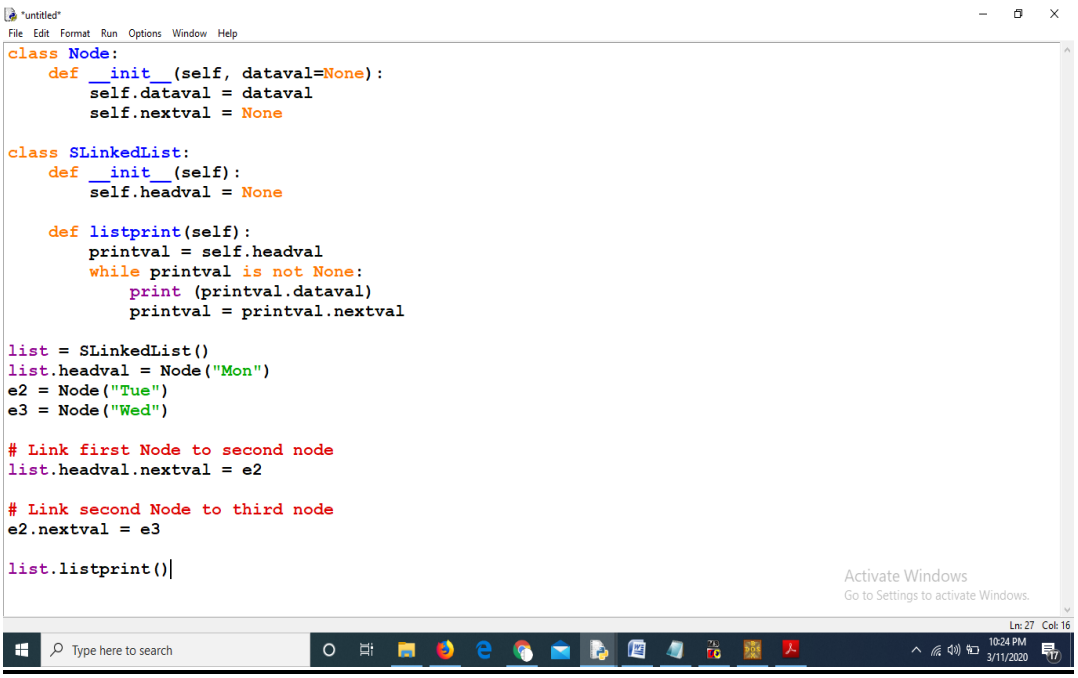
Code to create a node:



Traversing a Linked List

Singly linked lists can be traversed in only forwrad direction starting form the first data element. We simply print the value of the next data element by assgining the pointer of the next node to the current data element.

Code to traverse in a LinkedList:



Single Linked List

What is Linked List?

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".

What is Single Linked List?

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

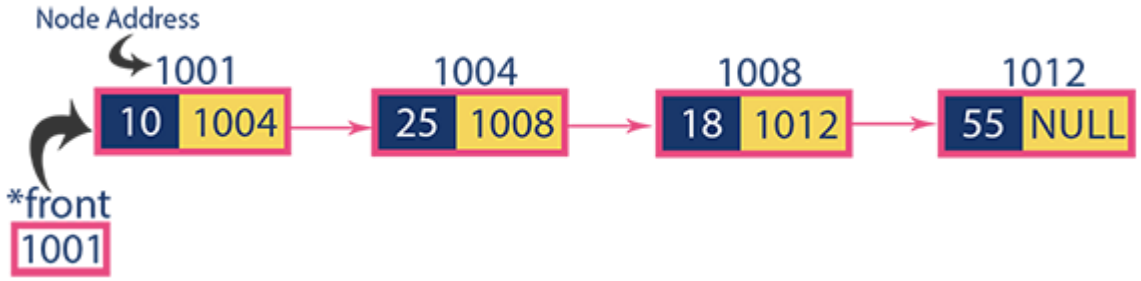
Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.
The graphical representation of a node in a single linked list is as follows...

Importent Points to be Remembered

- In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").
- Always next part (reference part) of the last node must be NULL.

Example



Operations on Single Linked List

The following operations are performed on a Single Linked List

- **Insertion**
- **Deletion**
- **Display**

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program.

Step 2 - Declare all the **user defined functions**.

Step 3 - Define a **Node** structure with two members **data** and **next**

Step 4 - Define a Node pointer **'head'** and set it to **NULL**.

Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then,
set **newNode→next** = **NULL** and **head** = **newNode**.

Step 4 - If it is **Not Empty** then,
set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1 - Create a **newNode** with given value and **newNode → next** as **NULL**.

Step 2 - Check whether list is **Empty** (**head** == **NULL**).

Step 3 - If it is **Empty** then, set **head** = **newNode**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).

Step 6 - Set **temp → next** = **newNode**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.

Step 7 - Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Check whether list is having only one node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.

Step 4 - Check whether list has only one Node (**temp1 → next** == **NULL**)

Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)

Step 7 - Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8 - If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).

Step 9 - If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.

Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).

Step 11 - If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).

Step 12 - If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display '**List is Empty!!!**' and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **head**.

Step 4 - Keep displaying **temp → data** with an arrow (--->) until **temp** reaches to the last node

Step 5 - Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Implementation of Single Linked List using Python Programming

Code:

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
class SLinkedList:
    def __init__(self):
        self.head = None
    def Insert_At_Begin(self):
        Newnode=Node()
        value=int(input("Enter the value to be inserted:"))
        Newnode.data=value
        if self.head is None:
            self.head=Newnode
            Newnode.next=None
```

```
    else:
        Newnode.next=self.head
        self.head=Newnode
    print('Inserted Successfully')


def Insert_At_Last(self):
    Newnode=Node()
    value=int(input("Enter the value to be inserted:"))
    Newnode.data=value
    if self.head is None:
        self.head=Newnode
        Newnode.next=None
    else:
        temp=self.head
        while temp.next is not None:
            temp=temp.next
        temp.next=Newnode
        Newnode.next=None
        print('Inserted Successfully')
def Insert_At_Middle(self):
    Newnode=Node()
    value=int(input("Enter the value to be inserted:"))
    Newnode.data=value
    Pos=int(input("Enter the value after which u want to be inserted:"))
```

```
    if self.head is None:
        self.head=Newnode
        Newnode.next=None
    else:
        temp1=self.head
        temp2=None
        while temp1.data!=Pos:
            temp1=temp1.next
        temp2=temp1.next
        temp1.next=Newnode
        Newnode.next=temp2
        print('Inserted Successfully')

def Remove_At_First(self):
    if self.head is None:
        print('List is Empty')
    else:
        temp=self.head
        self.head=temp.next
    print('Element is removed successfully')

def Remove_At_Last(self):
    if self.head is None:
        print('List is Empty')
```

```
    else:
        temp1=self.head
        while temp1.next is not None:
            temp2=temp1
            temp1=temp1.next
        temp2.next=None
    print('Element is removed successfully')

def Remove_At_Middle(self):
    if self.head is None:
        print('List is Empty')
    else:
        value=int(input('Enter the element to be deleted:'))
        temp1=self.head
        while temp1.data!=value :
            temp2=temp1
            temp1=temp1.next
        temp2.next = temp1.next;
    print('Element is removed successfully')

def Search_List(self):
     Searchvalue=int(input('Enter the element to be Searched:'))
     temp1=self.head
     flag=0
```

```
        while(temp1 is not None):
            if(temp1.data==Searchvalue):
                flag=1
            temp1=temp1.next
        if(flag==1):
            print('Element is found')
        else:
            print('Element is not found')


    def Display(self):
        tempnode = self.head
        while tempnode.next!=None:
          # if tempnode is None:
           #    break
            print(tempnode.data,end=" ")
            tempnode=tempnode.next
        print(tempnode.data,end=" ")


List=SLinkedList()
print("**********LikedList Operations**********")
while(1):
    print("'\n\t\t1.Insert_At_Begin()
            2.Insert_At_Last()
            3.Insert_At_Middle()
```

```
        4.Remove_At_First()
        5.Remove_At_Last()
        6.Remove_At_Middle()
        7.Seach()
        8.DiaplayList() "')
choice=int(input('Enter the choice :'))
if(choice==1):
   List.Insert_At_Begin()
elif(choice==2):
   List.Insert_At_Last()
elif(choice==3):
   List.Insert_At_Middle()
elif(choice==4):
   List.Remove_At_First()
elif(choice==5):
   List.Remove_At_Last()
elif(choice==6):
   List.Remove_At_Middle()
elif(choice==7):
   List.Search_List()
elif (choice==8):
   List.Display()
```

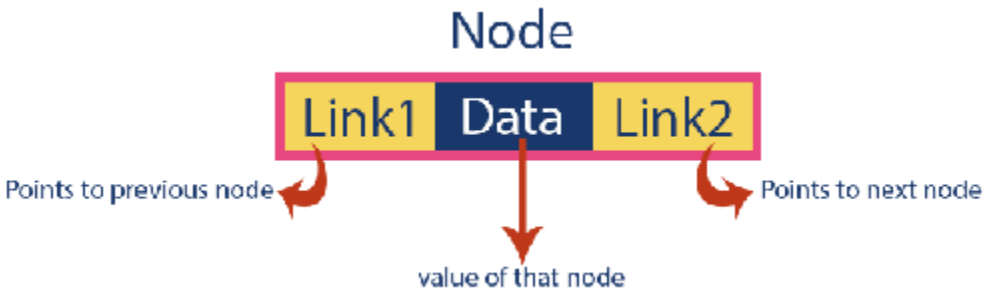
Double Linked List

What is Double Linked List?

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

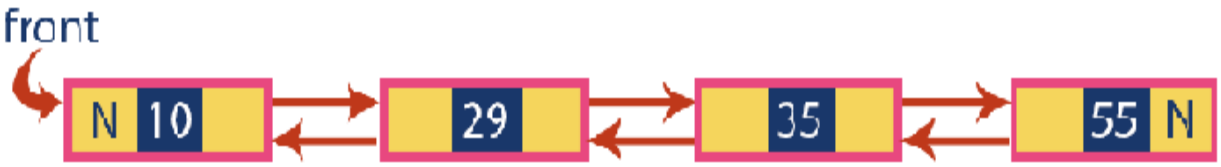
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example



Importent Points to be Remembered

In double linked list, the first node must be always pointed by head.

Always the previous field of the first node must be NULL.

Always the next field of the last node must be NULL.

Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → previous** is equal to **temp → next**)

- **Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- **Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- **Step 7 -** Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- **Step 5 -** If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).

- **Step 9 -** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- **Step 10 -** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- **Step 11 -** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- **Step 12 -** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty**, then display '**List is Empty!!!**' and terminate the function.

Step 3 - If it is not Empty, then define a Node pointer '**temp**' and initialize with **head**.

Step 4 - Display '**NULL <--- **'.

Step 5 - Keep displaying **temp → data** with an arrow (<===>) until **temp** reaches to the last node

Step 6 - Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Implementation of Double Linked List using Python Programming

Code:

```
class Node:

  # Constructor to create a new node

  def __init__(self):

     self.data = None

     self.next = None
```

```
        self.prev = None

# Class to create a Doubly Linked List
class DLinkedList:

    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    def Insert_At_Begin(self):
        Newnode=Node()
        value=int(input("Enter the value to be inserted:"))
        Newnode.data=value
        if self.head is None:
            self.head=Newnode
            Newnode.next=None
            Newnode.prev=None
        else:
            Newnode.next=self.head
            self.head=Newnode
            Newnode.prev=None
        print('Inserted Successfully')

    def Insert_At_Last(self):
```

```
    Newnode=Node()
    value=int(input("Enter the value to be inserted:"))
    Newnode.data=value
    if self.head is None:
        self.head=Newnode
        Newnode.next=None
        Newnode.prev=None
    else:
        temp=self.head
        while temp.next is not None:
            temp=temp.next
        temp.next=Newnode
        Newnode.prev=temp
        Newnode.next=None
        print('Inserted Successfully')
def Insert_At_Middle(self):
    Newnode=Node()
    value=int(input("Enter the value to be inserted:"))
    Newnode.data=value
    Pos=int(input("Enter the value after which u want to be inserted:"))
    if self.head is None:
        self.head=Newnode
        Newnode.next=None
        Newnode.prev=None
```

```
    else:
        temp1=self.head
        temp2=None
        while temp1.data!=Pos:
            temp1=temp1.next
        temp2=temp1.next
        temp1.next=Newnode
        Newnode.next=temp2
        Newnode.prev=temp1
        temp2.prev=Newnode
        print('Inserted Successfully')

def Remove_At_First(self):
    if self.head is None:
        print('List is Empty')
    else:
        temp=self.head
        self.head=temp.next
        temp=temp.next
        temp.prev=None
    print('Element is removed successfully')

def Remove_At_Last(self):
    if self.head is None:
```

```
        print('List is Empty')
    else:
        temp1=self.head
        while temp1.next is not None:
            temp2=temp1
            temp1=temp1.next
        temp2.next=None
    print('Element is removed successfully')

def Remove_At_Middle(self):
    if self.head is None:
        print('List is Empty')
    else:
        value=int(input('Enter the element to be deleted:'))
        temp1=self.head
        temp2=None
        temp3=None
        while temp1.data!=value :
            temp2=temp1
            temp1=temp1.next
        temp3=temp1.next
        temp2.next = temp1.next;
        temp3.prev=temp2.next
    print('Element is removed successfully')
```

```
    def Search_List(self):
        Searchvalue=int(input('Enter the element to be Searched:'))
        temp1=self.head
        flag=0
        while(temp1 is not None):
            if(temp1.data==Searchvalue):
                flag=1
            temp1=temp1.next
        if(flag==1):
            print('Element is found')
        else:
            print('Element is not found')

    def Display(self):
        tempnode = self.head
        while tempnode.next!=None:
            # if tempnode is None:
            #    break
            print(tempnode.data,end=" ")
            tempnode=tempnode.next
        print(tempnode.data,end=" ")

List=DLinkedList()
```

```
print("**********Doubly LikedList Operations**********")
while(1):
    print('''\n\t\t1.Insert_At_Begin()
            2.Insert_At_Last()
            3.Insert_At_Middle()
            4.Remove_At_First()
            5.Remove_At_Last()
            6.Remove_At_Middle()
            7.Seach()
            8.DiaplayList() ''')
    choice=int(input('Enter the choice :'))
    if(choice==1):
        List.Insert_At_Begin()
    elif(choice==2):
        List.Insert_At_Last()
    elif(choice==3):
        List.Insert_At_Middle()
    elif(choice==4):
        List.Remove_At_First()
    elif(choice==5):
        List.Remove_At_Last()
    elif(choice==6):
        List.Remove_At_Middle()
    elif(choice==7):
```

```
            List.Search_List()
        elif (choice==8):
            List.Display()
```

Circular Linked List

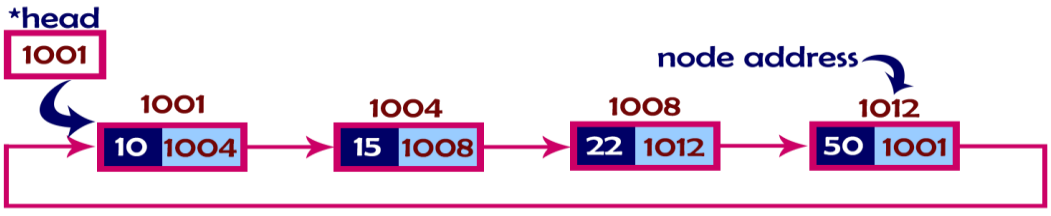
What is Circular Linked List?

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

Example



Operations

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion

3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined** functions.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 5 -** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').
- **Step 6 -** Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.

- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).
- **Step 6 -** Set **temp → next** = **newNode** and **newNode → next** = **head**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers '**temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- **Step 4 -** Check whether list is having only one node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head**)
- **Step 7 -** Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers '**temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)
- **Step 7 -** Set **temp2 → next** = **head** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers '**temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node

Step 5 - Finally display **temp → data** with arrow pointing to **head → data**.

Implementation of Circular Linked List using Python Programming

```
class Node:
# Constructor to create a new node
def __init__(self):
self.data = None
```

```
        self.next = None

# Class to create a Doubly Linked List
class CLinkedList:

    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    def Insert_At_Begin(self):
        Newnode=Node()
        value=int(input("Enter the value to be inserted:"))
        Newnode.data=value
        if self.head is None:
            self.head=Newnode
            Newnode.next=self.head
        else:
            temp=self.head
            while(temp.next!=self.head):
                temp=temp.next
            Newnode.next=self.head
            self.head=Newnode
            temp.next=self.head
        print('Inserted Successfully')
```

```
def Insert_At_Last(self):
    Newnode=Node()
    value=int(input("Enter the value to be inserted:"))
    Newnode.data=value
    if self.head is None:
        self.head=Newnode
        Newnode.next=self.head
    else:
        temp=self.head
        while(temp.next !=self.head):
            temp=temp.next
        temp.next=Newnode
        Newnode.next=self.head
    print('Inserted Successfully')
def Insert_At_Middle(self):
    Newnode=Node()
    value=int(input("Enter the value to be inserted:"))
    Newnode.data=value
    Pos=int(input("Enter the value after which u want to be inserted:"))
    if self.head is None:
        self.head=Newnode
        Newnode.next=self.head
    else:
```

```
        temp1=self.head
        temp2=None
        while temp1.data!=Pos:
            temp1=temp1.next
        Newnode.next=temp1.next
        temp1.next=Newnode
        print('Inserted Successfully')

def Remove_At_First(self):
    if self.head is None:
        print('List is Empty')
    else:
        temp1=self.head
        if temp1.next==self.head:
            self.head=None
        else:
            temp2=temp1
            while temp1.next!=self.head:
                temp1=temp1.next
            self.head=temp2.next
            temp1.next=self.head

    print('Element is removed successfully')
```

```
def Remove_At_Last(self):
    if self.head is None:
        print('List is Empty')
    else:
        temp=self.head
        if temp.next==self.head:
            self.head=None
        else:
            temp1=self.head
            while temp1.next!=self.head:
                temp2=temp1
                temp1=temp1.next
            temp2.next=self.head
    print('Element is removed successfully')

def Remove_At_Middle(self):
    if self.head is None:
        print('List is Empty')
    else:
        temp=self.head
        if temp.next==self.head:
            self.head=None
        else:
            value=int(input('Enter the element to be deleted:'))
```

```
            temp1=self.head
            temp2=None
            while temp1.data!=value :
               temp2=temp1
               temp1=temp1.next
            temp2.next = temp1.next;

      print('Element is removed successfully')

def Search_List(self):
     Searchvalue=int(input('Enter the element to be Searched:'))
     temp1=self.head
     flag=0
     while(temp1.next!=self.head):
        if(temp1.data==Searchvalue):
           flag=1
        temp1=temp1.next
     if(flag==1):
        print('Element is found')
     else:
        print('Element is not found')

def Display(self):
    tempnode = self.head
```

```
        while tempnode.next!=self.head:
            print(tempnode.data,end=" ")
            tempnode=tempnode.next
        print(tempnode.data,end=" ")


List=CLinkedList()
print("**********Circular LikedList Operations**********")
while(1):
    print('''\n\t\t1.Insert_At_Begin()
             2.Insert_At_Last()
             3.Insert_At_Middle()
             4.Remove_At_First()
             5.Remove_At_Last()
             6.Remove_At_Middle()
             7.Seach()
             8.DiaplayList() ''')
    choice=int(input('Enter the choice :'))
    if(choice==1):
        List.Insert_At_Begin()
    elif(choice==2):
        List.Insert_At_Last()
    elif(choice==3):
        List.Insert_At_Middle()
    elif(choice==4):
```

```
    List.Remove_At_First()
elif(choice==5):
    List.Remove_At_Last()
elif(choice==6):
    List.Remove_At_Middle()
elif(choice==7):
    List.Search_List()
elif (choice==8):
    List.Display()
```

Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Calls

1. Evaluation of Arithmetic Expressions

A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.

In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Example: A + (B - C)

To evaluate the expressions, one needs to be aware of the standard precedence rules for arithmetic expression. The precedence rules for the five basic arithmetic operators are:

Operators	Associativity	Precedence
^ exponentiation	Right to left	Highest followed by *Multiplication and /division
*Multiplication, /division	Left to right	Highest followed by + addition and - subtraction
+ addition, - subtraction	Left to right	lowest

Evaluation of Arithmetic Expression requires two steps:

- First, convert the given expression into special notation.
- Evaluate the expression in this new notation.

Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- Infix Notation

- Prefix Notation
- Postfix Notation

Infix Notation

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or unparenthesized depending upon the problem requirement.

Example: A + B, (C - D) etc.

All these expressions are in infix notation because the operator comes between the operands.

Prefix Notation

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.

Example: + A B, -CD etc.

All these expressions are in prefix notation because the operator comes before the operands.

Postfix Notation

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

Example: AB +, CD+, etc.

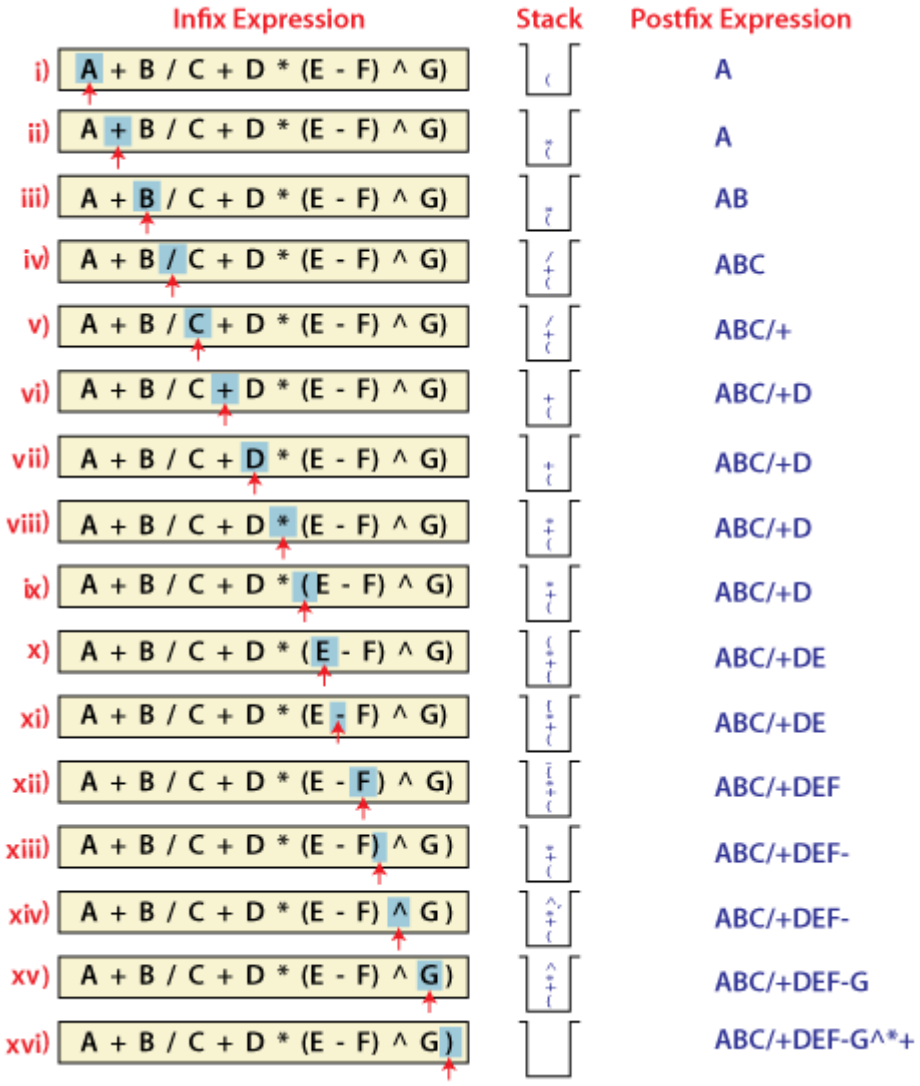
All these expressions are in postfix notation because the operator comes after the operands.

Conversion of Arithmetic Expression into various Notations:

Infix Notation	Prefix Notation	Postfix Notation

A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

Let's take the example of Converting an infix expression into a postfix expression.



In the above example, the only change from the postfix expression is that the operator is placed before the operands rather than between the operands.

Evaluating Postfix expression:

Stack is the ideal data structure to evaluate the postfix expression because the top element is always the most recent operand. The next element on the Stack is the second most recent operand to be operated on.

Before evaluating the postfix expression, the following conditions must be checked. If any one of the conditions fails, the postfix expression is invalid.

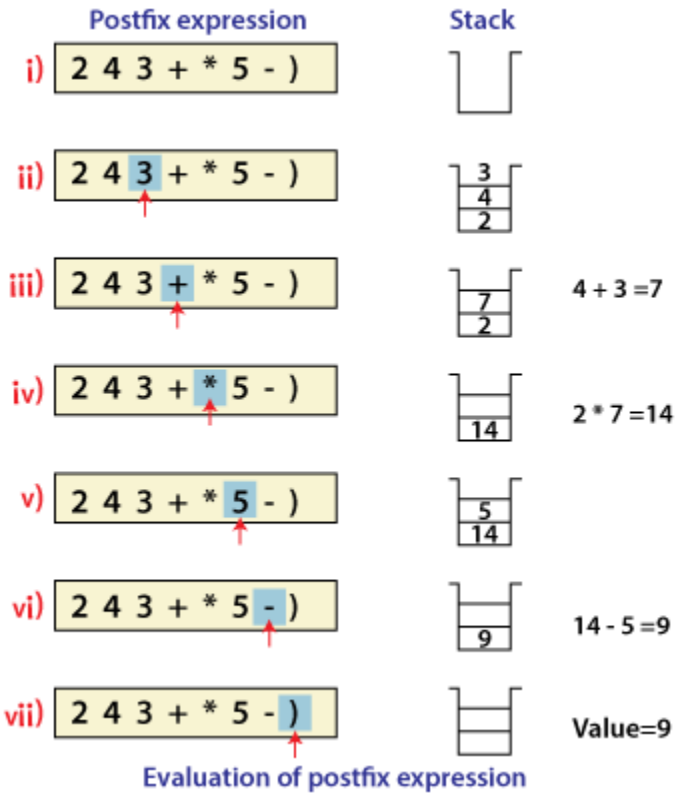
- When an operator encounters the scanning process, the Stack must contain a pair of operands or intermediate results previously calculated.
- When an expression has been completely evaluated, the Stack must contain exactly one value.

Example:

Now let us consider the following infix expression 2 * (4+3) - 5.

Its equivalent postfix expression is 2 4 3 + * 5.

The following step illustrates how this postfix expression is evaluated.



2. Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem.

3. Delimiter Checking

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters. We make use of different types of delimiters include the parenthesis checking (,), curly braces {,} and square brackets [,], and common delimiters /* and */. Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis. Also, the delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier.

Valid Delimiter	**Invalid Delimiter**
While (i > 0)	While (i >
/* Data Structure */	/* Data Structure
{ (a + b) - c }	{ (a + b) - c

To perform a delimiter checking, the compiler makes use of a stack. When a compiler translates a source program, it reads the characters one at a time, and if it finds an opening delimiter it places it on a stack. When a closing delimiter is found, it pops up the opening delimiter from the top of the Stack and matches it with the closing delimiter.

On matching, the following cases may arise.

- If the delimiters are of the same type, then the match is considered successful, and the process continues.
- If the delimiters are not of the same type, then the syntax error is reported.

When the end of the program is reached, and the Stack is empty, then the processing of the source program stops.

Example: To explain this concept, let's consider the following expression.

[{a -b) * (c -d)}/f]

Input left	Characters Read	Stack Contents
{{{ a-b) * (c-d)}/f]	[	[
{{ a-b) * (c-d)}/f]	{	[{
(a-b) * (c-d)}/f]	(	[{ (
a-b) * (c-d)}/f]	a	[{ (
-b) * (c-d)}/f]	-	[{ (
b) * (c-d)}/f]	b	[{ (
) * (c-d)}/f]	)	[{
* (c-d)}/f]	*	[{
(c-d)}/f]	(	[{ (
c-d)}/f]	g	[{ (
-d)}/f]	-	[{ (
d)}/f]	d	[{ (
)}/f]	)	[{
}/f]	}	[
/f]	/	[
f]	f	[
]		

4. Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

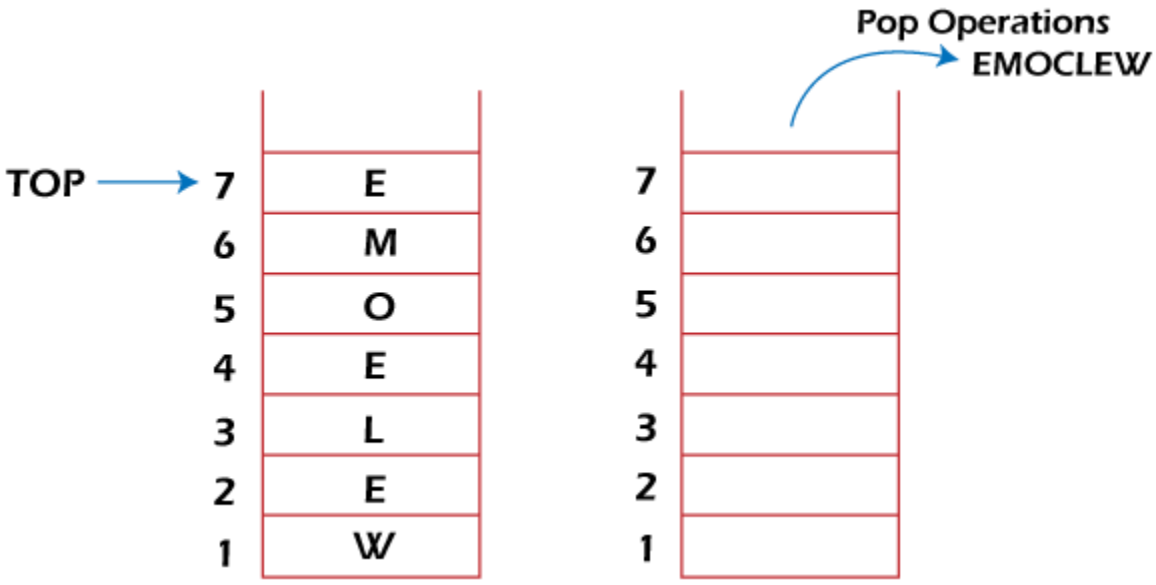
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

Reverse a String

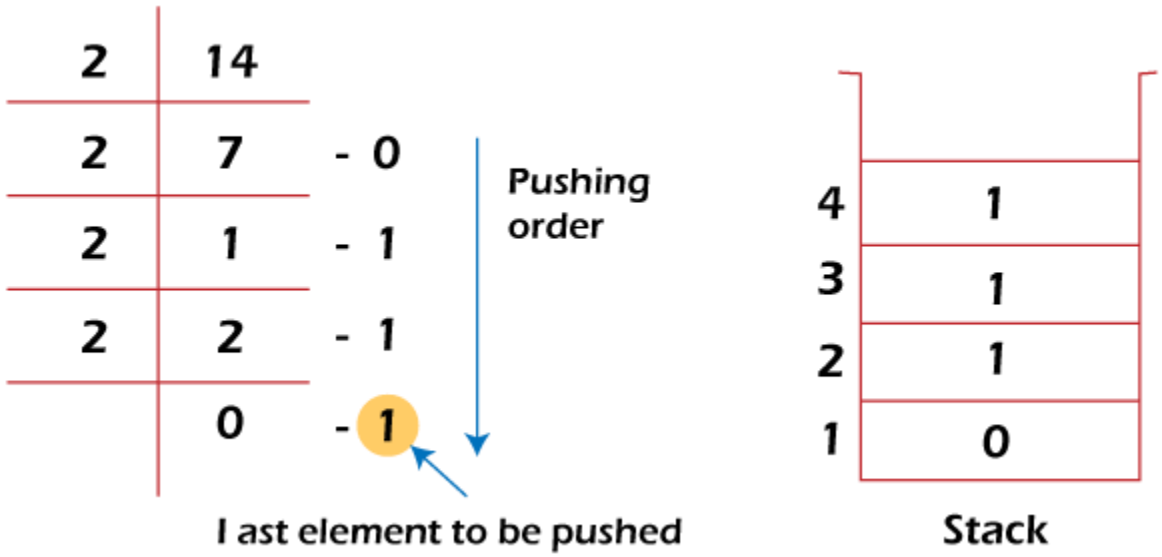
A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



Converting Decimal to Binary:

Although decimal numbers are used in most business applications, some scientific and technical applications require numbers in either binary, octal, or hexadecimal. A stack can be used to convert a number from decimal to binary/octal/hexadecimal form. For converting any decimal number to a binary number, we repeatedly divide the decimal number by two and push the remainder of each division onto the Stack until the number is reduced to 0. Then we pop the whole Stack and the result obtained is the binary equivalent of the given decimal number.

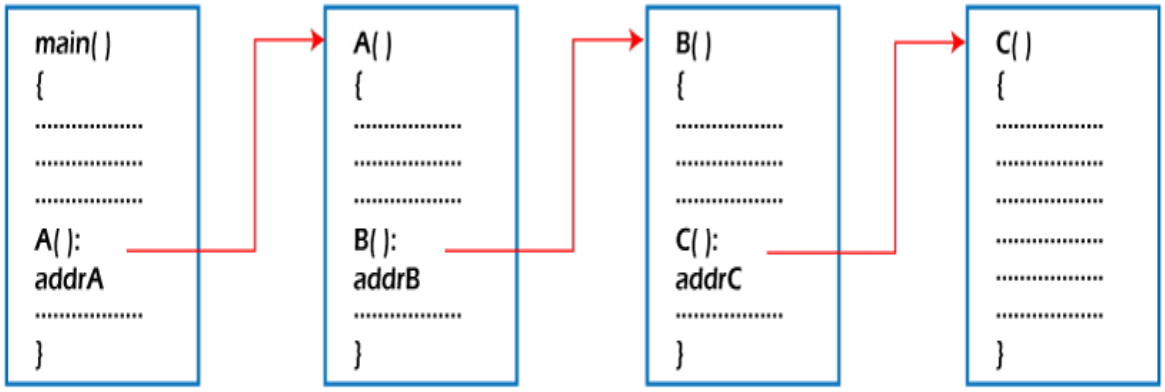
Example: Converting 14 number Decimal to Binary:



In the above example, on dividing 14 by 2, we get seven as a quotient and one as the reminder, which is pushed on the Stack. On again dividing seven by 2, we get three as quotient and 1 as the reminder, which is again pushed onto the Stack. This process continues until the given number is not reduced to 0. When we pop off the Stack completely, we get the equivalent binary number **1110.**

5. Processing Function Calls:

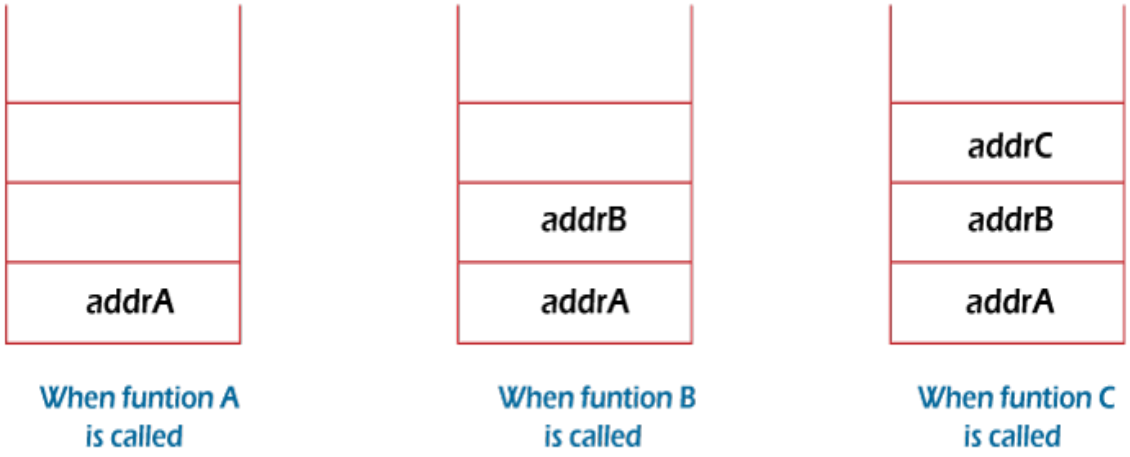
Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



Function call

When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.



Different states of stack

The above figure shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the

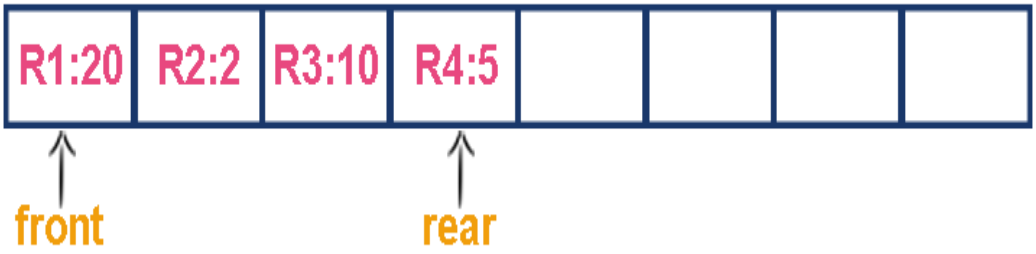
pop operation is performed, and execution continues at the address removed from the Stack. Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.

Queues Applications: Simulation, Scheduling, Shared Resource Management, Keyboard Buffer, Breadth-First Search, To handle congestion in the network etc.

Priority Queue

In the normal queue data structure, insertion is performed at the end of the queue and deletion is performed based on the FIFO principle. This queue implementation may not be suitable for all applications.

Consider a networking application where the server has to respond for requests from multiple clients using queue data structure. Assume four requests arrived at the queue in the order of R1, R2, R3 & R4 where R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. A queue is as follows...



Now, check to wait time of each request that to be completed.

1. **R1 : 20 units of time**
2. **R2 : 22 units of time (R2 must wait until R1 completes 20 units and R2 itself requires 2 units. Total 22 units)**
3. **R3 : 32 units of time (R3 must wait until R2 completes 22 units and R3 itself requires 10 units. Total 32 units)**
4. **R4 : 37 units of time (R4 must wait until R3 completes 35 units and R4 itself requires 5 units. Total 37 units)**

Here, the average waiting time for all requests (R1, R2, R3 and R4) is (20+22+32+37)/4 ≈ 27 units of time.

That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is 27 units of time.

Now, consider another way of serving these requests. If we serve according to their required amount of time, first we serve R2 which has minimum time (2 units) requirement. Then serve R4 which has second minimum time (5 units) requirement and then serve R3 which has third minimum time (10 units) requirement and finally R1 is served which has maximum time (20 units) requirement.

Now, check to wait time of each request that to be completed.

1. **R2 : 2 units of time**
2. **R4 : 7 units of time (R4 must wait until R2 completes 2 units and R4 itself requires 5 units. Total 7 units)**
3. **R3 : 17 units of time (R3 must wait until R4 completes 7 units and R3 itself requires 10 units. Total 17 units)**
4. **R1 : 37 units of time (R1 must wait until R3 completes 17 units and R1 itself requires 20 units. Total 37 units)**

Here, the average waiting time for all requests (R1, R2, R3 and R4) is (2+7+17+37)/4 ≈ 15 units of time.

From the above two situations, it is very clear that the second method server can complete all four requests with very less time compared to the first method. This is what exactly done by the priority queue.

Priority queue is a variant of a queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

There are two types of priority queues they are as follows...

1. **Max Priority Queue**
2. **Min Priority Queue**

1. Max Priority Queue

In a max priority queue, elements are inserted in the order in which they arrive the queue and the maximum value is always removed first from the queue. For example, assume that we insert in the order 8, 3, 2 & 5 and they are removed in the order 8, 5, 3, 2.

The following are the operations performed in a Max priority queue...

1. **isEmpty() - Check whether queue is Empty.**
2. **insert() - Inserts a new value into the queue.**
3. **findMax() - Find maximum value in the queue.**
4. **remove() - Delete maximum value from the queue.**

Max Priority Queue Representations

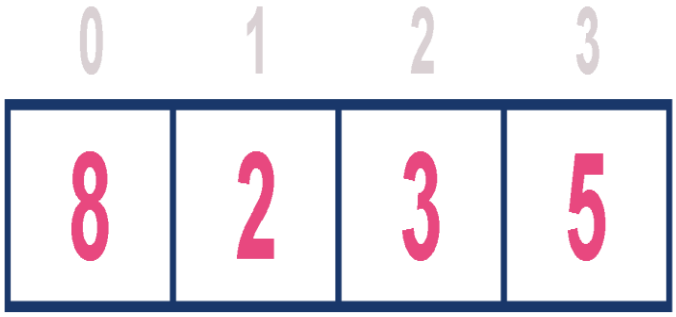
There are 6 representations of max priority queue.

1. **Using an Unordered Array (Dynamic Array)**
2. **Using an Unordered Array (Dynamic Array) with the index of the maximum value**
3. **Using an Array (Dynamic Array) in Decreasing Order**
4. **Using an Array (Dynamic Array) in Increasing Order**
5. **Using Linked List in Increasing Order**
6. **Using Unordered Linked List with reference to node with the maximum value**

#1. Using an Unordered Array (Dynamic Array)

In this representation, elements are inserted according to their arrival order and the largest element is deleted first from the max priority queue.

For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

insert() - New element is added at the end of the queue. This operation requires **O(1)** time complexity which means constant time complexity.

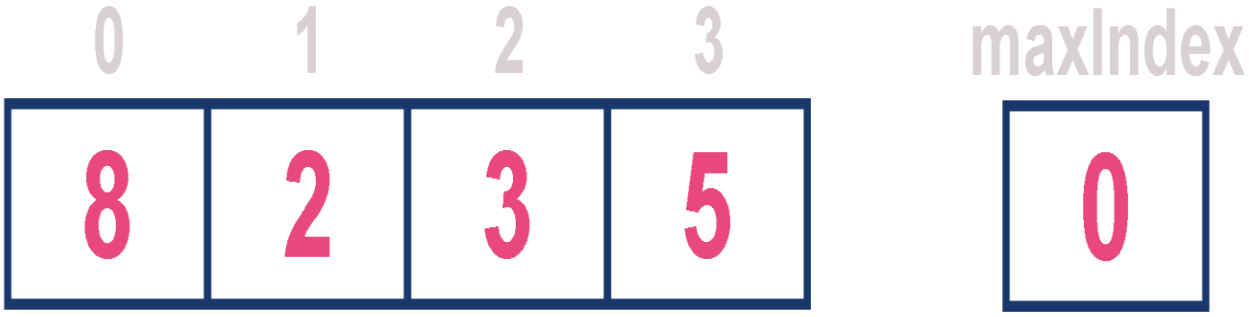
findMax() - To find the maximum element in the queue, we need to compare it with all the elements in the queue. This operation requires **O(n)** time complexity.

remove() - To remove an element from the max priority queue, first we need to find the largest element using **findMax()** which requires **O(n)** time complexity, then that

element is deleted with constant time complexity **O(1)**. The remove() operation requires **O(n) + O(1) ≈ O(n)** time complexity.

#2. Using an Unordered Array (Dynamic Array) with the index of the maximum value

In this representation, elements are inserted according to their arrival order and the largest element is deleted first from max priority queue.
For example, assume that elements are inserted in the order of 8, 2, 3 and 5. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

insert() - New element is added at the end of the queue with **O(1)** time complexity and for each insertion we need to update maxIndex with **O(1)** time complexity. This operation requires **O(1)** time complexity which means constant time complexity.

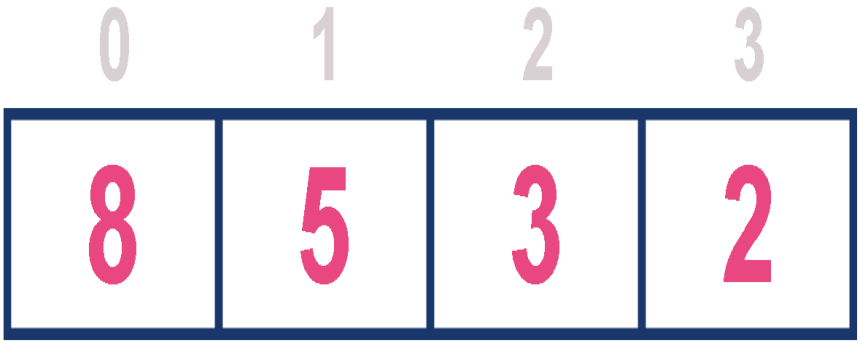
findMax() - Finding the maximum element in the queue is very simple because index of the maximum element is stored in maxIndex. This operation requires **O(1)** time complexity.

remove() - To remove an element from the queue, first we need to find the largest element using **findMax()** which requires **O(1)** time complexity, then that element is deleted with constant time complexity **O(1)** and finally we need to update the next largest element index value in maxIndex which requires **O(n)** time complexity. The remove() operation requires **O(1)+O(1)+O(n) ≈ O(n)** time complexity.

#3. Using an Array (Dynamic Array) in Decreasing Order

In this representation, elements are inserted according to their value in decreasing order and largest element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 8, 5, 3 and 2. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

insert() - New element is added at a particular position based on the decreasing order of elements which requires **O(n)** time complexity as it needs to shift existing elements inorder to insert new element in decreasing order. This insert() operation requires **O(n)** time complexity.

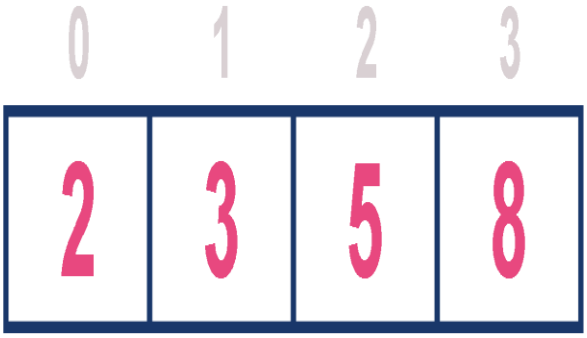
findMax() - Finding the maximum element in the queue is very simple because maximum element is at the beginning of the queue. This findMax() operation requires **O(1)** time complexity.

remove() - To remove an element from the max priority queue, first we need to find the largest element using **findMax()** operation which requires **O(1)** time complexity, then that element is deleted with constant time complexity **O(1)** and finally we need to rearrange the remaining elements in the list which requires **O(n)** time complexity. This remove() operation requires **O(1) + O(1) + O(n) ≈ O(n)** time complexity.

#4. Using an Array (Dynamic Array) in Increasing Order

In this representation, elements are inserted according to their value in increasing order and maximum element is deleted first from max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

isEmpty() - If '**front == -1**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

insert() - New element is added at a particular position in the increasing order of elements into the queue which requires **O(n)** time complexity as it needs to shift existing elements to maintain increasing order of elements. This insert() operation requires **O(n)** time complexity.

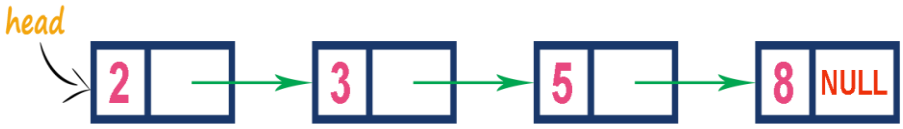
findMax() - Finding the maximum element in the queue is very simple becuase maximum element is at the end of the queue. This findMax() operation requires **O(1)** time complexity.

remove() - To remove an element from the queue first we need to find the largest element using **findMax()** which requires **O(1)** time complexity, then that element is deleted with constant time complexity **O(1)**. Finally, we need to rearrange the remaining elements to maintain increasing order of elements which requires **O(n)** time complexity. This remove() operation requires **O(1) + O(1) + O(n) ≈ O(n)** time complexity.

#5. Using Linked List in Increasing Order

In this representation, we use a single linked list to represent max priority queue. In this representation, elements are inserted according to their value in increasing order and a node with the maximum value is deleted first from the max priority queue.

For example, assume that elements are inserted in the order of 2, 3, 5 and 8. And they are removed in the order of 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

isEmpty() - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

insert() - New element is added at a particular position in the increasing order of elements which requires **O(n)** time complexity. This insert() operation requires **O(n)** time complexity.

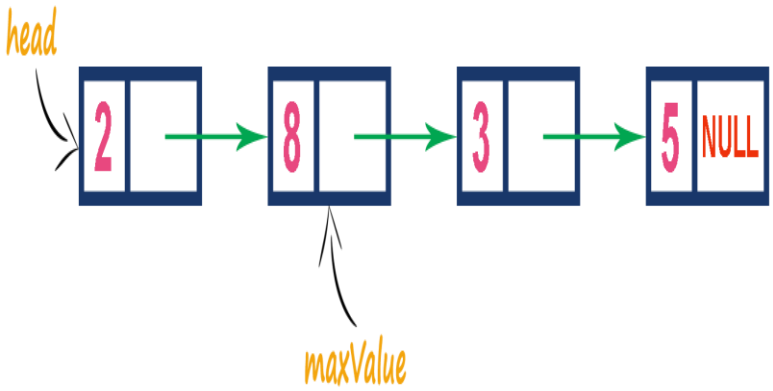
findMax() - Finding the maximum element in the queue is very simple because maximum element is at the end of the queue. This findMax() operation requires **O(1)** time complexity.

remove() - Removing an element from the queue is simple because the largest element is last node in the queue. This remove() operation requires **O(1)** time complexity.

#6. Using Unordered Linked List with reference to node with the maximum value

In this representation, we use a single linked list to represent max priority queue. We always maintain a reference (maxValue) to the node with the maximum value in the queue. In this representation, elements are inserted according to their arrival and the node with the maximum value is deleted first from the max priority queue.

For example, assume that elements are inserted in the order of 2, 8, 3 and 5. And they are removed in the order of 8, 5, 3 and 2.



Now, let us analyze each operation according to this representation...

isEmpty() - If '**head == NULL**' queue is Empty. This operation requires **O(1)** time complexity which means constant time complexity.

insert() - New element is added at end of the queue which requires **O(1)** time complexity. And we need to update maxValue reference with address of largest element in the queue which requires **O(1)** time complexity. This insert() operation requires **O(1)** time complexity.

findMax() - Finding the maximum element in the queue is very simple because the address of largest element is stored at maxValue. This findMax() operation requires **O(1)** time complexity.

remove() - Removing an element from the queue is deleting the node which is referenced by maxValue which requires **O(1)** time complexity. And then we need to update maxValue reference to new node with maximum value in the queue which requires **O(n) time complexity**. This remove() operation requires **O(n)** time complexity.

2. Min Priority Queue Representations

Min Priority Queue is similar to max priority queue except for the removal of maximum element first. We remove minimum element first in the min-priority queue.

The following operations are performed in Min Priority Queue...

1. **isEmpty()** - Check whether queue is Empty.
2. **insert()** - Inserts a new value into the queue.
3. **findMin()** - Find minimum value in the queue.
4. **remove()** - Delete minimum value from the queue.

UNIT 5

Graphs ,Trees

UNIT-V

Graphs -Introduction, Directed vs Undirected Graphs, Weighted vs Unweighted Graphs, Representations, Breadth First Search, Depth First Search.
Trees - Overview of Trees, Tree Terminology, Binary Trees: Introduction, Implementation, Applications. Tree Traversals, Binary Search Trees: Introduction, Implementation, AVL Trees: Introduction, Rotations, Implementation.

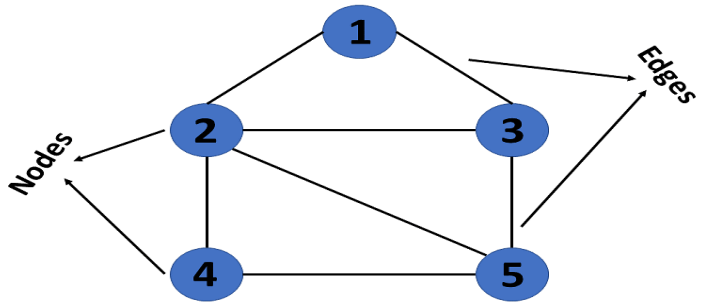
Graphs

Graphs in data structures are non-linear data structures made up of a finite number of nodes or vertices and the edges that connect them.

Graphs in data structures are used to address real-world problems in which it represents the problem area as a network like telephone networks, circuit networks, and social networks. For example, it can represent a single user as nodes or vertices in a telephone network, while the link between them via telephone represents edges.

Graphs in Data Structure

A graph is a non-linear kind of data structure made up of nodes or vertices and edges. The edges connect any two nodes in the graph, and the nodes are also known as vertices.



This graph has a set of vertices V= { 1,2,3,4,5} and a set of edges E= { (1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(4,50 }.

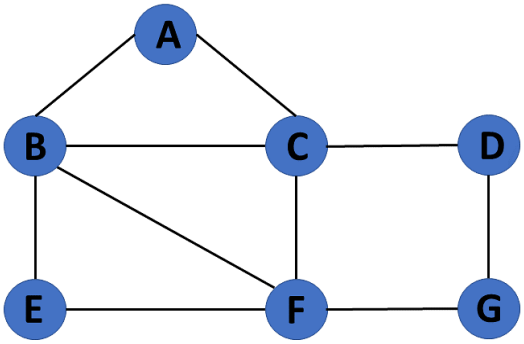
Now that you've learned about the definition of graphs in data structures, you will learn about their various types.

Types of Graphs in Data Structures

There are different types of graphs in data structures, each of which is detailed below.

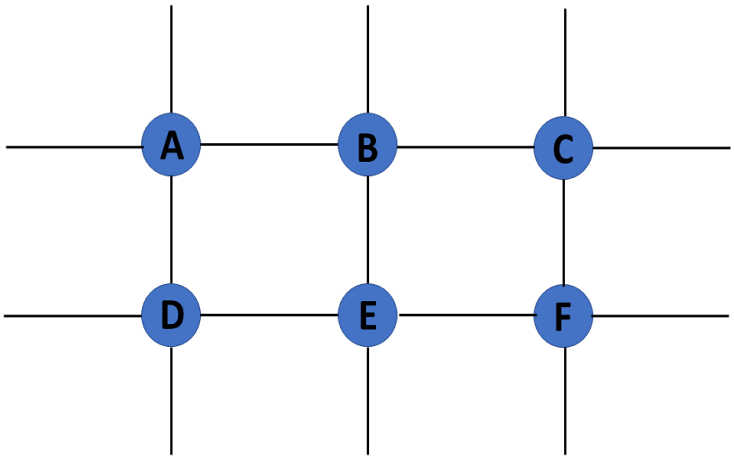
1. Finite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number



2. Infinite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.

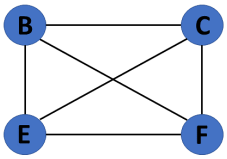


3. Trivial Graph

A graph G= (V, E) is trivial if it contains only a single vertex and no edges.

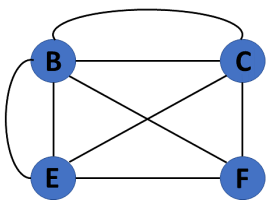
4. Simple Graph

If each pair of nodes or vertices in a graph G=(V, E) has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices, depicting one-to-one interactions between two elements.



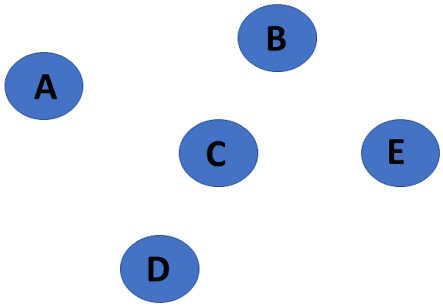
5. Multi Graph

If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



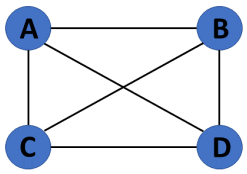
6. Null Graph

It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph G= (V, E) is a null graph.



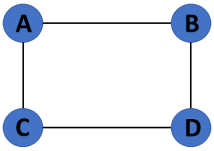
7. Complete Graph

If a graph G= (V, E) is also a simple graph, it is complete. Using the edges, with n number of vertices must be connected. It's also known as a full graph because each vertex's degree must be n-1.



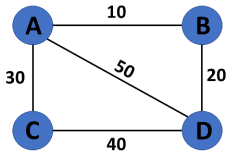
9. Regular Graph

If a graph G= (V, E) is a simple graph with the same degree at each vertex, it is a regular graph. As a result, every whole graph is a regular graph.



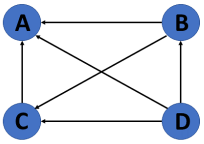
10. Weighted Graph

A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



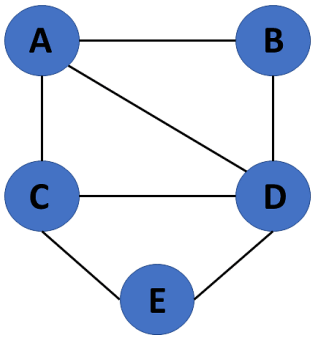
11. Directed Graph

A directed graph also referred to as a digraph, is a set of nodes connected by edges, each with a direction.



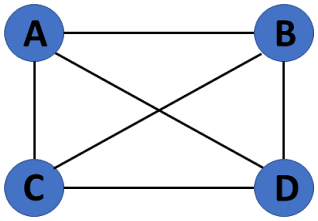
12. Undirected Graph

An undirected graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction. You can form an undirected graph with a finite number of vertices and edges.



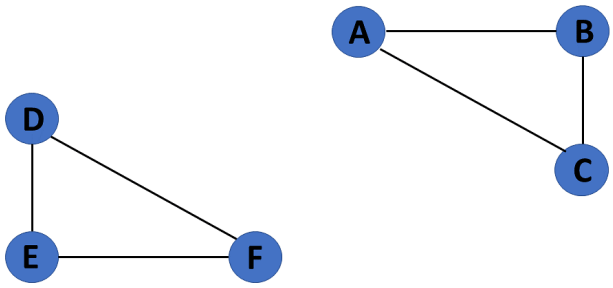
13. Connected Graph

If there is a path between one vertex of a graph data structure and any other vertex, the graph is connected.



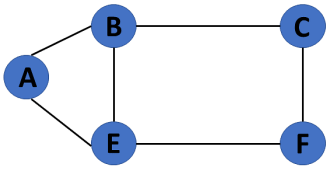
14. Disconnected Graph

When there is no edge linking the vertices, you refer to the null graph as a disconnected graph.



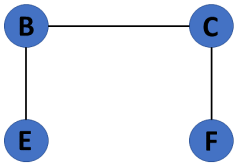
15. Cyclic Graph

If a graph contains at least one graph cycle, it is considered to be cyclic.



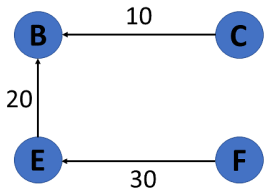
16. Acyclic Graph

When there are no cycles in a graph, it is called an acyclic graph.



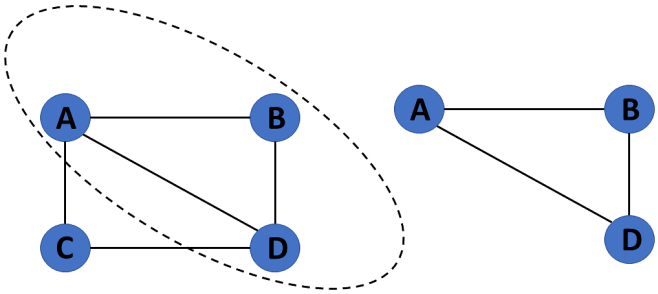
17. Directed Acyclic Graph

It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



18. Subgraph

The vertices and edges of a graph that are subsets of another graph are known as a subgraph.



After you learn about the many types of graphs in graphs in data structures, you will move on to graph terminologies.

Terminologies of Graphs in Data Structures

Following are the basic terminologies of graphs in data structures:

- An edge is one of the two primary units used to form graphs. Each edge has two ends, which are vertices to which it is attached.
- If two vertices are endpoints of the same edge, they are adjacent.
- A vertex's outgoing edges are directed edges that point to the origin.
- A vertex's incoming edges are directed edges that point to the vertex's destination.
- The total number of edges occurring to a vertex in a graph is its degree.
- The out-degree of a vertex in a directed graph is the total number of outgoing edges, whereas the in-degree is the total number of incoming edges.
- A vertex with an in-degree of zero is referred to as a source vertex, while one with an out-degree of zero is known as sink vertex.
- An isolated vertex is a zero-degree vertex that is not an edge's endpoint.
- A path is a set of alternating vertices and edges, with each vertex connected by an edge.
- The path that starts and finishes at the same vertex is known as a cycle.
- A path with unique vertices is called a simple path.
- For each pair of vertices x, y, a graph is strongly connected if it contains a directed path from x to y and a directed path from y to x.
- A directed graph is weakly connected if all of its directed edges are replaced with undirected edges, resulting in a connected graph. A weakly linked graph's vertices have at least one out-degree or in-degree.
- A tree is a connected forest. The primary form of the tree is called a rooted tree, which is a free tree.
- A spanning subgraph that is also a tree is known as a spanning tree.
- A connected component is the unconnected graph's most connected subgraph.
- A bridge, which is an edge of removal, would sever the graph.

- Forest is a graph without a cycle.

Following that, you will look at the graph representation in this data structures tutorial.

Representation of Graphs in Data Structures

Graphs in data structures are used to represent the relationships between objects. Every graph consists of a set of points known as vertices or nodes connected by lines known as edges. The vertices in a network represent entities.

The most frequent graph representations are the two that follow:

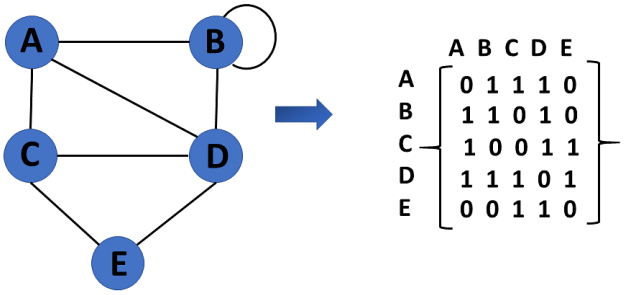
- Adjacency matrix
- Adjacency list

You'll look at these two representations of graphs in data structures in more detail:

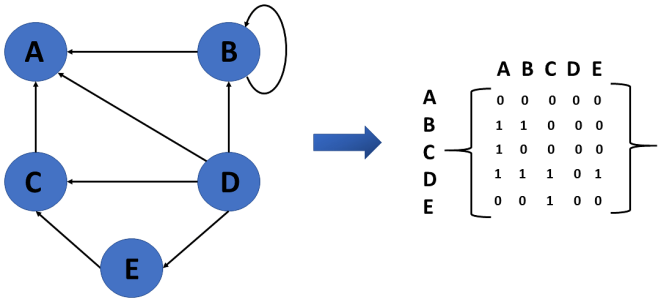
Adjacency Matrix

- A sequential representation is an adjacency matrix.
- It's used to show which nodes are next to one another. I.e., is there any connection between nodes in a graph?
- You create an MXM matrix G for this representation. If an edge exists between vertex a and vertex b, the corresponding element of G, gi,j = 1, otherwise gi,j = 0.
- If there is a weighted graph, you can record the edge's weight instead of 1s and 0s.

Undirected Graph Representation

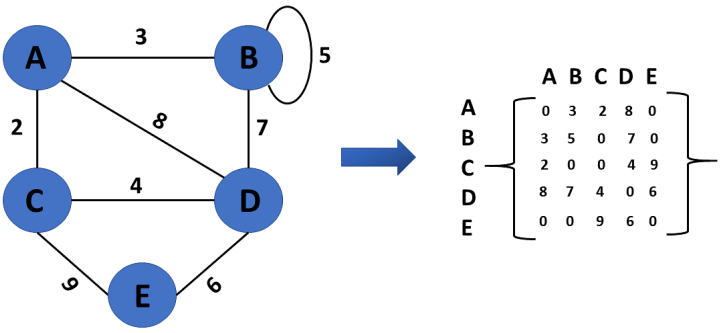


Directed Graph Representation



Weighted Undirected Graph Representation

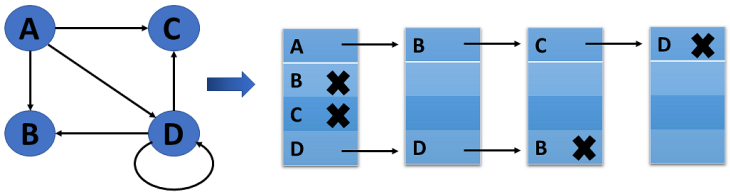
Weight or cost is indicated at the graph's edge, a weighted graph representing these values in the matrix.



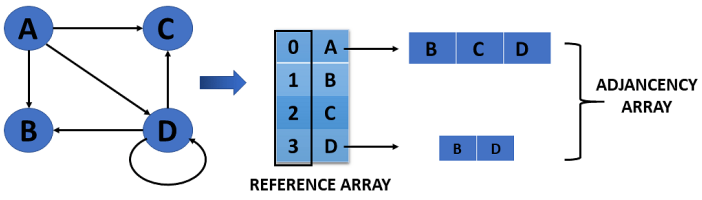
Adjacency List

- A linked representation is an adjacency list.
- You keep a list of neighbors for each vertex in the graph in this representation. It means that each vertex in the graph has a list of its neighboring vertices.
- You have an array of vertices indexed by the vertex number, and the corresponding array member for each vertex x points to a singly linked list of x's neighbors.

Weighted Undirected Graph Representation Using Linked-List



Weighted Undirected Graph Representation Using an Array



We will now see which all operations are conducted in graphs data structure after understanding the representation of graphs in the data structure.

Operations on Graphs in Data Structures

The operations you perform on the graphs in data structures are listed below:

- Creating graphs
- Insert vertex
- Delete vertex
- Insert edge
- Delete edge

You will go over each operation in detail one by one:

Creating Graphs

There are two techniques to make a graph:

1. Adjacency Matrix

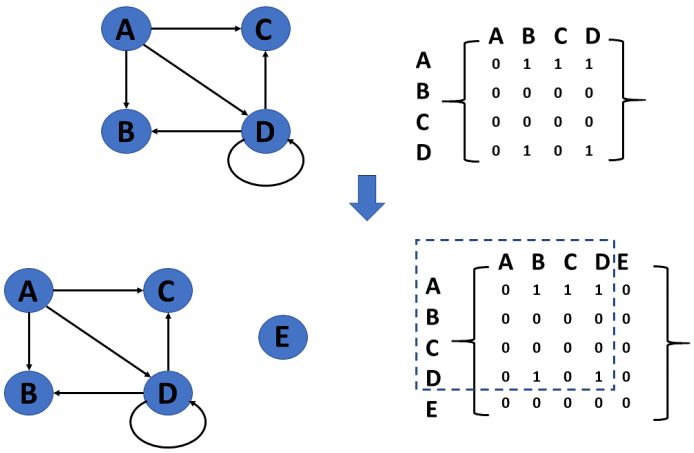
The adjacency matrix of a simple labeled graph, also known as the connection matrix, is a matrix with rows and columns labeled by graph vertices and a 1 or 0 in position depending on whether they are adjacent or not.

2. Adjacency List

A finite graph is represented by an adjacency list, which is a collection of unordered lists. Each unordered list describes the set of neighbors of a particular vertex in the graph within an adjacency list.

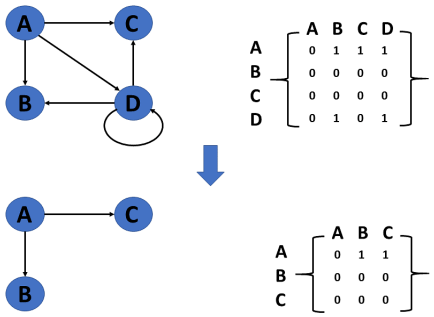
Insert Vertex

When you add a vertex that after introducing one or more vertices or nodes, the graph's size grows by one, increasing the matrix's size by one at the row and column levels.



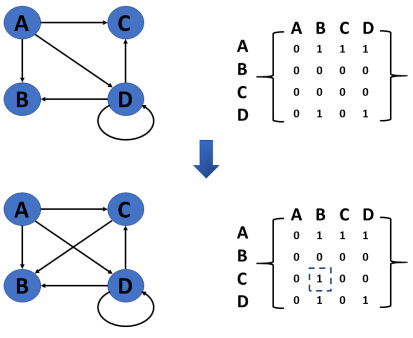
Delete Vertex

- Deleting a vertex refers to removing a specific node or vertex from a graph that has been saved.
- If a removed node appears in the graph, the matrix returns that node. If a deleted node does not appear in the graph, the matrix returns the node not available.



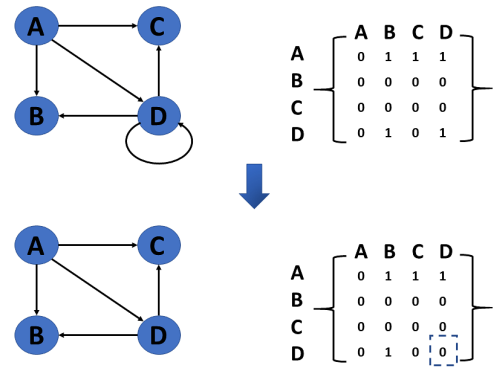
Insert Edge

Connecting two provided vertices can be used to add an edge to a graph.



Delete Edge

The connection between the vertices or nodes can be removed to delete an edge.



The types of graph traversal algorithms will be discussed next in the graphs in this data structures tutorial.

Graph Traversal Algorithm

The process of visiting or updating each vertex in a graph is known as graph traversal. The sequence in which they visit the vertices is used to classify such traversals. Graph traversal is a subset of tree traversal.

There are two techniques to implement a graph traversal algorithm:

- Breadth-first search
- Depth-first search

Breadth-First Search or BFS

BFS is a search technique for finding a node in a graph data structure that meets a set of criteria.

- It begins at the root of the graph and investigates all nodes at the current depth level before moving on to nodes at the next depth level.
- To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally you require a queue.

Algorithm of breadth-first search

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in your graph, say v1, from which you want to traverse the graph.

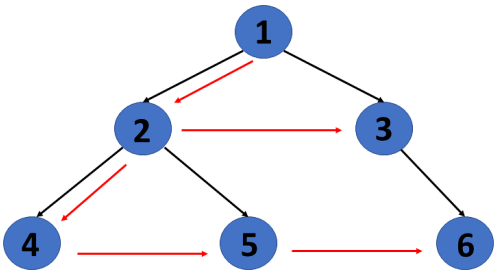
Step 3: Examine any two data structures for traversing the graph.

- Visited array (size of the graph)
- Queue data structure

Step 4: Starting from the vertex, you will add to the visited array, and afterward, you will v1's adjacent vertices to the queue data structure.

Step 5: Now, using the FIFO concept, you must remove the element from the queue, put it into the visited array, and then return to the queue to add the adjacent vertices of the removed element.

Step 6: Repeat step 5 until the queue is not empty and no vertex is left to be visited.



BFS	1	2	3	4	5	6

Depth-First Search or DFS

DFS is a search technique for finding a node in a graph data structure that meets a set of criteria.

- The depth-first search (DFS) algorithm traverses or explores data structures such as trees and graphs. The DFS algorithm begins at the root node and examines each branch as far as feasible before backtracking.
- To maintain track of the child nodes that have been encountered but not yet inspected, more memory, generally a stack, is required.

Algorithm of depth-first search

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in our graph, say v1, from which you want to begin traversing the graph.

Step 3: Examine any two data structures for traversing the graph.

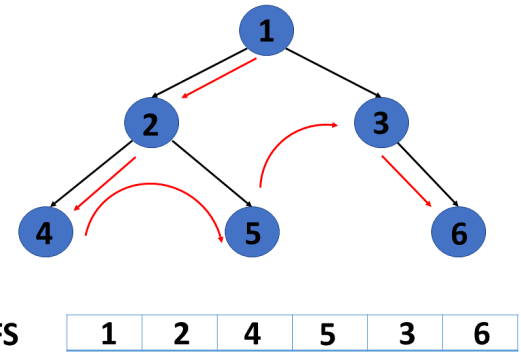
- Visited array (size of the graph)
- Stack data structure

Step 4: Insert v1 into the array's first block and push all the adjacent nodes or vertices of vertex v1 into the stack.

Step 5: Now, using the FIFO principle, pop the topmost element and put it into the visited array, pushing all of the popped element's nearby nodes into it.

Step 6: If the topmost element of the stack is already present in the array, discard it instead of inserting it into the visited array.

Step 7: Repeat step 6 until the stack data structure isn't empty.



You will now look at applications of graph data structures after understanding the graph traversal algorithm in this tutorial.

Application of Graphs in Data Structures

Following are some applications of graphs in data structures:

- Graphs are used in computer science to depict the flow of computation.
- Users on Facebook are referred to as vertices, and if they are friends, there is an edge connecting them. The Friend Suggestion system on Facebook is based on graph theory.
- We come across the Resource Allocation Graph in the Operating System, where each process and resource are regarded vertically. Edges are drawn from resources to assigned functions or from the requesting process to the desired resource. A stalemate will develop if this results in the establishment of a cycle.

- Web pages are referred to as vertices on the World Wide Web. Suppose there is a link from page A to page B that can represent an edge. This application is an illustration of a directed graph.
- Graph transformation systems manipulate graphs in memory using rules. Graph databases store and query graph-structured data in a transaction-safe, permanent manner.

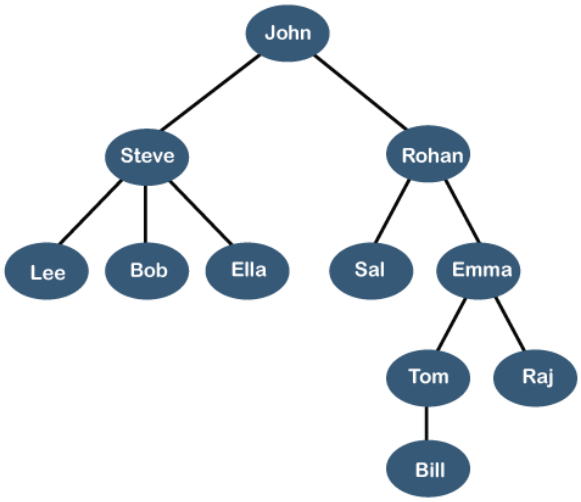
Trees in Data Structures

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



Let's understand some key points of the Tree data structure.

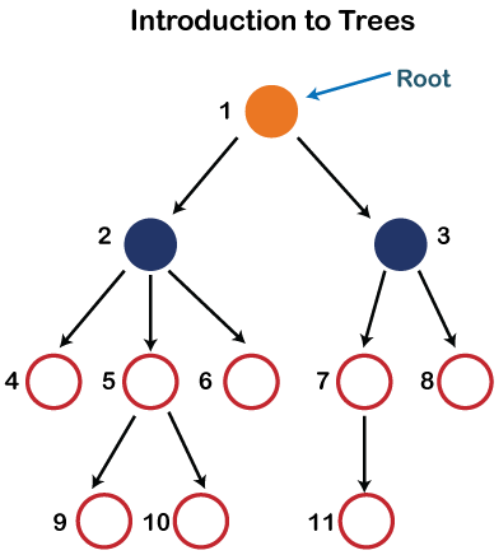
- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree

structure, the node contains the name of the employee, so the type of data would be a string.

- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure.

Let's consider the tree structure, which is shown below:



In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

Root: The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.

Child node: If the node is a descendant of any node, then the node is known as a child node.

Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node.

Sibling: The nodes that have the same parent are known as siblings.

Leaf Node:- The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

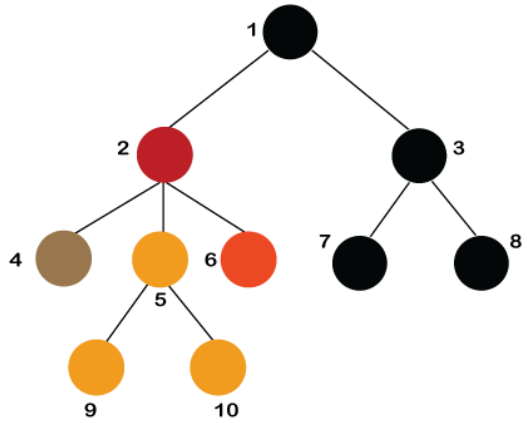
Internal nodes: A node has atleast one child node known as an ***internal***

Ancestor node:- An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

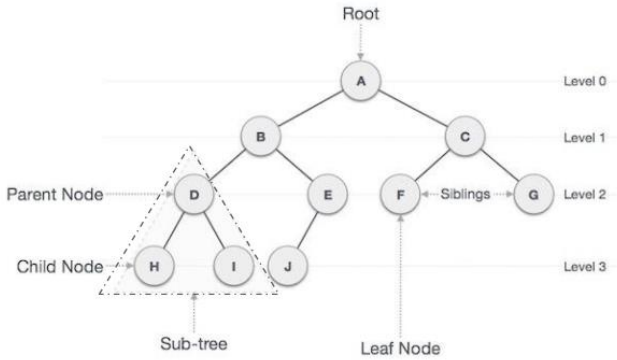
Descendant: The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.



- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.



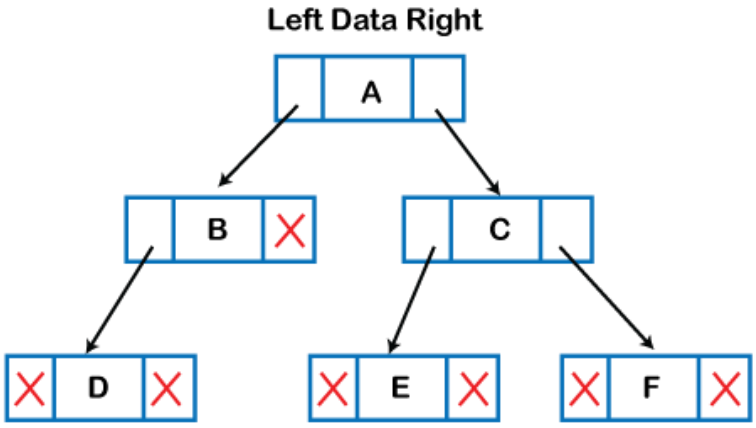
Terminology	Description	Example From Diagram
Root	Root is a special node in a tree. The entire tree originates from it. It does not have a parent.	Node A
Parent Node	Parent node is an immediate predecessor of a node.	B is parent of D & E
Child Node	All immediate successors of a node are its children.	D & E are children of B
Leaf	Node which does not have any child is called as leaf	H, I, J, F and G are leaf nodes
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.	Line between A & B is edge
Siblings	Nodes with the same parent are called Siblings.	D & E are siblings
Path / Traversing	Path is a number of successive edges from source node to destination node.	A – B – E – J is path from node A to E
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.	A, B, C, D & E can have height. Height of A is no. of edges between A and H, as that is the longest path, which is 3. Height of C is 1
Levels of node	Level of a node represents the generation of a node. If the root	Level of H, I & J is 3. Level of D, E, F & G is 2

	node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on	
Degree of Node	Degree of a node represents the number of children of a node.	Degree of D is 2 and of E is 1
Sub tree	Descendants of a node represent subtree.	Nodes D, H, I represent one subtree.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

Applications of trees

The following are the applications of trees:

Storing naturally hierarchical data: Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.

Organize data: It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.

Trie: It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.

Heap: It is also a tree data structure implemented using arrays. It is used to implement priority queues.

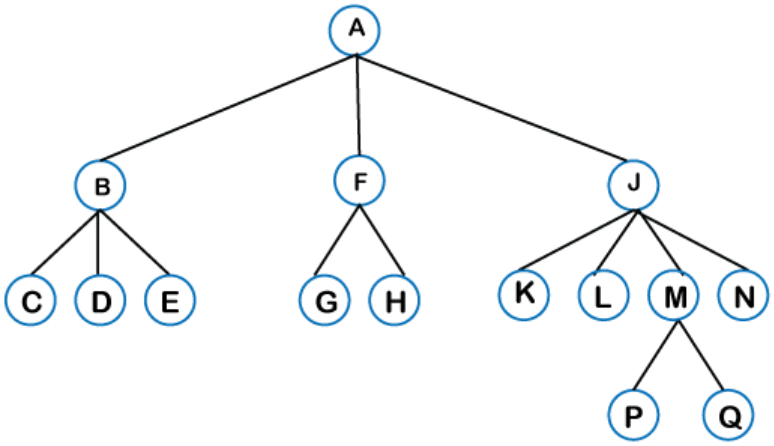
B-Tree and B+Tree: B-Tree and B+Tree are the tree data structures used to implement indexing in databases.

Routing table: The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

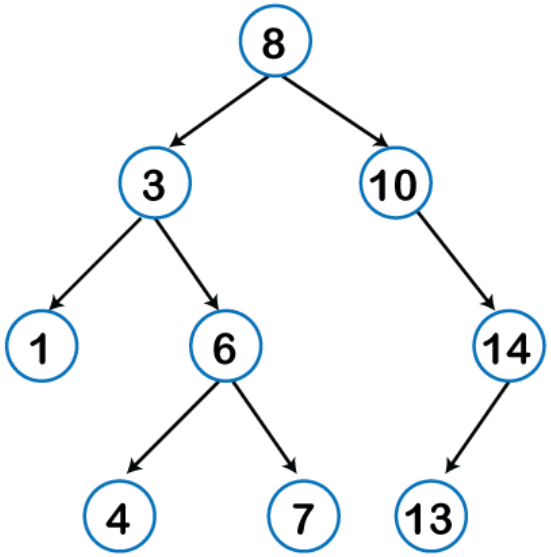
The following are the types of a tree data structure:

General tree: The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.
Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

Binary tree: Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



Binary Search tree: Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer).
Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

AVL tree

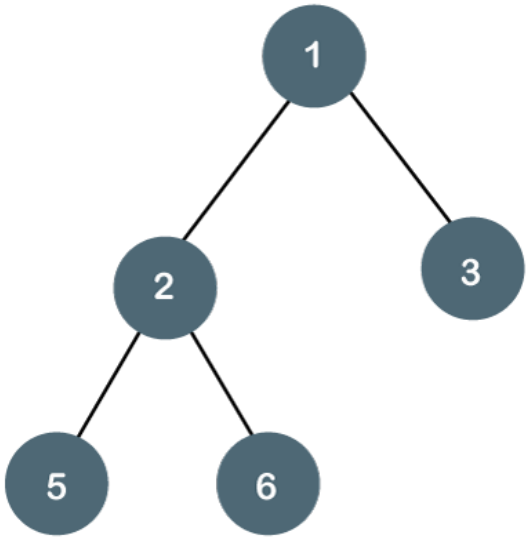
It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the ***difference between the height of the left subtree and the height of the right subtree***. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

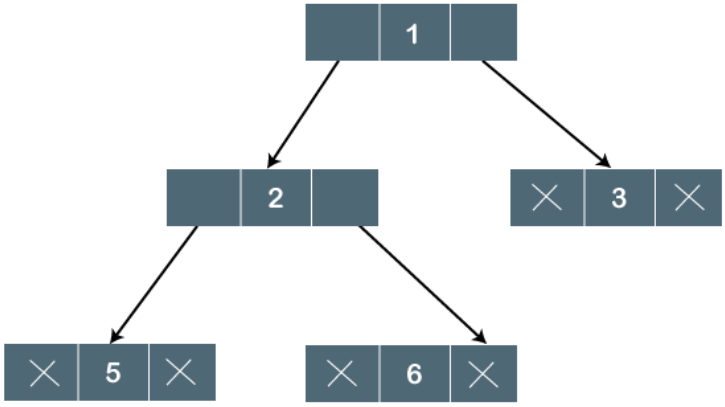
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In

general, the maximum number of nodes possible at height h is (2^0 + 2^1 + 2^2+….2^h) = 2^{h+1} -1.

- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$$n = 2^{h+1} - 1$$

$$n+1 = 2^{h+1}$$

Taking log on both the sides,

$$\log_2(n+1) = \log2(2^{h+1})$$

$$\log_2(n+1) = h+1$$

$$\mathbf{h = \log_2(n+1) - 1}$$

The maximum height can be computed as:

As we know that,

$$n = h+1$$

$$\mathbf{h= n-1}$$

Types of Binary Tree

There are four types of Binary tree:

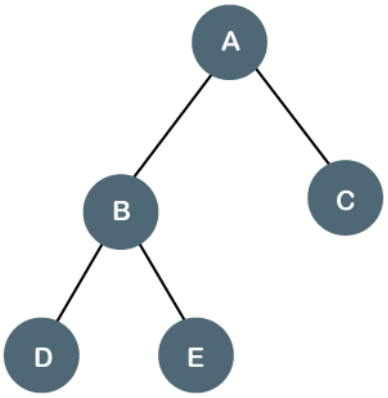
- **Full/ proper/ strict Binary tree**
- **Complete Binary tree**
- **Perfect Binary tree**
- **Degenerate Binary tree**
- **Balanced Binary tree**

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children.

The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of the full binary tree can be computed as:

$n= 2*h - 1$

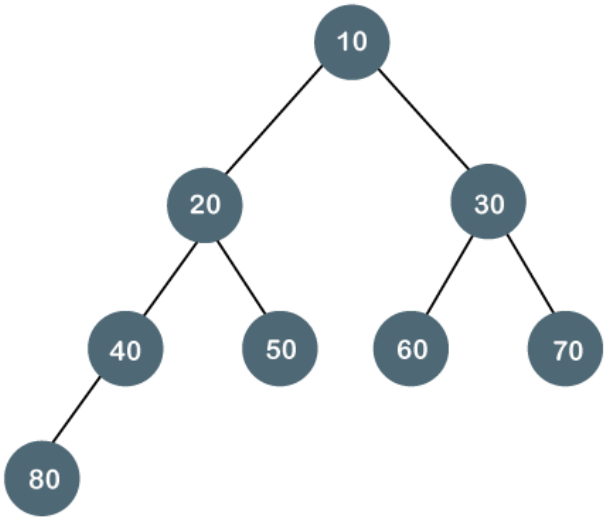
$n+1 = 2*h$

$h = n+1/2$

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



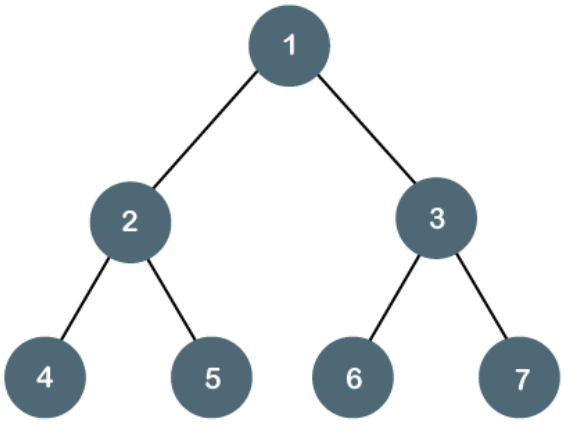
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

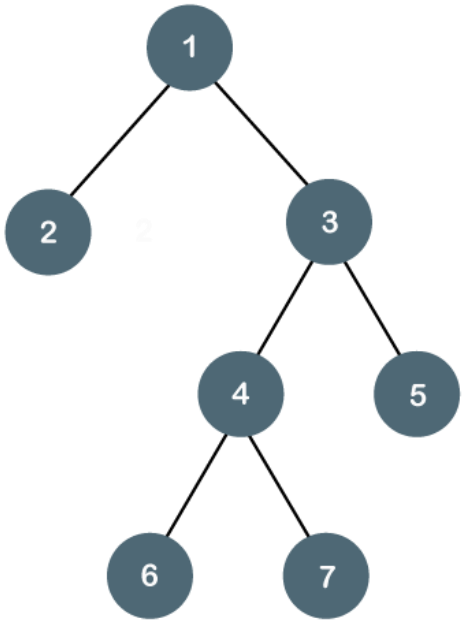
Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Let's look at a simple example of a perfect binary tree.

The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

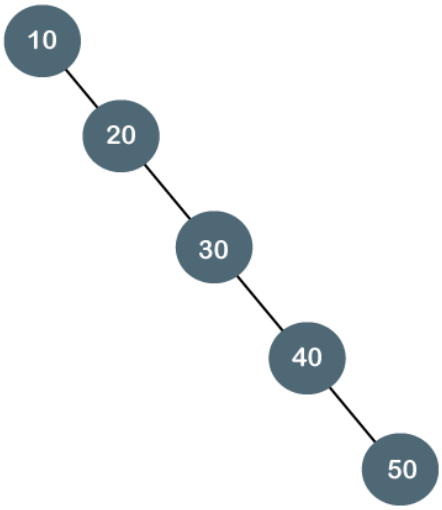


Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

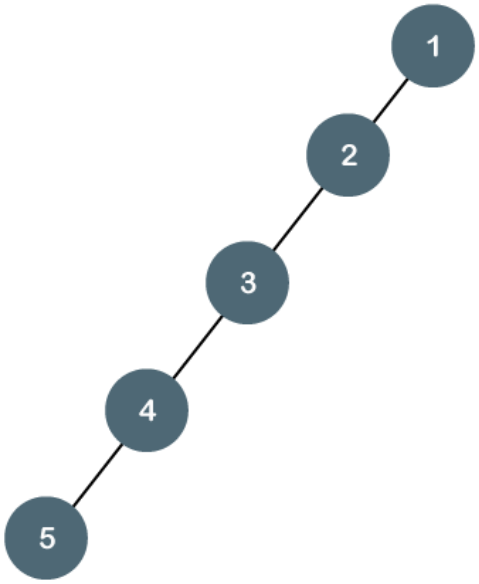
Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

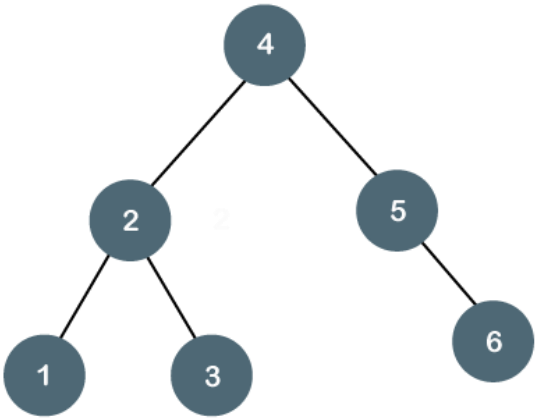


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

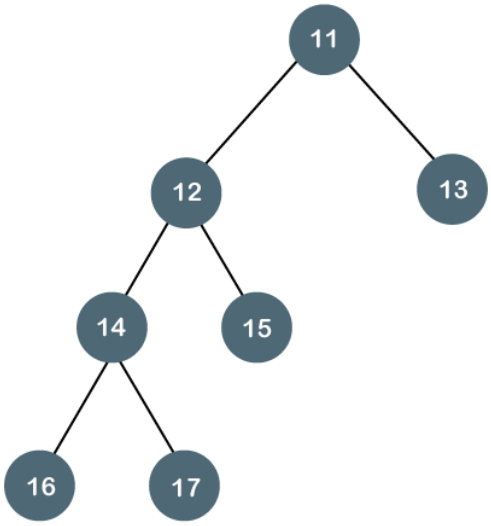
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

Create Root Node

We just create a Node class and add assign a value to the node. This becomes tree with only a root node.

Example

```
class Node:
  def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data
  def PrintTree(self):
    print(self.data)

root = Node(10)
root.PrintTree()
```

Output

When the above code is executed, it produces the following result −

```
10
```

Inserting into a Tree

To insert into a tree we use the same node class created above and add a insert class to it. The insert class compares the value of the node to the parent node and decides to add it as a left node or a right node. Finally the PrintTree class is used to print the tree.

Example

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data

   def insert(self, data):
# Compare the new value with the parent node
      if self.data:
         if data < self.data:
            if self.left is None:
               self.left = Node(data)
            else:
               self.left.insert(data)
            elif data > self.data:
               if self.right is None:
                  self.right = Node(data)
               else:
                  self.right.insert(data)
```

```
    else:
      self.data = data

# Print the tree
  def PrintTree(self):
    if self.left:
      self.left.PrintTree()
    print( self.data),
    if self.right:
      self.right.PrintTree()

# Use the insert method to add nodes
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
root.PrintTree()
```

Output

When the above code is executed, it produces the following result −

```
3 6 12 14
```

Traversing a Tree

The tree can be traversed by deciding on a sequence to visit each node. As we can clearly see we can start at a node then visit the left sub-tree first and right sub-tree next. Or we can also visit the right sub-tree first and left sub-tree next. Accordingly there are different names for these tree traversal methods.

Tree Traversal Algorithms

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

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then, we create an insert function to add data to the tree. Finally, the In-order traversal logic is implemented by creating an empty list and adding the left node first followed by the root or parent node.

At last the left node is added to complete the In-order traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed.

Example

```
class Node:
  def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data
# Insert Node
  def insert(self, data):
    if self.data:
      if data < self.data:
        if self.left is None:
```

```
            self.left = Node(data)
          else:
            self.left.insert(data)
        else data > self.data:
          if self.right is None:
            self.right = Node(data)
          else:
            self.right.insert(data)
      else:
        self.data = data
# Print the Tree
   def PrintTree(self):
      if self.left:
        self.left.PrintTree()
      print( self.data),
      if self.right:
        self.right.PrintTree()
# Inorder traversal
# Left -> Root -> Right
   def inorderTraversal(self, root):
      res = []
      if root:
        res = self.inorderTraversal(root.left)
        res.append(root.data)
        res = res + self.inorderTraversal(root.right)
      return res
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
```

```
root.insert(42)
print(root.inorderTraversal(root))
```

Output

When the above code is executed, it produces the following result −

```
[10, 14, 19, 27, 31, 35, 42]
```

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then, we create an insert function to add data to the tree. Finally, the Pre-order traversal logic is implemented by creating an empty list and adding the root node first followed by the left node.

At last, the right node is added to complete the Pre-order traversal. Please note that, this process is repeated for each sub-tree until all the nodes are traversed.

Example

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data
# Insert Node
   def insert(self, data):
      if self.data:
         if data < self.data:
            if self.left is None:
```

```
            self.left = Node(data)
          else:
            self.left.insert(data)
        elif data > self.data:
          if self.right is None:
            self.right = Node(data)
          else:
            self.right.insert(data)
        else:
          self.data = data
# Print the Tree
  def PrintTree(self):
    if self.left:
      self.left.PrintTree()
    print( self.data),
    if self.right:
      self.right.PrintTree()
# Preorder traversal
# Root -> Left ->Right
  def PreorderTraversal(self, root):
    res = []
    if root:
      res.append(root.data)
      res = res + self.PreorderTraversal(root.left)
      res = res + self.PreorderTraversal(root.right)
    return res
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
```

```
root.insert(42)
print(root.PreorderTraversal(root))
```

Output

When the above code is executed, it produces the following result −

```
[27, 14, 10, 19, 35, 31, 42]
```

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.

In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes. Then, we create an insert function to add data to the tree. Finally, the Post-order traversal logic is implemented by creating an empty list and adding the left node first followed by the right node.

At last the root or parent node is added to complete the Post-order traversal. Please note that, this process is repeated for each sub-tree until all the nodes are traversed.

Example

```
class Node:
  def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data
# Insert Node
  def insert(self, data):
    if self.data:
      if data < self.data:
        if self.left is None:
          self.left = Node(data)
```

```
            else:
                self.left.insert(data)
        else if data > self.data:
            if self.right is None:
                self.right = Node(data)
            else:


                self.right.insert(data)
    else:
        self.data = data
# Print the Tree
  def PrintTree(self):
    if self.left:
        self.left.PrintTree()
print( self.data),
if self.right:
self.right.PrintTree()
# Postorder traversal
# Left ->Right -> Root
def PostorderTraversal(self, root):
res = []
if root:
res = self.PostorderTraversal(root.left)
res = res + self.PostorderTraversal(root.right)
res.append(root.data)
return res
root = Node(27)
root.insert(14)
root.insert(35)
root.insert(10)
root.insert(19)
root.insert(31)
```

```
root.insert(42)
print(root.PostorderTraversal(root))
```

Output

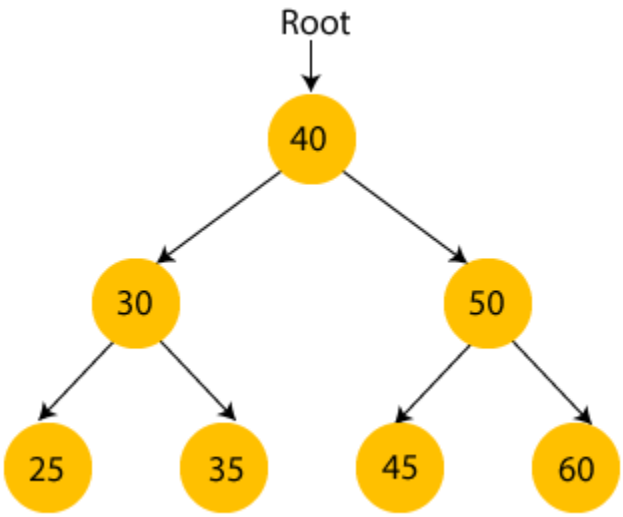
When the above code is executed, it produces the following result −

```
[10, 19, 14, 31, 42, 35, 27]
```

Binary Search tree

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

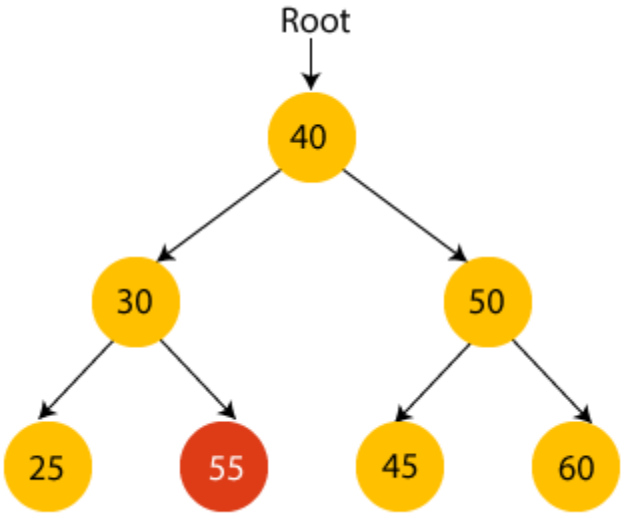
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree

Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

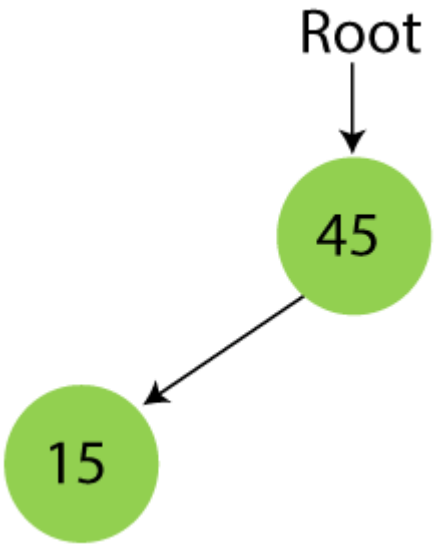
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.

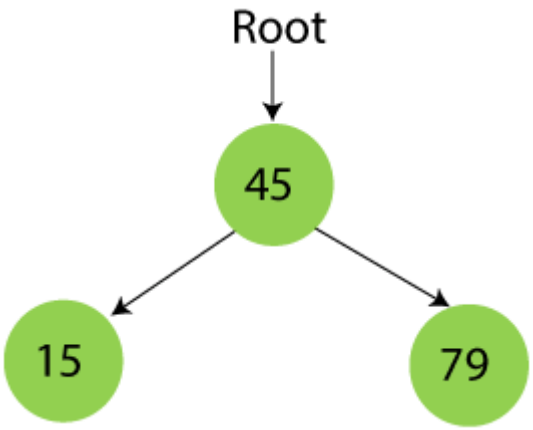
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



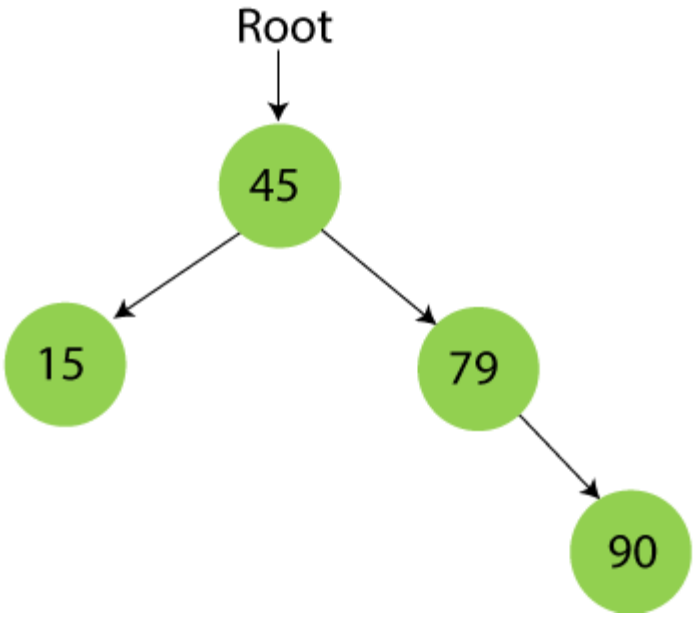
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



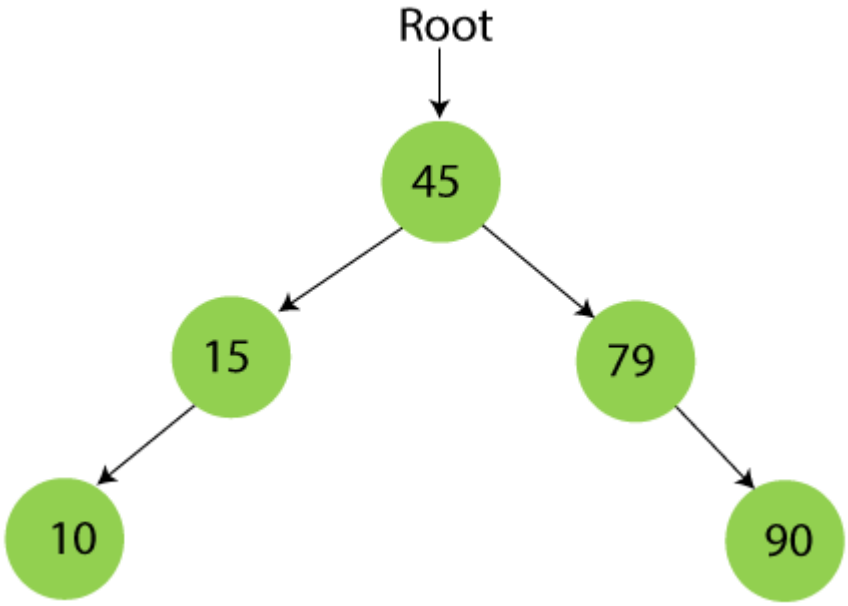
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



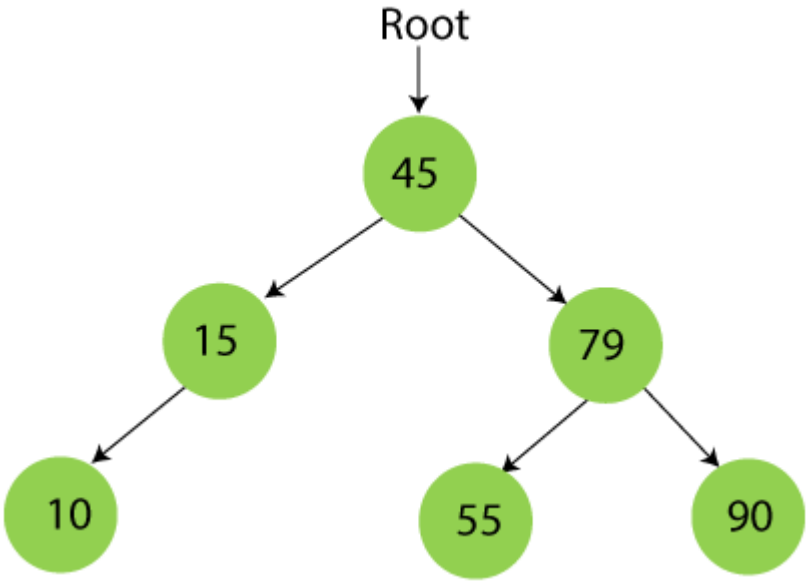
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



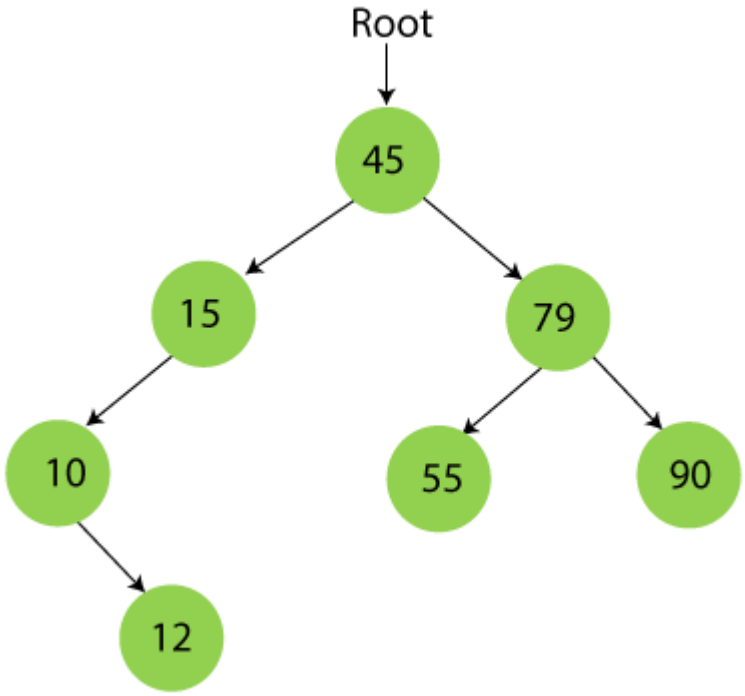
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



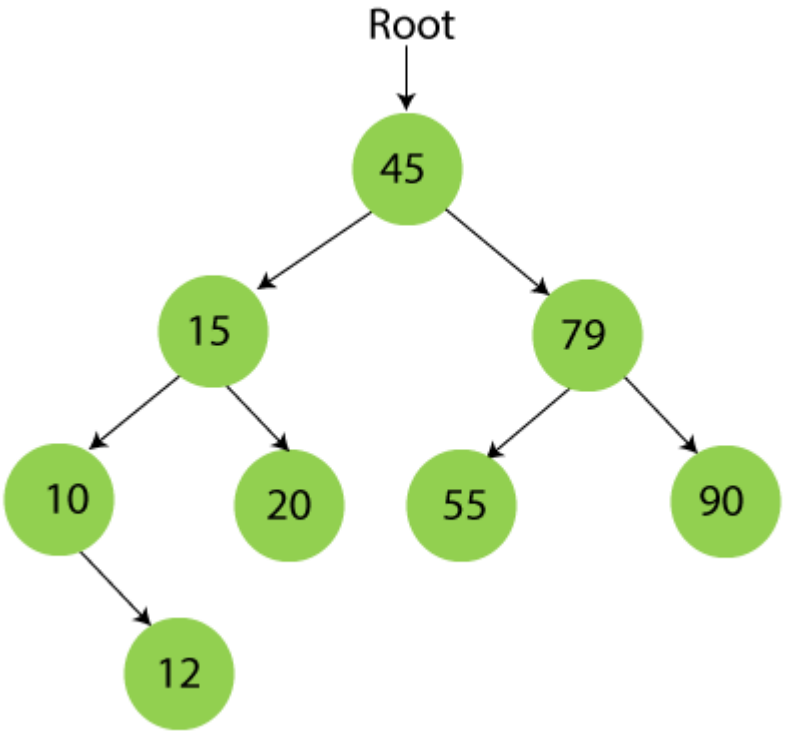
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



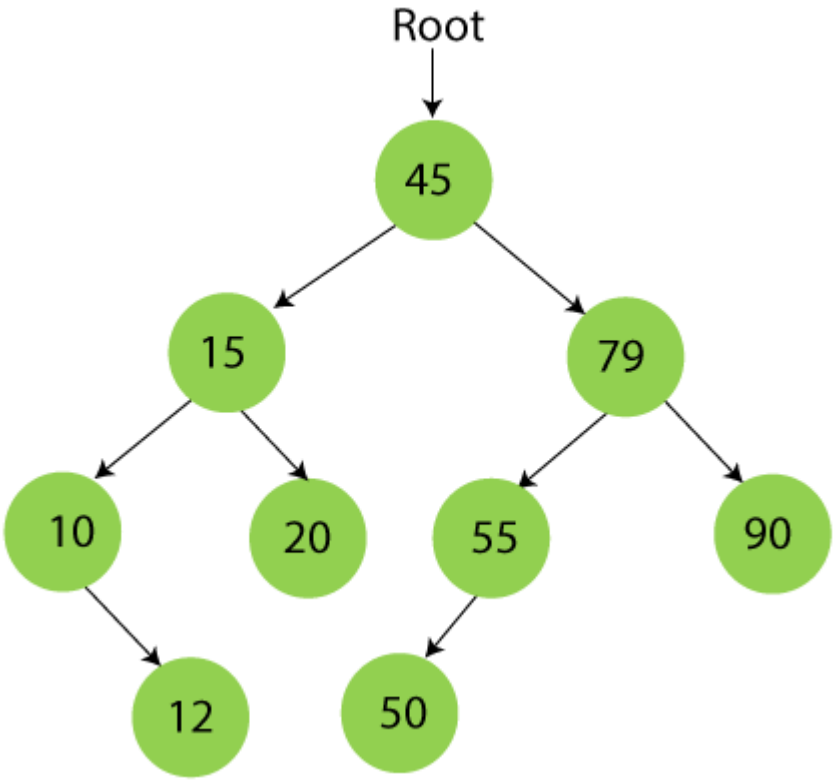
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

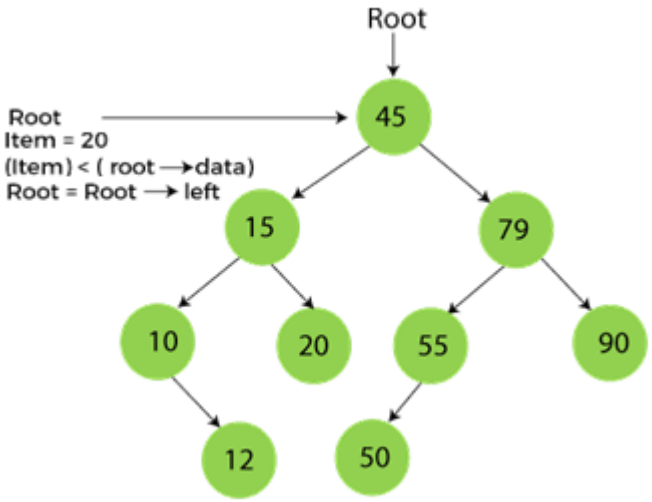
Searching in Binary search tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

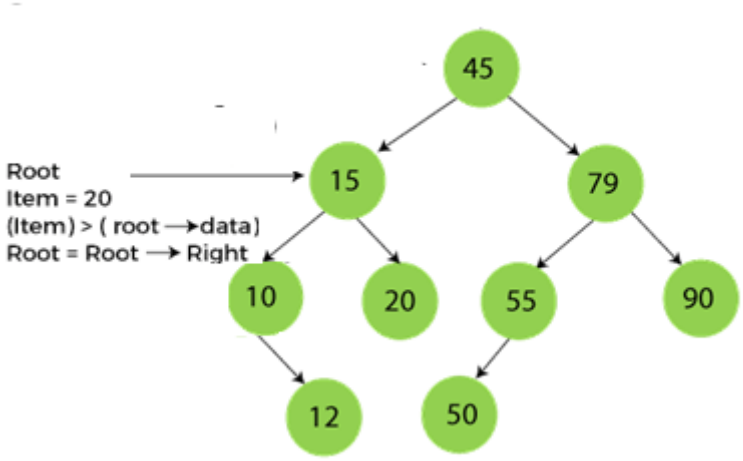
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

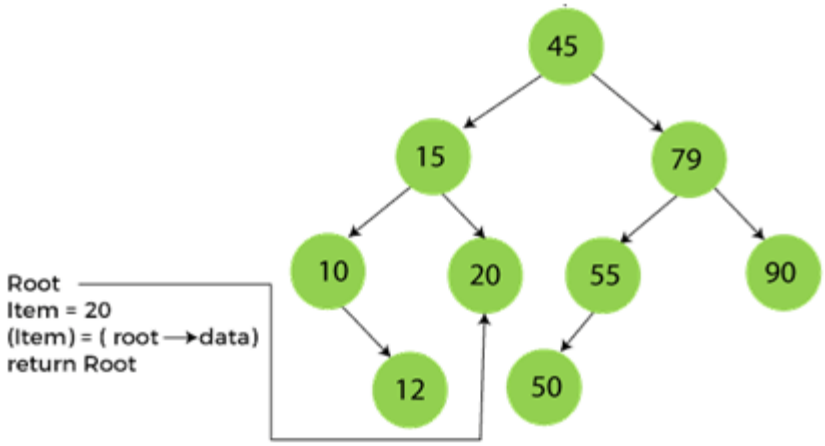
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Algorithm to search an element in Binary search tree

```
Search (root, item)
Step 1 - if (item = root → data) or (root = NULL)
            return root
else if (item < root → data)
        return Search(root → left, item)
else
        return Search(root → right, item)
   END if
Step 2 - END
```

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

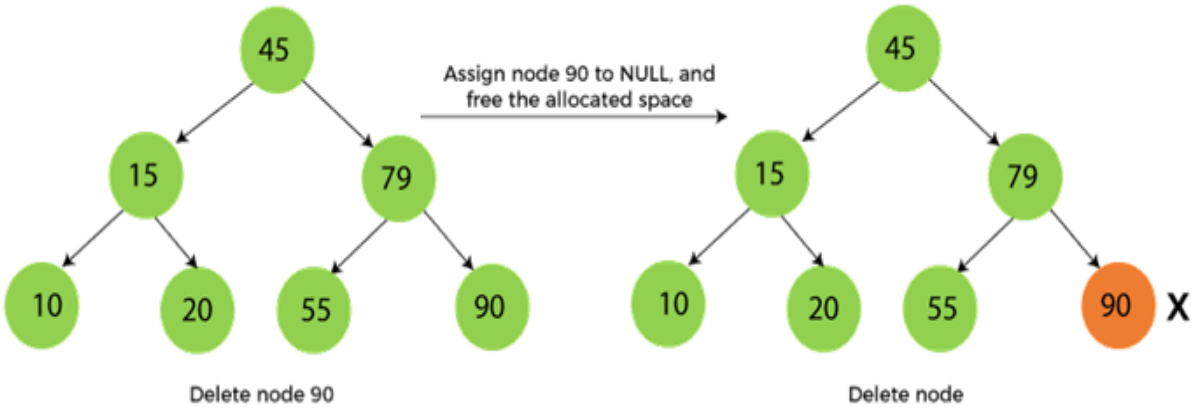
1. The node to be deleted is the leaf node, or,
2. The node to be deleted has only one child, and,
3. The node to be deleted has two children

We will understand the situations listed above in detail.

(1) When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

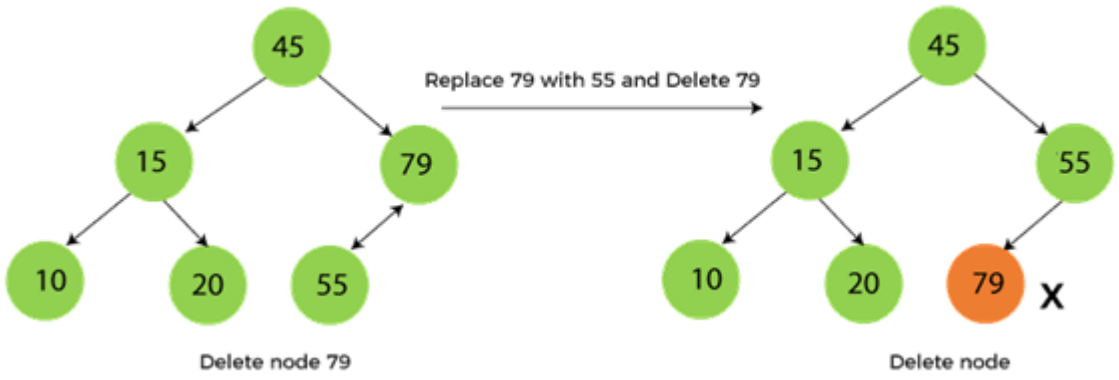


(2)When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



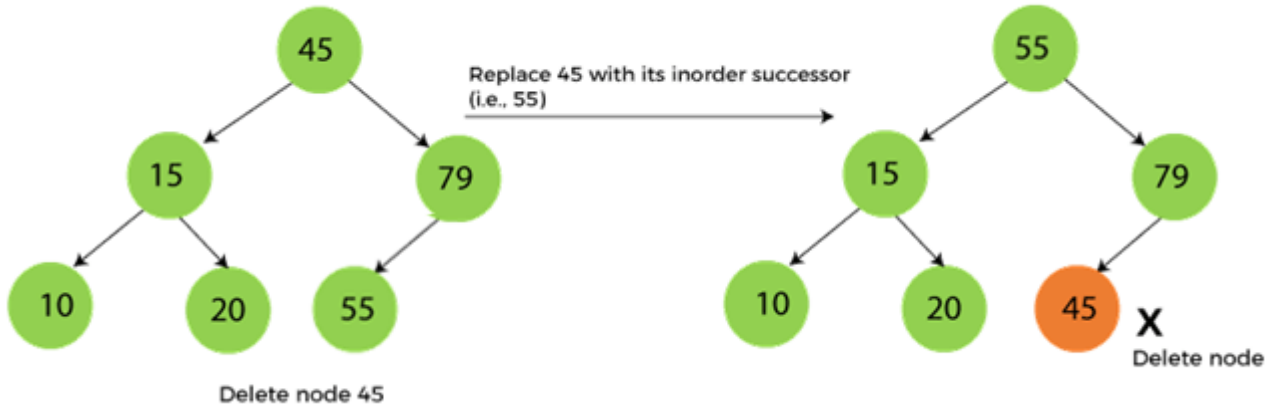
(3)When the node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



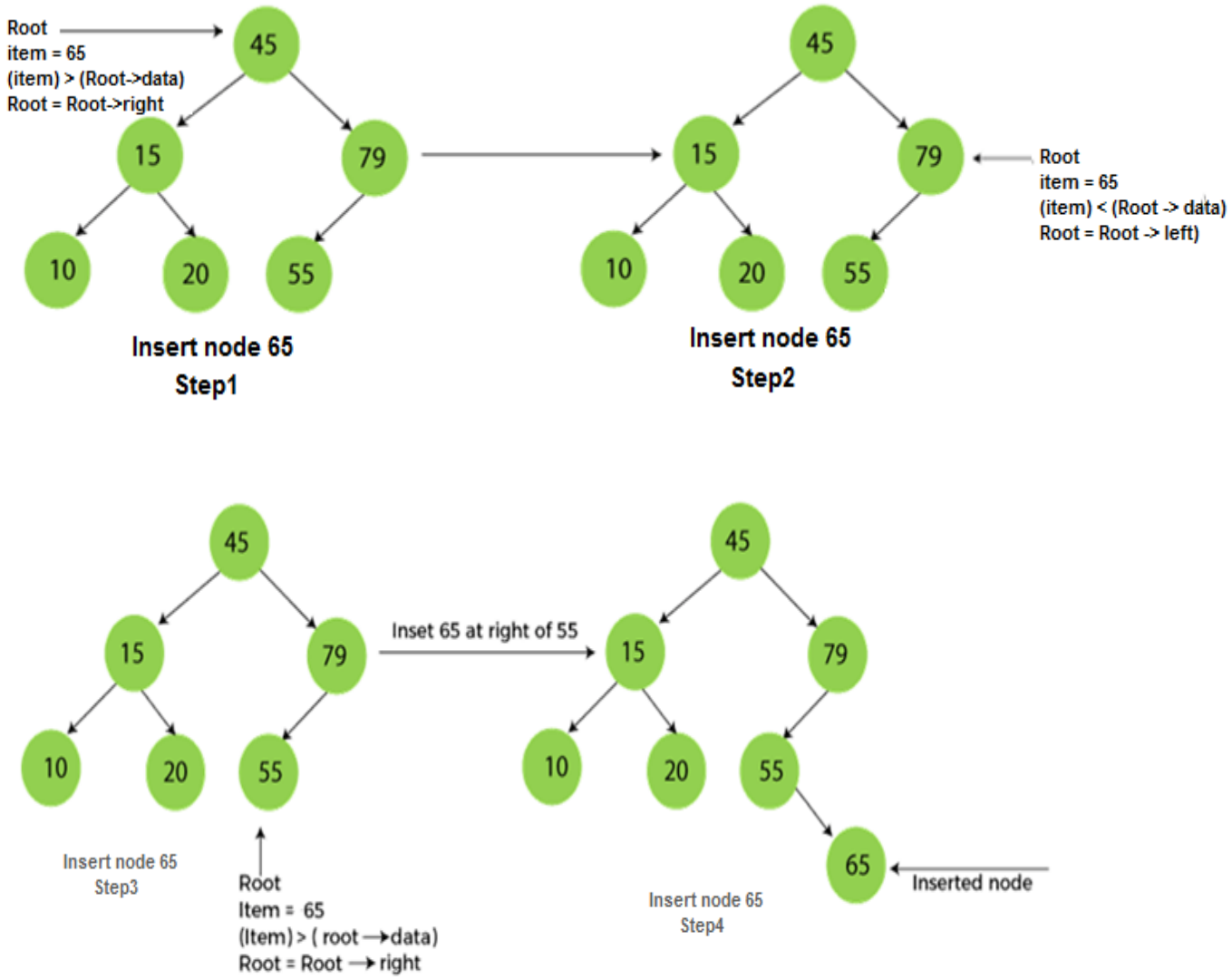
Now let's understand how insertion is performed on a binary search tree.

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the

root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.



The complexity of the Binary Search tree

Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity

Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Where 'n' is the number of nodes in the given tree.

Search Operation

Algorithm:

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

Insert Operation

Algorithm:

```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left  = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;
```

Deletion Operation

```
# Binary Search Tree operations in Python


# Create a node
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


# Inorder traversal
```

```
def inorder(root):
    if root is not None:
        # Traverse left
        inorder(root.left)

        # Traverse root
        print(str(root.key) + "->", end=' ')

        # Traverse right
        inorder(root.right)


# Insert a node
def insert(node, key):

    # Return a new node if the tree is empty
    if node is None:
        return Node(key)

    # Traverse to the right place and insert the node
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)

    return node


# Find the inorder successor
def minValueNode(node):
    current = node

    # Find the leftmost leaf
    while(current.left is not None):
        current = current.left

    return current


# Deleting a node
def deleteNode(root, key):

    # Return if the tree is empty
    if root is None:
        return root

    # Find the node to be deleted
```

```
    if key < root.key:
        root.left = deleteNode(root.left, key)
    elif(key > root.key):
        root.right = deleteNode(root.right, key)
    else:
        # If the node is with only one child or no child
        if root.left is None:
            temp = root.right
            root = None
            return temp

        elif root.right is None:
            temp = root.left
            root = None
            return temp

        # If the node has two children,
        # place the inorder successor in position of the node to be deleted
        temp = minValueNode(root.right)

        root.key = temp.key

        # Delete the inorder successor
        root.right = deleteNode(root.right, temp.key)

    return root


root = None
root = insert(root, 8)
root = insert(root, 3)
root = insert(root, 1)
root = insert(root, 6)
root = insert(root, 7)
root = insert(root, 10)
root = insert(root, 14)
root = insert(root, 4)

print("Inorder traversal: ", end=' ')
inorder(root)

print("\nDelete 10")
root = deleteNode(root, 10)
print("Inorder traversal: ", end=' ')
inorder(root)
```

AVL Tree Datastructure

Introduction:

To control the height of the binary search tree by not letting it to be skewed the developers introduced "AVL Trees". The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

- 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 search 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...

Def: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...

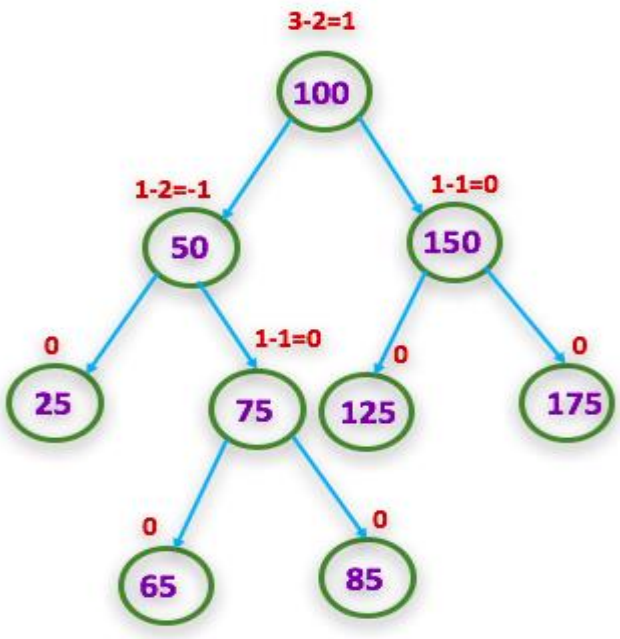
Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Note:

- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

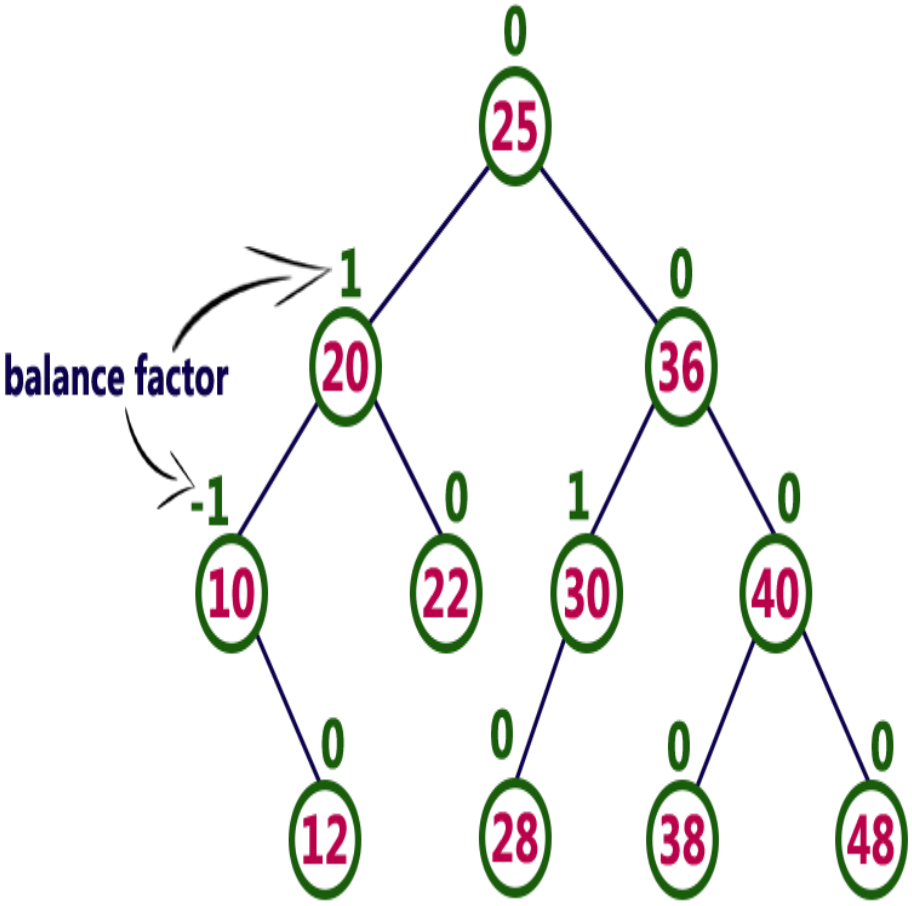
Example of AVL Tree

Example-1



AVL Tree

Example-2



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.

Note:Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)

Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

- Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree.
- Searching and traversing do not lead to the violation in property of AVL tree.
- However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL Tree Rotations

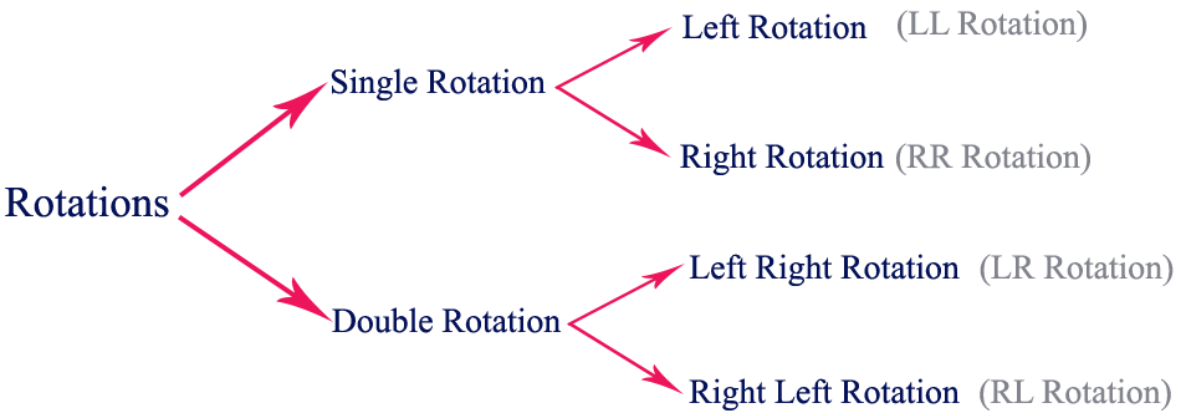
- In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.

- 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.

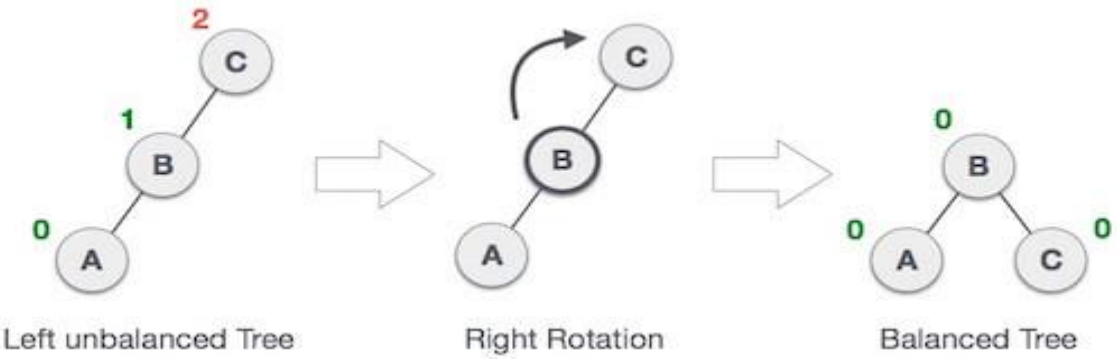
Def: Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are **four** rotations and they are classified into **two** types.



Single Left Rotation (LL Rotation)

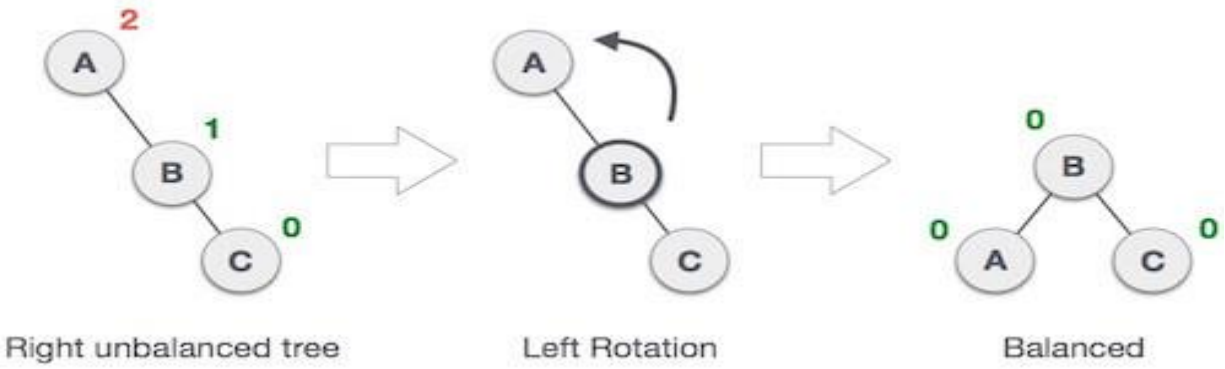
When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

Single Right Rotation (RR Rotation)

When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2

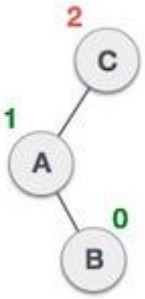


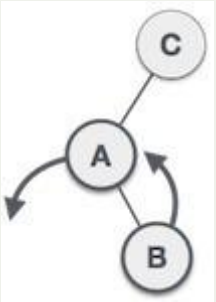
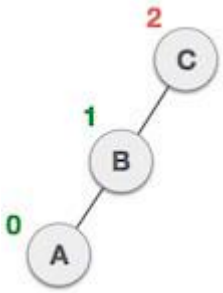
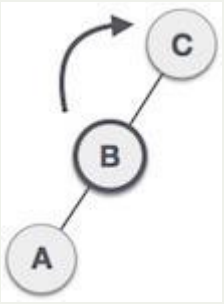
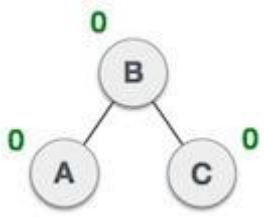
In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

Left Right Rotation (LR Rotation)

LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

Let us understand each and every step very clearly:

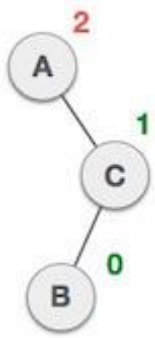
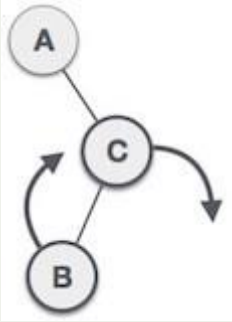
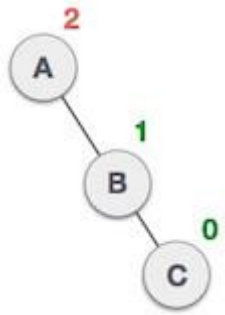
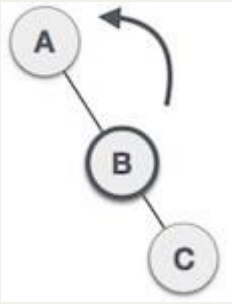
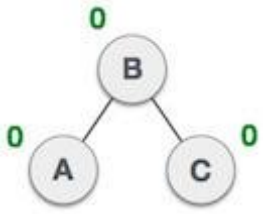
State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C

	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

Right Left Rotation (RL Rotation)

R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action

	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

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...

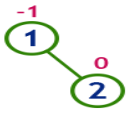
- **Step 1 -** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2 -** After insertion, check the **Balance Factor** of every node.
- **Step 3 -** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4 -** If the **Balance 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 1: Construct an AVL Tree by inserting numbers from 1 to 8.

insert 1

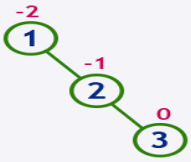
Tree is balanced

insert 2

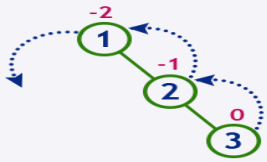


Tree is balanced

insert 3



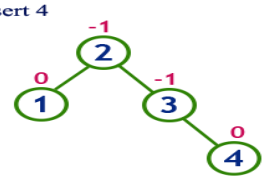
Tree is imbalanced



LL Rotation

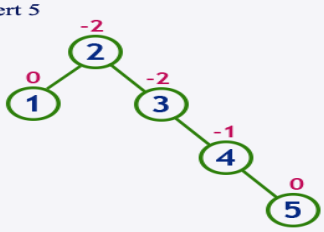
Tree is balanced

insert 4

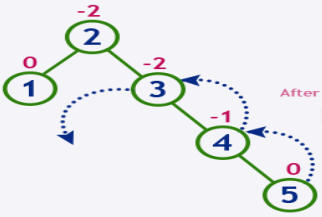


Tree is balanced

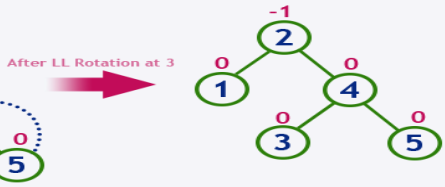
insert 5



Tree is imbalanced

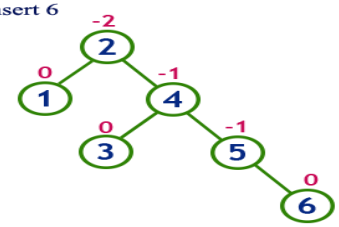


LL Rotation at 3

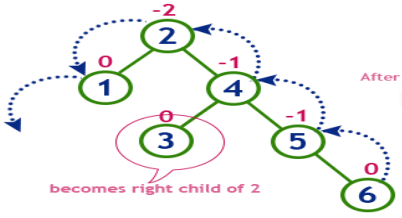


Tree is balanced

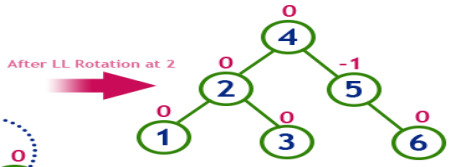
insert 6



Tree is imbalanced

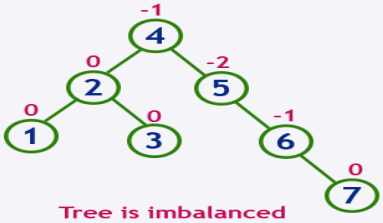


LL Rotation at 2

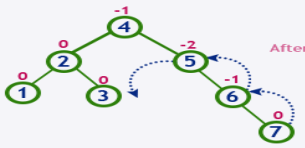


Tree is balanced

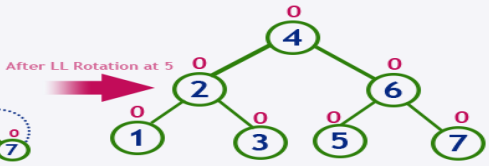
insert 7



Tree is imbalanced

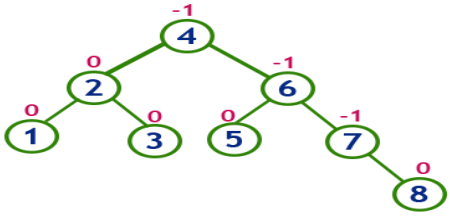


LL Rotation at 5



Tree is balanced

insert 8



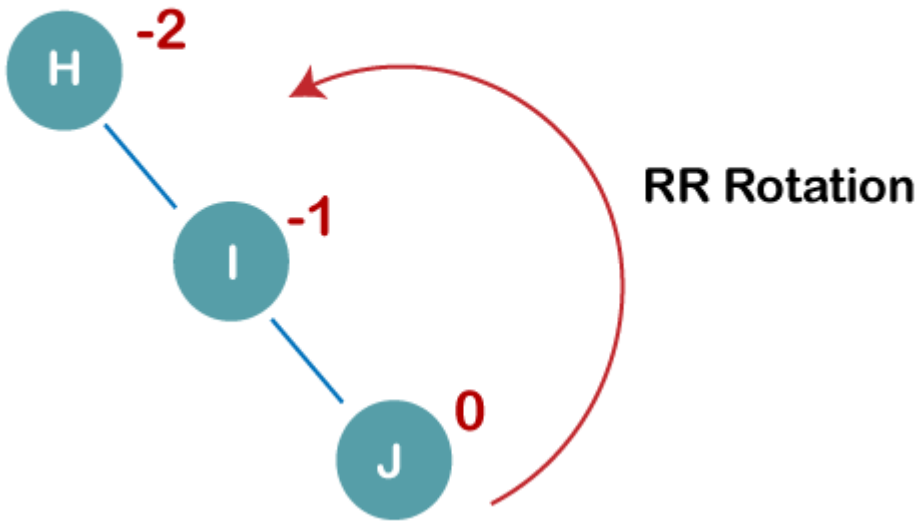
Tree is balanced

Example 2:

Q: Construct an AVL tree having the following elements

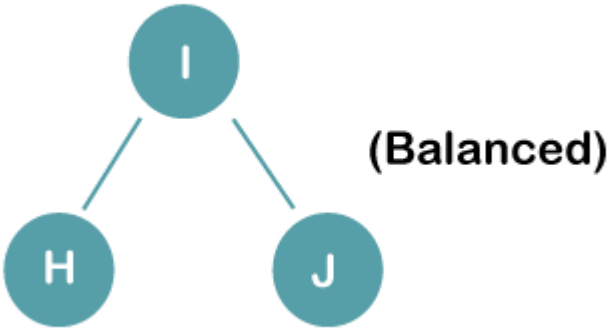
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

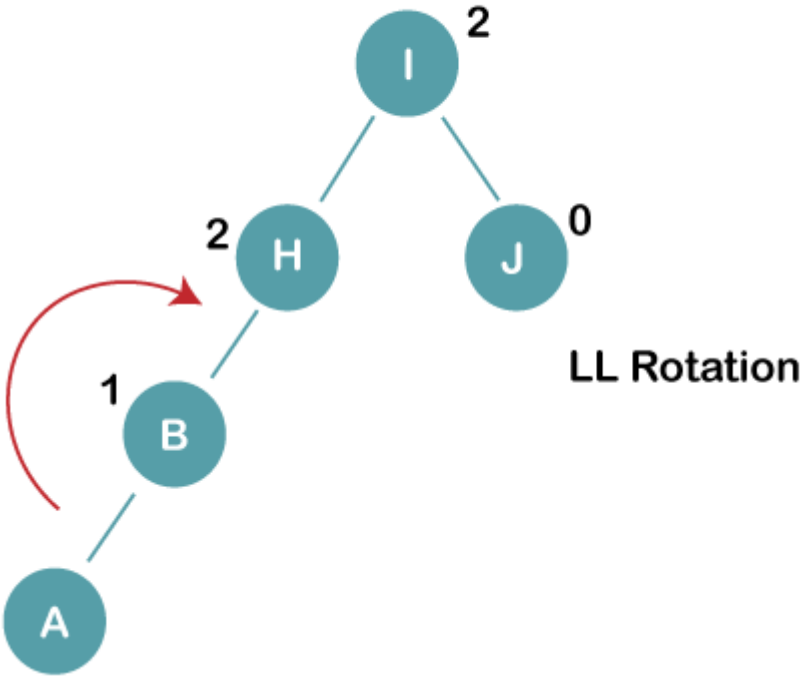


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

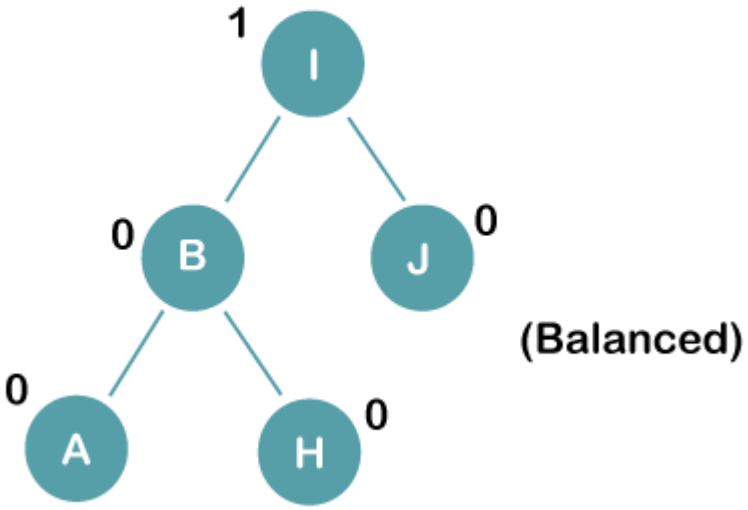


2. Insert B, A

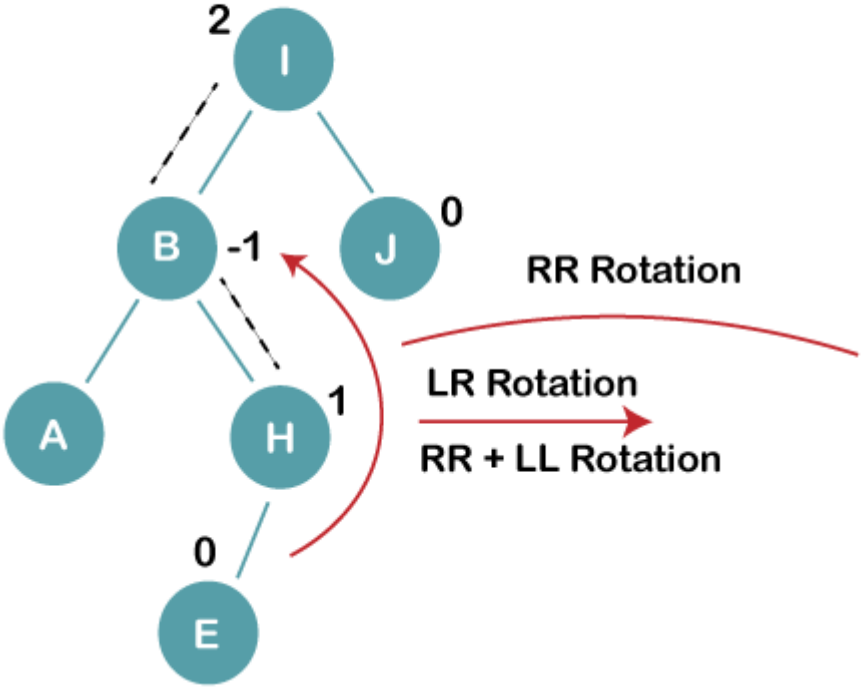


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



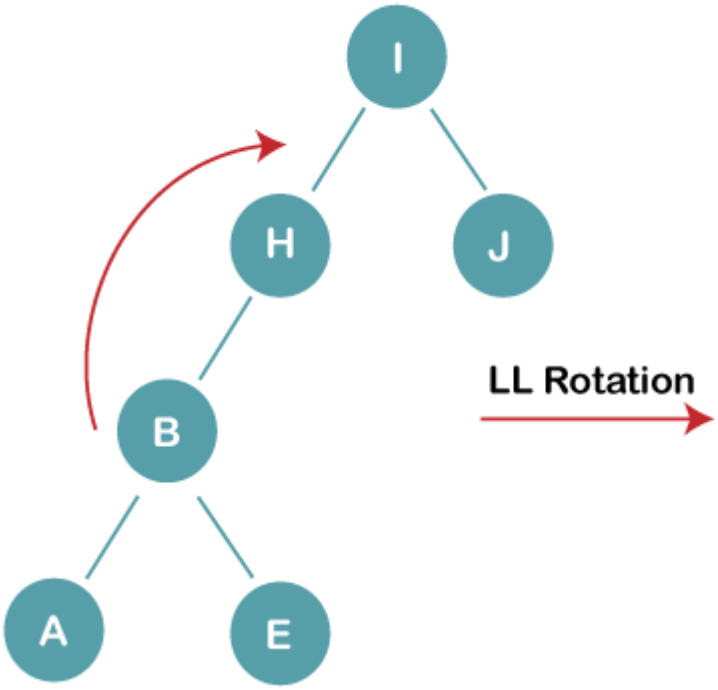
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

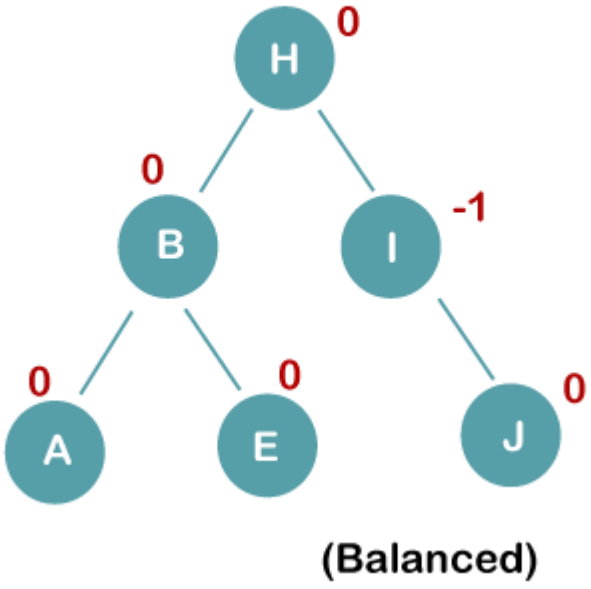
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

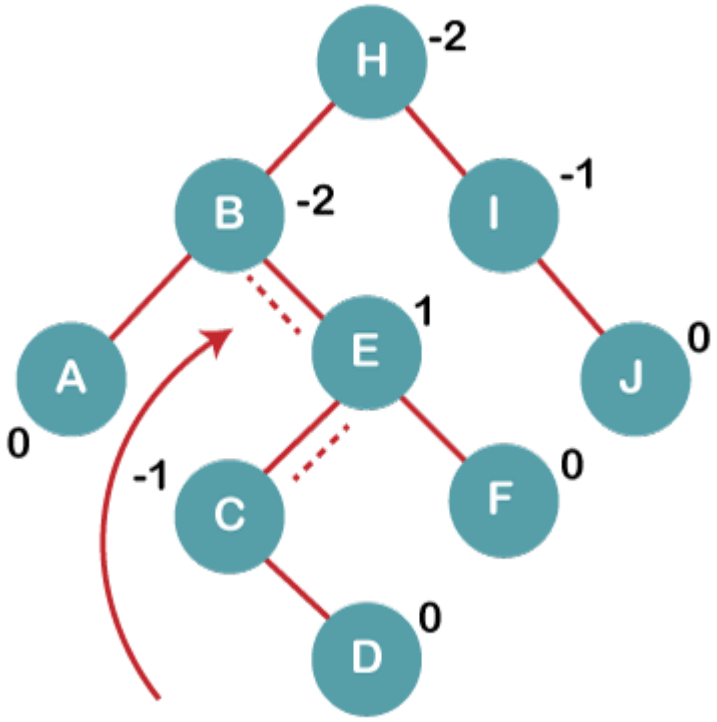


3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



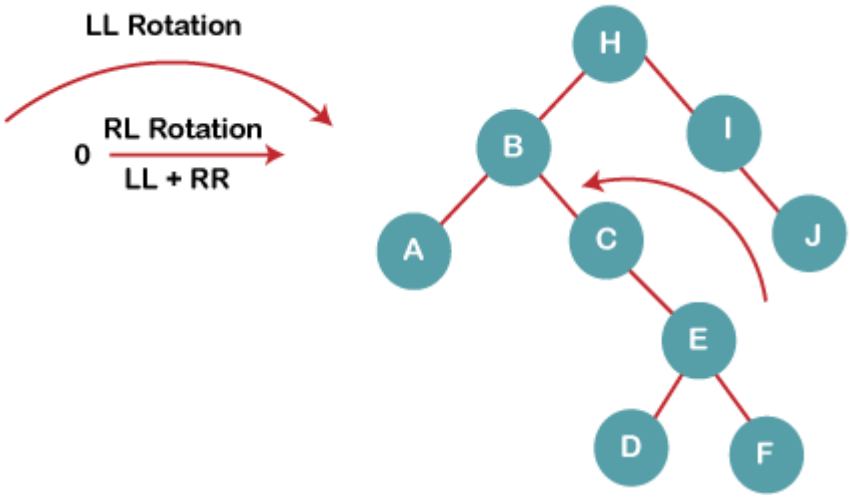
4. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

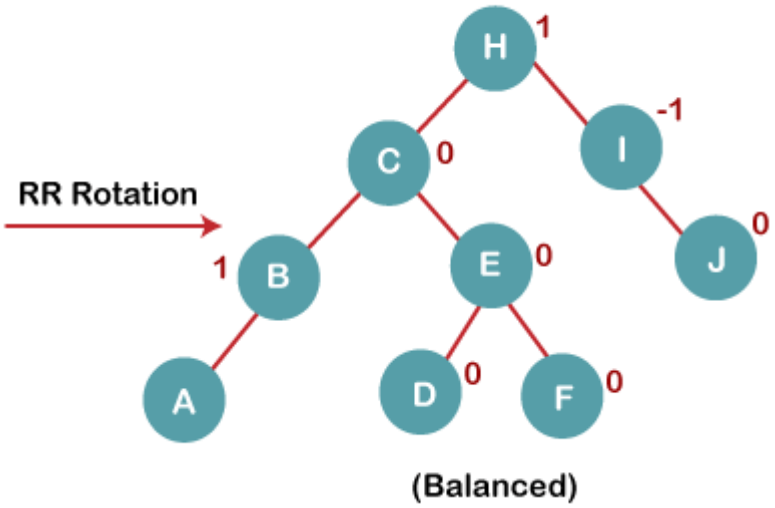
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

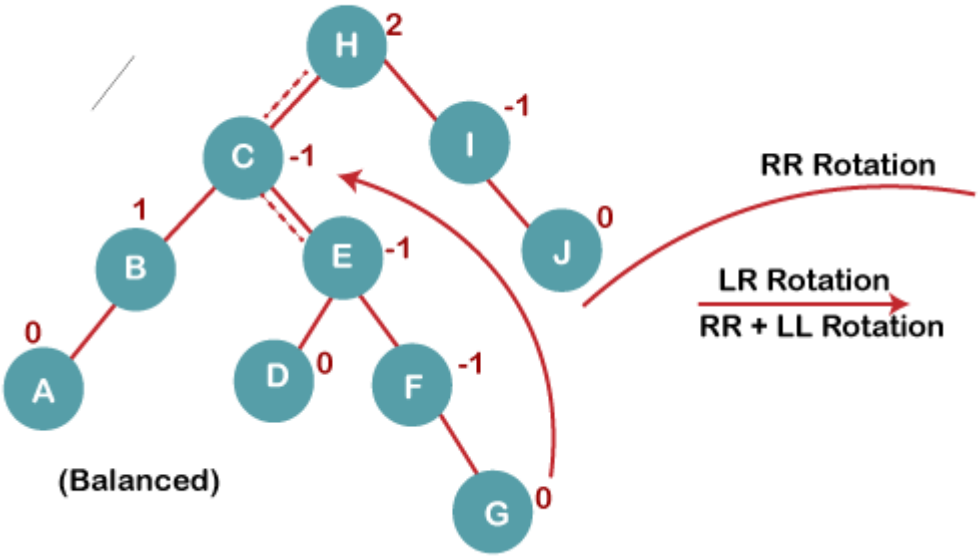


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



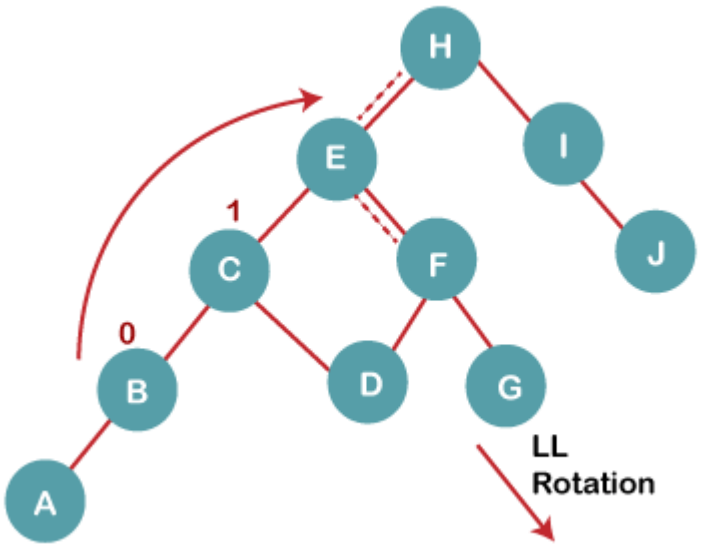
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

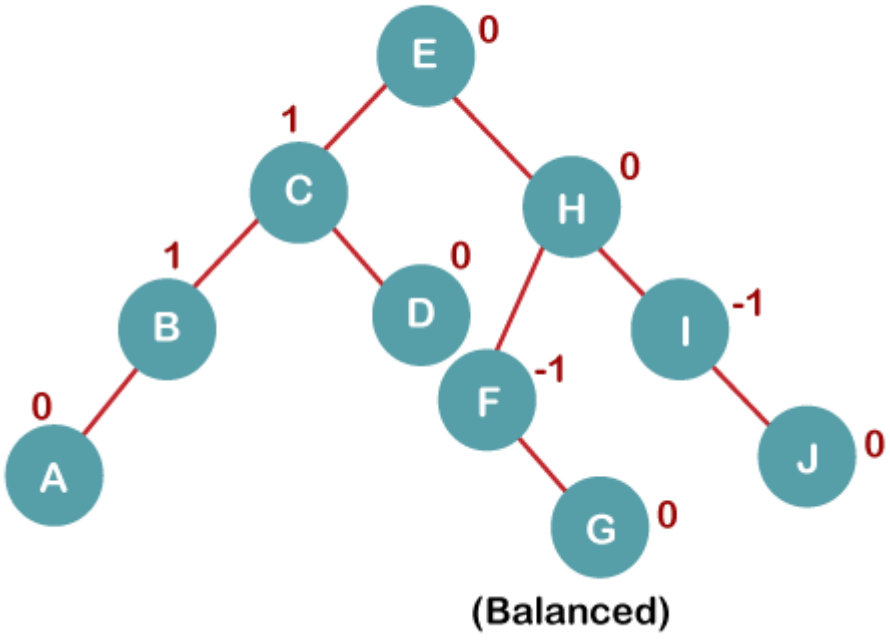
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

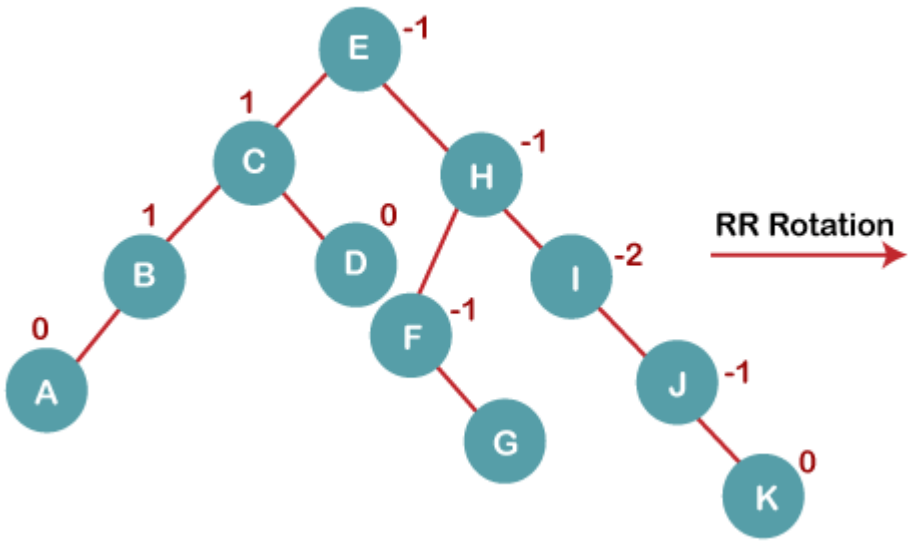


5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

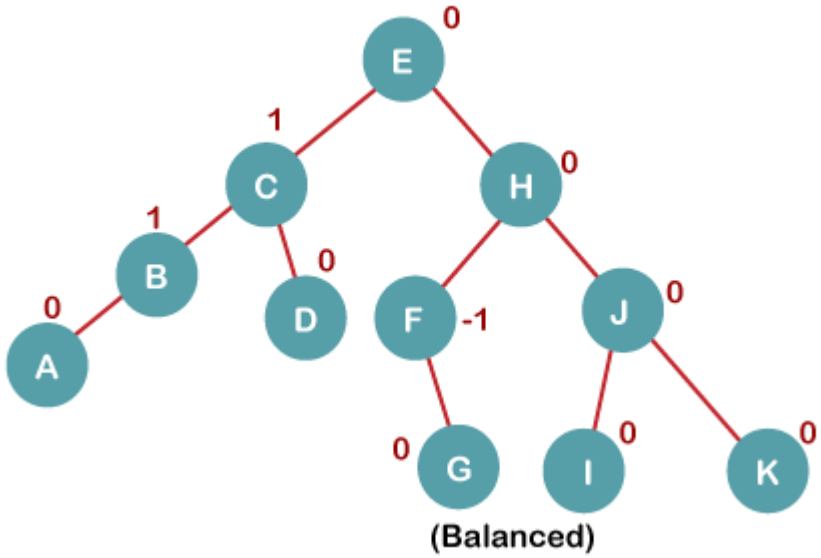


6. Insert K



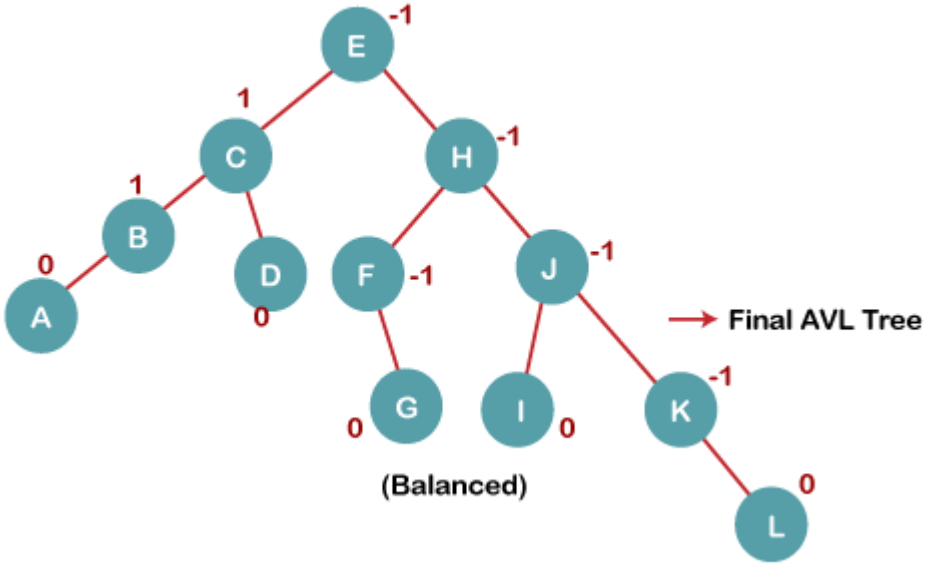
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



Deletion Operation in AVL Tree

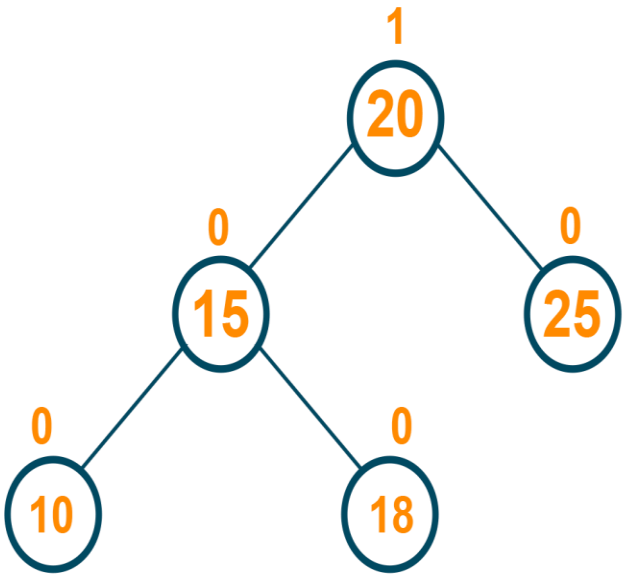
The deletion operation in the AVL tree is the same as the deletion operation in BST. In the AVL tree, the node is always deleted as a leaf node and after the deletion of the node, the balance factor of each node is modified accordingly. Rotation operations are used to modify the balance factor of each node.

The algorithm steps of deletion operation in an AVL tree are:

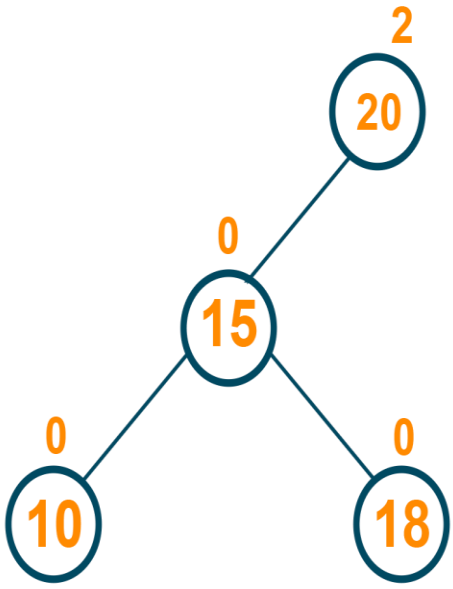
1. Locate the node to be deleted
2. If the node does not have any child, then remove the node
3. If the node has one child node, replace the content of the deletion node with the child node and remove the node
4. If the node has two children nodes, find the inorder successor node ‘k' which has no child node and replace the contents of the deletion node with the ‘k’ followed by removing the node.
5. Update the balance factor of the AVL tree

Example:

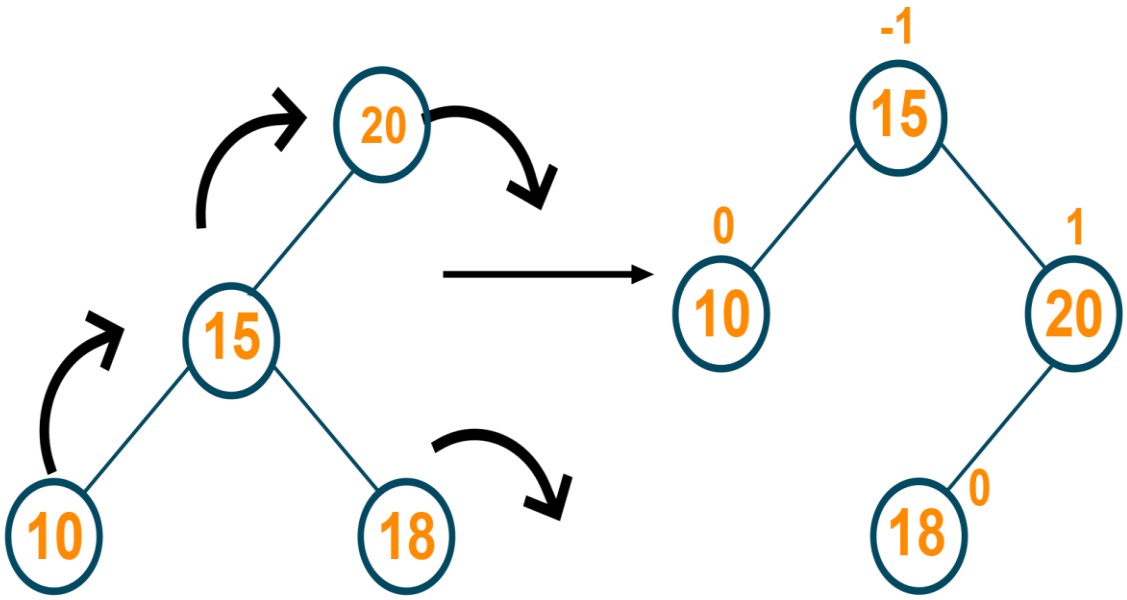
Let us consider the below AVL tree with the given balance factor as shown in the figure below



Here, we have to delete the node '25' from the tree. As the node to be deleted does not have any child node, we will simply remove the node from the tree



After removal of the tree, the balance factor of the tree is changed and therefore, the rotation is performed to restore the balance factor of the tree and create the perfectly balanced tree



Example:

```
class treeNode(object):
        def __init__(self, value):
                self.value = value
                self.l = None
                self.r = None
                self.h = 1

class AVLTree(object):

        def insert(self, root, key):

                if not root:
                        return treeNode(key)
                elif key < root.value:
                        root.l = self.insert(root.l, key)
                else:
                        root.r = self.insert(root.r, key)

                root.h = 1 + max(self.getHeight(root.l),self.getHeight(root.r))

                b = self.getBal(root)

                if b > 1 and key < root.l.value:
                        return self.rRotate(root)

                if b < -1 and key > root.r.value:
                        return self.lRotate(root)
```

```
        if b > 1 and key > root.l.value:
                root.l = self.lRotate(root.l)
                return self.rRotate(root)

        if b < -1 and key < root.r.value:
                root.r = self.rRotate(root.r)
                return self.lRotate(root)

        return root

def lRotate(self, z):

        y = z.r
        T2 = y.l

        y.l = z
        z.r = T2

        z.h = 1 + max(self.getHeight(z.l),self.getHeight(z.r))
        y.h = 1 + max(self.getHeight(y.l),self.getHeight(y.r))

        return y

def rRotate(self, z):

        y = z.l
        T3 = y.r

        y.r = z
        z.l = T3

        z.h = 1 + max(self.getHeight(z.l),self.getHeight(z.r))
        y.h = 1 + max(self.getHeight(y.l),self.getHeight(y.r))

        return y

def getHeight(self, root):
        if not root:
                return 0

        return root.h

def getBal(self, root):
        if not root:
                return 0

        return self.getHeight(root.l) - self.getHeight(root.r)
```

```
        def preOrder(self, root):

                if not root:
                        return

                print("{0} ".format(root.value), end="")
                self.preOrder(root.l)
                self.preOrder(root.r)

Tree = AVLTree()
root = None

root = Tree.insert(root, 1)
root = Tree.insert(root, 2)
root = Tree.insert(root, 3)
root = Tree.insert(root, 4)
root = Tree.insert(root, 5)
root = Tree.insert(root, 6)

# Preorder Traversal
print("Preorder traversal of the","constructed AVL tree is")
Tree.preOrder(root)
print()
```

Output

4 2 1 3 5 6

olf