DIGITAL NOTES

ON

DATA STRUCTURES

(R22A0503)

B.TECH II YEAR–I-SEM

(R22 Regulation)

(2024-2025)

Prepared By
Dr. N. Satheesh Kumar
Associate Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALLAREDDYCOLLEGEOFENGINEERING&TECHNOLOGY

(Autonomous Institution– UGC, Govt. of India)

Recognized under2(f) and 12(B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE- Accredited by NBA & NAAC –'A' Grade-ISO9001:2015Certified)

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

SYLLABUS

MALLAREDDY COLLEGE OF ENGINEERING ANDTECHNOLOGY

II Year B.Tech CSE-I SEM	**L**	**T/P/DC**
	3	-/ -/ -3

COURSE OBJECTIVES:

This course will enable students to

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

UNIT – I

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

UNIT – II

Searching- Linear Search and Binary Search.

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

UNIT -III

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

Stacks - Overview of Stack, Implementation of Stack (List), Applications of Stack

Queues: Overview of Queue, Implementation of Queue (List), Applications of Queues, Priority Queues

Linked Lists – Implementation of Singly Linked Lists, Doubly Linked Lists, Circular Linked Lists. Implementation of Stack and Queue using Linked list.

UNIT -IV

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

UNIT -V

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

Binary Search Trees: Introduction, Implementation

AVL Trees: Introduction, Rotations, Implementation, B-Trees and B+ Trees.

TEXTBOOKS:

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

REFERENCE BOOKS:

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

I N D E X

UNIT – I

1. INTRODUCTION:

Python is a high-level programming language that is translated by the python interpreter. As is known, an interpreter works by translating line-by-line and executing. It was developed by Guido-van-rossum in 1990, at the National Research Institute for Mathematics and Computer Science in Netherlands. Python doesn't refer to the snake but was named after the famous British comedy troupe, Monty Python's Flying Circus.

Python is Interpreted: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.

Python is Interactive: we can actually sit at a Python prompt and interact with the interpreter directly to write our programs.

Python is Object-Oriented: Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Application of python used in Search engine. In mission critical projects in Naza, in processing financial transaction at New york stock Exchange.

The following are some of the features of Python:

- Python is an Open Source: It is freely downloadable, from the link "http:// python.org/"
- Python is portable: It runs on different operating systems / platforms3
- Python has automatic memory management
- Python is flexible with both procedural oriented and object oriented programming
- Python is easy to learn, read and maintain
- Python is Extendable. You can add low-level modules to the Python is Interpreted. These modules enable programmers to add to or customize their tools to be more efficient.
- Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- IT supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.
- It supports functional and structured programming methods as well as OOP.

Points to Remember:

- Case sensitive : Example - In case of print statement use only lower case and not upper case
- Punctuation is not required at end of the statement
- In case of string use single or double quotes i.e. ‘ ’ or “ ”
- Must use proper indentation.
- The following are used Without Indentation
    - Special characters like (,),# etc. are used
    - () ->Used in opening and closing parameters of functions
    - #-> The Pound sign is used to comment a line

2. OOPS CONCEPTS

An object-oriented paradigm is to design the program using classes and objects. The object is related to real-word entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. It is a widespread technique to solve the problem by creating objects. The main concept of object-oriented Programming (OOPs) or oops concepts in Python is to bind the data and the functions that work together as a single unit so that no other part of the code can access this data which includes Class, Objects, Polymorphism, Encapsulation, Inheritance and Data Abstraction.

I. CLASS:

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
- Syntax:

```
class ClassName:
# Statement-1
.
.
# Statement-N
```

II. **OBJECT:**

In object-oriented programming Python, The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

All classes have a function called __init__(), which is always executed when the class is being initiated. Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

Note: The __init__() function is called automatically every time the class is being used to create a new object.

III. **METHOD:**

The method is a function that is associated with an object. In Python, a method is not unique to class instances. Any object type can have methods.

IV. INHERITANCE:

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object. By using inheritance, we can create a class which uses all the properties and behavior of another class.

The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

V. POLYMORPHISM:

Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape. By polymorphism, we understand that one task can be performed in different ways. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

VI. ENCAPSULATION:

It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables. A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

VII. DATA ABSTRACTION:

It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved by creating abstract classes.

3. STRUCTURE OF PYTHON PROGRAM

i. **Comments:** Comments are used to explain the purpose of the code or to make notes for other programmers. They start with a '#' symbol and are ignored by the interpreter.

ii. **Import Statements:** Import statements are used to import modules or libraries into the program. These modules contain predefined functions that can be used to accomplish tasks.

iii. **Variables:** Variables are used to store data in memory for later use. In Python, variables do not need to be declared with a specific type.

iv. **Data Types:** Python supports several built-in data types including integers, floats, strings, booleans, and lists.

v. **Operators:** Operators are used to perform operations on variables and data. Python supports arithmetic, comparison, and logical operators.

vi. **Control Structures:** Control structures are used to control the flow of a program. Python supports if-else statements, for loops, and while loops.

vii. **Functions:** Functions are used to group a set of related statements together and give them a name. They can be reused throughout a program.

viii. **Classes:** Classes are used to define objects that have specific attributes and methods. They are used to create more complex data structures and encapsulate code.

ix. **Exceptions:** Exceptions are used to handle errors that may occur during the execution of a program.

Ex: # This program prints Hello, world!
print('Hello, world!')

Example 2:
 # This program adds two numbers
num1 = 1.5

```
num2 = 6.3
# Add two numbers
sum = num1 + num2
# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
Output : The sum of 1.5 and 6.3 is 7.8
Example 3:
# Store input numbers
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')
# Add two numbers
sum = float(num1) + float(num2)
# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

Output:

Enter first number: 1.5

Enter second number: 6.3

The sum of 1.5 and 6.3 is 7.8

4. CONSTRUCTORS

A constructor is an instance method in a class, that is automatically called whenever a new object of the class is created. The constructor's role is to assign value to instance variables as soon as the object is declared.

Python uses a special method called __init__() to initialize the instance variables for the object, as soon as it is declared. The "__init__()" method acts as a constructor. It needs a mandatory argument self, which is the reference to the object.

Syntax

```
def __init__(self):     #initialize instance variables
```

The __init__() method as well as any instance method in a class has a mandatory parameter, self. However, you can give any name to the first parameter, not necessarily self.

Ex:

```
class Employee:
        'Common base class for all employees'
                def __init__(self):
                self.name = "Bhavana"
                self.age = 24
e1 = Employee()
```

```
print ("Name: {}".format(e1.name))
print ("age: {}".format(e1.age))
```

Output: *Name: Bhavana , age: 24*

Parameterized Constructor

For the above Employee class, each object we declare will have same value for its instance variables name and age. To declare objects with varying attributes instead of the default, define arguments for the __init__() method. (A method is nothing but a function defined inside a class.)

Ex: The __init__() constructor has two formal arguments. We declare Employee objects with different values –

```
class Employee:
  'Common base class for all employees'
  def __init__(self, name, age):
    self.name = name
    self.age = age
e1 = Employee("Bhavana", 24)
e2 = Employee("Bharat", 25)
print ("Name: {}".format(e1.name))
print ("age: {}".format(e1.age))
print ("Name: {}".format(e2.name))
print ("age: {}".format(e2.age))
```

Output:

```
Name: Bhavana
age: 24
Name: Bharat
age: 25
```

Non-parameterized Constructor

On the opposite spectrum, we have non-parameterized constructors. These don't take any parameters, ensuring that every object has a standardized structure by allocating default values. The highlight of non-parameterized constructors is their simplicity. When an object comes to life, it gets furnished with predefined values, ensuring uniformity and consistency.

5. TYPES OF VARIABLES

Scope refers to the region of a program where a variable is defined and can be accessed. In Python, there are three types of variable scopes: global, local, and nonlocal.

Global Variables - Global variables are defined outside any function or class and can be accessed from anywhere within the program. They have a global scope, meaning they are visible to all functions and classes. Global variables are useful when you want to share data between different parts of your program.

```
Ex:
count = 0
def increment():
    global count
    count += 1
increment()
print(count)
```

Output : 1

Local Variables - Local variables are defined within a function or a block of code and can only be accessed within that specific function or block. They have a local scope, meaning they are only visible within the function or block where they are defined. Local variables are temporary and are destroyed once the function or block of code is executed.

```
Ex:
def calculate_sum(a, b):
    result = a + b
    return result
sum = calculate_sum(5, 10)
print(sum)
```

Output : 15

Nonlocal Variables - Nonlocal variables are used in nested functions, where a function is defined inside another function. They are neither global nor local variables. Nonlocal variables can be accessed and modified by the inner function, as well as the outer function that encloses it.

```
Ex;
def outer_function():
    x = 10
    def inner_function():
        nonlocal x
        x += 5
        print(x)
# Output: 15
```

```
    inner_function()
outer_function()
```

Output : 15

Identifier :

- Identifier is a name given to various programming elements such as a variable, function, class, module or any other object. Following are the rules to create an identifier.
  - The allowed characters are a-z, A-Z, 0-9 and underscore (_)
  - It should begin with an alphabet or underscore
  - It should not be a keyword
  - It is case sensitive
  - No blank spaces are allowed.
  - It can be of any size
  - Valid identifiers examples : si, rate_of_interest, student1, ageStudent
  - Invalid identifier examples : rate of interest, 1student, @age

Keywords

- Keywords are the identifiers which have a specific meaning in python, there are 33 keywords in python. These may vary from version to version.
  - False, None, True, and, as, assert, break, class, continue, def, del, elif, else, except, finally, for, From, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield

Variables and Types

- When we create a program, we often need store values so that it can be used in a program. We use variables to store data which can be manipulated by the computer program.

Every variable:

- Can be of any size
- Have allowed characters, which are a-z, A-Z, 0-9 and underscore (_)
- should begin with an alphabet or underscore
- should not be a keyword

6. TYPES OF METHODS

In Python, there are three different method types: the static method, the class method and the instance method. Each one of them has different characteristics and should be used in different situations.

Static Methods

A static method in Python must be created by decorating it with @staticmethod. This lets Python know that the method should be static. The main characteristic of a static method is that they can be called without instantiating the class. These methods are self-contained, meaning that they can't access any other attribute or call any other method within that class.

You could use a static method when you have a class, but you don't need a specific instance in order to access that method. For example, if you have a class called Math and you have a method called factorial, you probably won't need a specific instance to call that method. So, you could use a static method.

```
class Math:
 @staticmethod
 def factorial(number):
     if number == 0:
        return 1
     else:
        return number * MethodTypes.factorial(number - 1)
   factorial = MethodTypes.factorial(5)
print(factorial)
```

Class Method

Class methods have to be created with the decorator @classmethod, and these methods share a characteristic with the static methods in that they can be called without having an instance of the class. The difference relies on the capability to access other methods and class attributes but no instance attributes.

Instance Methods

This method can only be called if the class has been instantiated. Once an object of that class has been created, the instance method can be called and can access all the attributes of that class through the reserved word self. An instance method is capable of creating, getting and setting new instance attributes and calling other instance, class and static methods.

In simple terms, it comes down to method types. self is used to access an instance method, cls is used to access a class method and nothing is applied when it's a static method.

The difference between the keywords selfand cls reside only in the method type. If the created method is an instance method then the reserved word self to be used, but if the method is a class method then the keyword cls must be used. Finally, if the method is a static method then none of those words will be used. Static methods are self-contained and don't have access to the instance or class variables nor to the instance or class methods.

7. INHERITANCE

The new class inherits from older classes. The new class copies all the older class's functions and attributes without rewriting the syntax in the new classes. These new classes are called derived classes, and old ones are called base classes.

```
Syntax:
class DerivedClass(BaseClass):      # Class definition
```

Derived Class is the class that will inherit from the Base Class. The Base Class is the class that will serve as the parent or superclass.

```
Example:
class Person:
   def __init__(self, name, age):
# Constructor to initialize name and age attributes
      self.name = name
      self.age = age

   def say_hello(self):  # Method to greet and introduce the person
      print(f"Hello, my name is {self.name} and I am {self.age} years old.")
class Student(Person):  # Student class inherits from Person class
   def __init__(self, name, age, grade):
# Constructor to initialize name, age, and grade attributes
      super().__init__(name, age)
# Call the parent class constructor to initialize name and age
      self.grade = grade  # Additional attribute specific to the Student class
   def say_hello(self):  # Override the say_hello method of the parent class
      super().say_hello()
# Call the parent class say_hello method to introduce the student as a person
      print(f"I am a student in grade {self.grade}.")
 # Print additional information specific to the Student class

# Creating an instance of the base class
person = Person("John", 30)
```

```
person.say_hello()

# Creating an instance of the derived class
student = Student("Mary", 18, 12)
student.say_hello()
```

Output:
Hello, my name is John and I am 30 years old.
Hello, my name is Mary and I am 18 years old.
I am a student in grade 12.

a. SINGLE INHERITANCE

Single Inheritance is the simplest form of inheritance where a single child class is derived from a single parent class. Due to its candid nature, it is also known as Simple Inheritance.

```
# python 3 syntax
# single inheritance example

class parent:               # parent class
   def func1(self):
      print("Hello Parent")

class child(parent):
  # child class
   def func2(self):              # we include the parent class
      print("Hello Child")   # as an argument in the child
                        # class

# Driver Code
test = child()             # object created
test.func1()               # parent method called via child object
test.func2()               # child method called
```

Output:
Hello Parent
Hello Child

b. MULTIPLE INHERITANCE

In multiple inheritance, a single child class is inherited from two or more parent classes. It means the child class has access to all the parent classes' methods and attributes.

However, if two parents have the same “named” methods, the child class performs the method of the first parent in order of reference. To better understand which class’s methods shall be executed first, we can use the Method Resolution Order function (mro). It tells the order in which the child class is interpreted to visit the other classes.

```
# python 3 syntax
# multiple inheritance example

class parent1:              # first parent class
  def func1(self):
    print("Hello Parent1")

class parent2:              # second parent class
  def func2(self):
    print("Hello Parent2")

class parent3:              # third parent class
  def func2(self):              # the function name is same as parent2
    print("Hello Parent3")

class child(parent1, parent2, parent3):    # child class
  def func3(self):              # we include the parent classes
    print("Hello Child")      # as an argument comma separated

# Driver Code
test = child()      # object created
test.func1()        # parent1 method called via child
test.func2()        # parent2 method called via child instead of parent3
test.func3()        # child method called

# to find the order of classes visited by the child class, we use __mro__ on the child class
print(child.__mro__)
```

```
Output:
> Hello Parent1
> Hello Parent2
> Hello Child
>(<class '__main__.child'>, <class '__main__.parent1'>, <class '__main
```

c. MULTI-LEVEL INHERITANCE

In multilevel inheritance, we go beyond just a parent-child relation. We introduce grandchildren, great-grandchildren, grandparents, etc. We have seen only two levels of inheritance with a superior parent class/es and a derived class/es, but here we can have multiple levels where the parent class/es itself is derived from another class/es.

```
class grandparent:              # first level
  def func1(self):
    print("Hello Grandparent")
```

```
class parent(grandparent):        # second level
   def func2(self):
      print("Hello Parent")

class child(parent):              # third level
   def func3(self):
      print("Hello Child")


# Driver Code
test = child()                    # object created
test.func1()                      # 3rd level calls 1st level
test.func2()                      # 3rd level calls 2nd level
test.func3()                      # 3rd level calls 3rd level
```

Output:
> Hello Grandparent
> Hello Parent
> Hello Child

d. HIERARCHICAL INHERITANCE

Hierarchical Inheritance is the right opposite of multiple inheritance. It means that there are multiple derived child classes from a single-parent class.

```
# python 3 syntax
# hierarchical inheritance example

class parent:                     # parent class
   def func1(self):
      print("Hello Parent")

class child1(parent):             # first child class
   def func2(self):
      print("Hello Child1")


class child2(parent):             # second child class
   def func3(self):
      print("Hello Child2")


# Driver Code
test1 = child1()                  # objects created
test2 = child2()

test1.func1()                     # child1 calling parent method
```

```
test1.func2()              # child1 calling its own method

test2.func1()              # child2 calling parent method
test2.func3()              # child2 calling its own method
```

Output:
> Hello Parent
> Hello Child1
> Hello Parent
> Hello Child2

e. HYBRID INHERITANCE

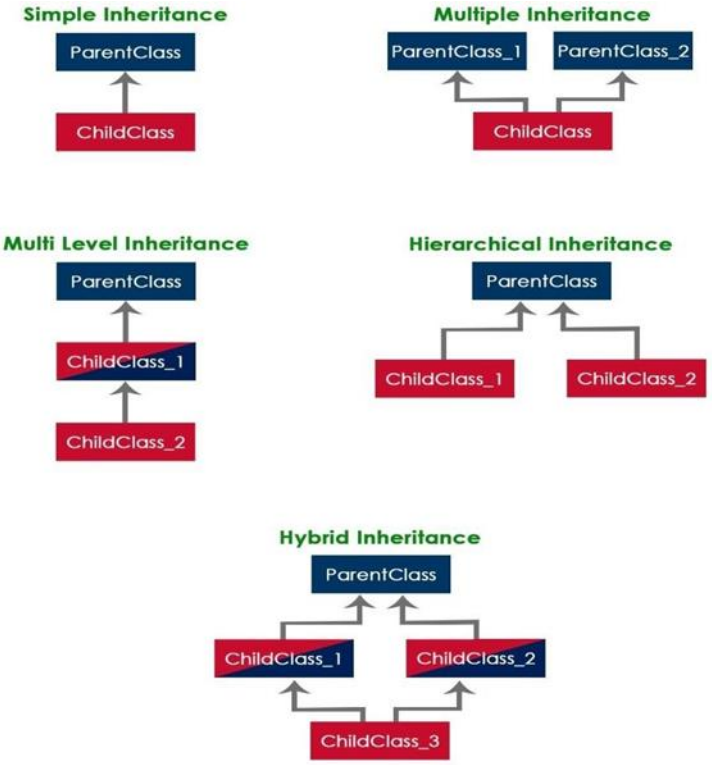
Hybrid Inheritance is the mixture of two or more different types of inheritance. Here we can have many relationships between parent and child classes with multiple levels.

```
# python 3 syntax
# hybrid inheritance example

class parent1:                  # first parent class
  def func1(self):
    print("Hello Parent1")


class parent2:                  # second parent class
  def func2(self):
    print("Hello Parent2")

class child1(parent1):              # first child class
  def func3(self):
    print("Hello Child1")


class child2(child1, parent2):          # second child class
  def func4(self):
    print("Hello Child2")


# Driver Code
test1 = child1()                # object created
test2 = child2()

test1.func1()              # child1 calling parent1 method
test1.func3()              # child1 calling its own method

test2.func1()              # child2 calling parent1 method
test2.func2()              # child2 calling parent2 method
```

```
test2.func3()          # child2 calling child1 method
test2.func4()          # child2 calling its own method
```

Output:

```
> Hello Parent1
> Hello Child1
> Hello Parent1
> Hello Parent2
> Hello Child1
> Hello Child2
```



8. POLYMORPHISM

Polymorphism is defined as the circumstance of occurring in several forms. It refers to the usage of a single type entity (method, operator, or object) to represent several types in various contexts. Polymorphism is made from 2 words – 'poly' and 'morphs.' The word 'poly' means 'many' and 'morphs' means 'many forms.

Polymorphism has the following advantages:

- It is beneficial to reuse the codes.
- The codes are simple to debug.
- A single variable can store multiple data types.

Types of Polymorphism:

- **Compile-Time Polymorphism (Static Binding / Method Overloading):** Compile-time polymorphism is primarily achieved through function overloading, although Python does not support true function overloading. However, you can define functions with the same name in Python, but only the latest defined function will be considered.

```
def add(a, b):
    return a + b

def add(a, b, c):
    return a + b + c

result = add(2, 3)  # Error: Only the latest defined function is available
```

- **Run-Time Polymorphism (Dynamic Binding / Method Overriding):** Run-time polymorphism in Python is typically achieved through method overriding. You can override methods in a subclass to provide specific implementations. The method that gets called is determined at runtime based on the object's actual type.

```
class Animal:
    def make_sound(self):
        print("Animal makes a sound")

class Dog(Animal):
    def make_sound(self):
        print("Dog barks")

my_animal = Dog()
my_animal.make_sound()  # Calls the overridden method in the Dog class
```

- **Interface Polymorphism:** Python follows a concept called "duck typing," which is a form of interface polymorphism. If an object behaves like a particular interface (has the required methods and attributes), it can be treated as an instance of that interface.

```
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        print("Sparrow flies")

class Airplane:
```

```
    def fly(self):
        print("Airplane flies")

def perform_flight(flying_object):
    flying_object.fly()

sparrow = Sparrow()
airplane = Airplane()

perform_flight(sparrow)  # Output: Sparrow flies
perform_flight(airplane)  # Output: Airplane flies
```

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

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

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

a. FUNCTION POLYMORPHISM

There are certain Python functions that can be used with different data types. The len() function is one example of such a function. Python allows it to work with a wide range of data types.

The built-in function len() estimates an object's length based on its type. If an object is a string, it returns the number of characters; or if an object is a list, it returns the number of elements in the list. If the object is a dictionary, it gives the total number of keys found in the dictionary.

Example:

```
mystr = 'Programming'
print('Length of string:', len(mystr))

mylist = [1, 2, 3, 4, 5]
print('Length of list:', len(mylist))
```

```
mydict = {1: 'One', 2: 'Two'}
print('Length of dict:', len(mydict))
```

Output:
Length of string: 11
Length of list: 5
Length of dict: 2

b. CLASS POLYMORPHISM

Because Python allows various classes to have methods with the same name, we can leverage the concept of polymorphism when constructing class methods. We may then generalize calling these methods by not caring about the object we're working with. Then we can write a for loop that iterates through a tuple of items.

Example:

```
class Tiger():
  def nature(self):
    print('I am a Tiger and I am dangerous.')

  def color(self):
    print('Tigers are orange with black strips')

class Elephant():
  def nature(self):
    print('I am an Elephant and I am calm and harmless')

  def color(self):
    print('Elephants are grayish black')

obj1 = Tiger()
obj2 = Elephant()

for animal in (obj1, obj2): # creating a loop to iterate through the obj1 and obj2
  animal.nature()
  animal.color()
```

Output:
I am a Tiger and I am dangerous.
Tigers are orange with black strips
I am an Elephant and I am calm and harmless
Elephants are grayish black

c. POLYMORPHISM WITH INHERITANCE (Method Overrriding)

Polymorphism is most commonly associated with inheritance. In Python, child classes, like other programming languages, inherit methods and attributes from the parent class. Method Overriding is the process of redefining certain methods and attributes to fit the child class. This is especially handy when the method inherited from the parent class does not exactly fit the child class. In such circumstances, the method is re-implemented in the child class. Method Overriding refers to the technique of re-implementing a method in a child class.

Example:

```
class Vehicle:
   def __init__(self, brand, model, price):
      self.brand = brand
      self.model = model
      self.price = price

   def show(self):
      print('Details:', self.brand, self.model, 'Price:', self.price)

   def max_speed(self):
      print('Vehicle max speed is 160')

   def gear_system(self):
      print('Vehicle has 6 shifter gearbox')

# inherit from vehicle class
class Car(Vehicle):
   def max_speed(self):
      print('Car max speed is 260')

   def gear_system(self):
      print('Car has Automatic Transmission')

# Car Object
car = Car('Audi', 'R8', 9000000)
car.show()
# call methods from Car class
car.max_speed()
car.gear_system()

# Vehicle Object
vehicle = Vehicle('Nissan', 'Magnite', 550000)
```

```
vehicle.show()
# call method from a Vehicle class
vehicle.max_speed()
vehicle.gear_system()
```

Output:

Details: Audi R8 Price: 9000000
Car max speed is 260
Car has Automatic Transmission
Details: Nissan Magnite Price: 550000
Vehicle max speed is 160
Vehicle has 6 shifter gearbox

9. ABSTRACTION

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." An abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency

For 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.

10. ABSTRACT CLASSES

An abstraction can be achieved by using abstract classes and interfaces. An abstract class can be considered a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class that contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. We use an abstract class while we are designing large functional units or when we want to provide a common interface for different implementations of a component.

```
Example: # Python program showing abstract base class work
from abc import ABC, abstractmethod
class Polygon(ABC):
    @abstractmethod
    def noofsides(self):
        pass
```

```
class Triangle(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")


class Pentagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 5 sides")


class Hexagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")


class Quadrilateral(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 4 sides")


# Driver code
R = Triangle()
R.noofsides()
K = Quadrilateral()
K.noofsides()
R = Pentagon()
R.noofsides()
K = Hexagon()
K.noofsides()
```

Output

```
I have 3 sides
I have 4 sides
I have 5 sides
I have 6 sides
```

UNIT II

SORTING AND SEARCHING

1. SEARCHING

Search algorithms are methods that allow us to find the location of a specific element within a list of elements. Depending on the list, you will need to use one algorithm or another; for example, if the list has ordered elements, you can use a binary search algorithm, but if the list contains the elements in an unordered way this algorithm will not work. To search for an element in an unordered list you must use a linear search algorithm.

2. LINEAR SEARCH

Linear search algorithms, also known as sequential search, involve going through a list of items one by one until a specific item is found. This algorithm is very simple to implement in code but can be very inefficient depending on the length of the list and the location of the item.

def linear_search(list, objective):

Algorithm:

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

```
    for i in range(len(list)):
        if list[i] == objective:
            return i
    return -1
list = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]
objective_number = 39
result = linear_search(list, objective_number)
if result != -1:
    print(fThe number {objective_number} is located at position: {result}")
else:
    print(f"The number {objective_number} is NOT in the list")
```

code output:
The Number 39 is located at position: 14

Pros:

- Simplicity: Linear search is one of the simplest and easiest search algorithms to implement. It only requires iterating through the list of items one by one until the target is found.
- flexibility: The linear search can be applied to any type of list, regardless of whether it is sorted or not.

Cons:

- Inefficiency in large lists: The main disadvantage of linear search is its inefficiency in large lists. Because it compares each item one by one, its execution time grows linearly with the size of the list.
- Not suitable for ordered lists: Although it can work on unordered lists, linear search is not efficient for ordered lists. In such cases, more efficient search algorithms, such as binary search, are preferable.

3. BINARY SEARCH

The binary search algorithm is a very efficient algorithm that applies only to ordered lists. It works by repeatedly dividing the list into two halves and comparing the target element with the middle element, this significantly reduces the number of comparisons needed.

Algorithm:

In binary search, we follow the following steps:

i. *We start by comparing the element to be searched with the element in the middle of the list/array.*
ii. *If we get a match, we return the index of the middle element.*
iii. *If we do not get a match, we check whether the element to be searched is less or greater than invalue than the middle element.*
iv. *If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.*
v. *If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.*

```
# Example of Binary Search
def binary_search(list, objective, start, end ):
   if start > end:
```

```
        return -1

    center = (start + end) // 2
    if list[center] == objeotive:
        return center
    elif list[center] < objeotive:
        return binary_search(list, objective, center + 1, end)
    else:
        return binary_search(list, objective, start, center - 1)

# Example of use
list = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]
objective = 27
start_search = 0
end_search = len(list) - 1

result = binary_search(list, objective_number, start_search, end_search)

if result != -1:
    print(f"The number {objective_number} is in position: {result}.")
else:
    print(f"The number {objective_number} is NOT in the list")
```

code output:

The number 27 is in position: 12.

Pros:

- Efficiency of ordered lists: The main advantage of binary search is its efficiency on ordered lists. Its execution time is O(log n), which means that it decreases rapidly as the list size increases.
- Fewer comparisons: Compared to linear search, binary search performs fewer comparisons on average, making it faster to find the target.

Cons:

- Requires a sorted list: Binary search is only applicable to sorted lists. If the list is not sorted, an additional operation must be performed to sort the list before using binary search.
- Higher deployment complexity: Compared to linear search, binary search is more complex to implement due to its recursive nature.

4. SORTING

A sorting algorithm allows us to rearrange a list of elements or nodes in a specific order, for example in ascending or descending order depending on the occasion.

a. BUBBLE SORT

The bubble sort algorithm is one of the simplest but least efficient algorithms. It works by comparing pairs of elements and swapping them if they are in the wrong order, this process is done over and over again until the list is sorted correctly. It has a time complexity of O(n2) in the average and worst cases scenarios and O(n) in the best-case scenario.

```
def bubble_sort(list):
    length = len(list)
    for i in range(length):
        for j in range(0, (length-i) - 1):


            if list[j] > list[j + 1]:
                auxiliar = list[j + 1]
                list[j + 1] = list[j]
                list[j] = auxiliar
    return list

unordered_list = [3, 6, 7, 8, 3, 45, 23, 0, 16, 26, 6, 7, 50]
ordered_list = bubble_sort(unordered_list)
print(ordered_list)

                    # output: [0, 3, 3, 6, 6, 7, 7, 8, 16, 23, 26, 45, 50]
```

Pros:

- Simplicity: The bubble algorithm is easy to understand and implement, which makes it a good choice for introducing ordering concepts in programming.
- Simple deployment: Requires little amount of code and does not involve complex data structures.

Cons:

- Slow for large lists: Due to its quadratic complexity the bubble algorithm becomes slow in practice for lists of considerable size.
- Does not consider partial order: Unlike other algorithms, the bubble algorithm performs the same number of comparisons and swaps regardless of whether the list is already largely sorted.

b. SELECTION SORT

This sorting technique repeatedly finds the minimum element and sort it in order. Bubble Sort does not occupy any extra memory space. During the execution of this algorithm, two subarrays are maintained, the subarray which is already sorted, and the remaining subarray which is unsorted. During the execution of Selection Sort for every iteration, the minimum element of the unsorted subarray is arranged in the sorted subarray. Selection Sort is a more efficient algorithm than bubble sort. Sort has a Time-Complexity of O(n2) in the average, worst, and in the best cases.

Algorithm
Step 1: Set MIN to location 0
Step 2: Search the minimum element in the list
Step 3: Swap with value at location MIN
Step 4: Increment MIN to point to next element
Step 5: Repeat until list is sorted

```
# Example of Selection Sort
def selectionSort(array, size):

  for s in range(size):
    min_idx = s

    for i in range(s + 1, size):

      # For sorting in descending order
      # for minimum element in each loop
      if array[i] < array[min_idx]:
        min_idx = i

    # Arranging min at the correct position
    (array[s], array[min_idx]) = (array[min_idx], array[s])

# Driver code
data = [ 7, 2, 1, 6 ]
size = len(data)
selectionSort(data, size)

print('Sorted Array in Ascending Order is :')
print(data)
```

Output
Sorted Array in Ascending Order is : [1, 2, 6, 7]

Advantages

- The main advantage of the selection sort is that it performs well on a small list.
- Because it is an in-place sorting algorithm, no additional temporary storage is required beyond
- what is needed to hold the original list.
- Its performance is easily influenced by the initial ordering of the items before the sorting process.

Disadvantages

- The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items.
- The selection sort requires n-squared number of steps for sorting n elements.
- Quick Sort is much more efficient than selection sort

c. INSERTION SORT

The insertion sort algorithm is a simple but efficient algorithm. It works by dividing the list into two parts, an ordered part and an unordered part. As the unordered list is traversed, elements are inserted in the correct position in the ordered part. Insertion Sort has a Time-Complexity of O(n2) in the average and worst case, and O(n) in the best case.

Algorithm
Step 1: If it is the first element, it is already sorted. return 1;
Step 2: Pick next element
Step 3: Compare with all elements in the sorted sub-list
Step 4: Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5: Insert the value
Step 6: Repeat until list is sorted

```
#Example of Insertion Sort
def insertion_sort(list):
   for i in range(1, len(list)):
      actual = list[i]
      index = i

      """
      This loop interchanges the two position numbers, as long as the previous number is larger than the current number.
      """
```

```
        while index > 0 and list[index - 1] > actual:
            list[index] = list[index - 1]
            index = index - 1
        list[index] = actual

    return list

unordered_list = [39, 45, 32, 4, 2, 85, 43, 7, 18, 16, 5, 67, 32]
ordered_list = insertion_sort(unordered_list)
print(ordered_list)

                    # output: [2, 4, 5, 7, 16, 18, 32, 32, 39, 43, 45, 67, 85]
```

Pros:

- Low overhead: Requires fewer comparisons and moves than algorithms such as bubble sort, which makes it more efficient in terms of item exchanges.
- Simplicity: insertion sort is one of the simplest sorting algorithms to implement and understand. This makes it suitable for teaching basic sorting concepts.

Cons:

- Inefficiency in large lists: As the list size increases, the performance of insertion sort decreases. Its quadratic complexity of O(n^2) in the worst case makes it inefficient for large lists.
- Non-scalable: Like other quadratic complexity algorithms, insertion sort is not scalable for large lists, as its execution time increases considerably with the size of the list.

d. MERGE SORT

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. Time Complexity of merge sort is O(n*log(n))

```
# Python program for implementation of MergeSort

# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]
```

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted
```

```
def mergeSort(arr, l, r):
    if l < r:

        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)


# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print("Given array is")
for i in range(n):
    print("%d" % arr[i],end=" ")

mergeSort(arr, 0, n-1)
print("\n\nSorted array is")
for i in range(n):
    print("%d" % arr[i],end=" ")
```

Output
Given array is 12 11 13 5 6 7
Sorted array is 5 6 7 11 12 13

e. QUICK SORT

Quick sort is a well-known sorting algorithm. It is highly efficient and also known as partition exchange sort. In this sorting algorithm the array is divided into 2 sub array. One contain smaller

values than pivot value and other array contain elements having greater values than pivot value.

Pivot is an element that is used to compare and divide the elements of the main array into two. Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays. This algorithm is quite efficient for large data sets. The Average and worst case

complexity are of this algorithm is O(n2), where n is the number of items.

Algorithm
Step 1: Choose the highest index value has pivot
Step 2: Take two variables to point left and right of the list excluding pivot
Step 3: left points to the low index
Step 4: right points to the high
Step 5: while value at left is less than pivot move right
Step 6: while value at right is greater than pivot move left
Step 7: if both step 5 and step 6 does not match swap left and right
Step 8: if left ≥ right, the point where they met is new pivot

Advantages

- The quick sort is regarded as the best sorting algorithm.
- It is able to deal well with a huge list of items.
- Because it sorts in place, no additional storage is required as well

Disadvantages

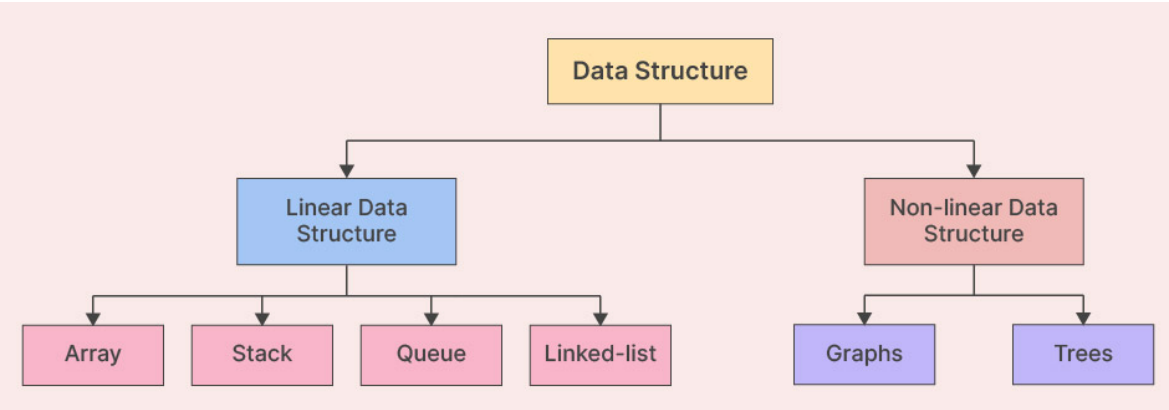
- The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble, insertion or selections sorts.
- If the list is already sorted than bubble sort is much more efficient than quick sort
- If the sorting element is integers than radix sort is more efficient than quick sort.

UNIT – III
DATA STRUCTURES

1. DEFINITION

A data structure refers to organizing and storing data in a computer's memory in a way that enables efficient access, manipulation, and retrieval of the data. Data structures are fundamental concepts in computer science and are extensively used in programming and software development to solve various computational problems. Data structures are essential for organizing and storing data in a manner that allows users to utilize it effectively.

There are two categories of data structure - linear data structure and non-linear data structure. In real life, linear data structure is used to develop software, and non-linear data structure is used in image processing and artificial intelligence.



2. LINEAR DATA STRUCTURES

A linear data structure is a type of data structure that stores the data linearly or sequentially. In the linear data structure, data is arranged in such a way that one element is adjacent to its previous and the next element. It includes the data at a single level such that we can traverse all data into a single run.

The implementation of the linear data structure is always easy as it stores the data linearly. The common examples of linear data types are Stack, Queue, Array, LinkedList, and Hashmap, etc.

3. NON-LINEAR DATA STRUCTURES

In a non-linear data structure, data is connected to its previous, next, and more elements like a complex structure. In simple terms, data is not organized sequentially in such a type of data structure.

It is one type of non-primitive data structure. In non-linear data structures, data is not stored in

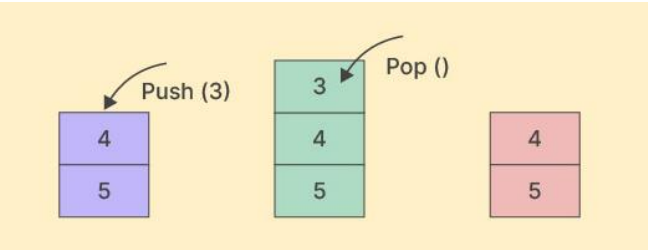
a linear manner. There are multiple levels of nonlinear data structures. It is also called a multilevel data structure. Implementing a non-linear data structure can be more challenging compared to its linear counterpart. However, it offers more efficient memory utilization. Examples of non-linear data structures include trees and graphs.

Factor	**Linear Data Structure**	**Non-Linear Data Structure**
Data Element Arrangement	In a linear data structure, data elements are sequentially connected, allowing users to traverse all elements in one run.	In a non-linear data structure, data elements are hierarchically connected, appearing on multiple levels.
Implementation Complexity	Linear data structures are relatively easier to implement.	Non-linear data structures require a higher level of understanding and are more complex to implement.
Levels	All data elements in a linear data structure exist on a single level.	Data elements in a non-linear data structure span multiple levels.
Traversal	A linear data structure can be traversed in a single run.	Traversing a non-linear data structure is more complex, requiring multiple runs.
Memory Utilization	Linear data structures do not efficiently utilize memory.	Non-linear data structures are more memory friendly.
Time Complexity	The time complexity of a linear data structure is directly proportional to its size, increasing as input size increases.	The time complexity of a non-linear data structure often remains constant, irrespective of its input size.
Applications	Linear data structures are ideal for application software development.	Non-linear data structures are commonly used in image processing and Artificial Intelligence.
Examples	Linked List, Queue, Stack, Array.	Tree, Graph, Hash Map.

4. STACKS - OVERVIEW OF STACK

Can push/pop data from a single end of the stack. Users can insert the data into the stack via push operation and remove data from the stack via pop operation. The stack follows the rule of LIFO (last in first out). Users can access all the stack data from the top of the stack in a linear manner. In real-life problems, the stack data structure is used in many applications. For

example, the web browsers use the stack to store the backward/forward operations. For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1)



There are some basic operations that allow us to perform different actions on a stack.

- Push: Add an element to the top of a stack
- Pop: Remove an element from the top of a stack
- IsEmpty: Check if the stack is empty
- IsFull: Check if the stack is full
- Peek: Get the value of the top element without removing it

a. IMPLEMENTATION OF STACK (LIST)

The operations work as follows:

- A pointer called TOP is used to keep track of the top element in the stack.
- When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
- On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
- On popping an element, we return the element pointed to by TOP and reduce its value.
- Before pushing, we check if the stack is already full
- Before popping, we check if the stack is already empty

Python's built-in list data structure is a simple and effective way to implement a stack. The key operations in a stack, namely 'push' (add an element to the top of the stack) and 'pop' (remove an element from the top of the stack), can be easily achieved using the append() and pop() methods of a list.

- To add an element to the stack (push operation), you can use the append() method. This method adds an element to the end of the list, which corresponds to the top of the

stack in our analogy.

```
Ex:
    stack = []
   stack.append('a')
   stack.append('b')
  stack.append('c')
  print(stack)

                    # Output:  ['a', 'b', 'c']
```

- To remove an element from the stack (pop operation), you can use the pop() method. This method removes the last element from the list, which corresponds to the top of the stack.

```
last_element = stack.pop()
print(f'Popped Element: {last_element}')
print(f'Stack after Pop: {stack}')

                    # Output:
                          # Popped Element: c
                         # Stack after Pop: ['a', 'b']



# Stack implementation using LIst


# Creating a stack
def create_stack():
   stack = []
   return stack

 # Creating an empty stack
def check_empty(stack):
   return len(stack) == 0

 # Adding items into the stack
def push(stack, item):
   stack.append(item)
   print("pushed item: " + item)

 # Removing an element from the stack
def pop(stack):
   if (check_empty(stack)):
      return "stack is empty"
```

```
    return stack.pop()

stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

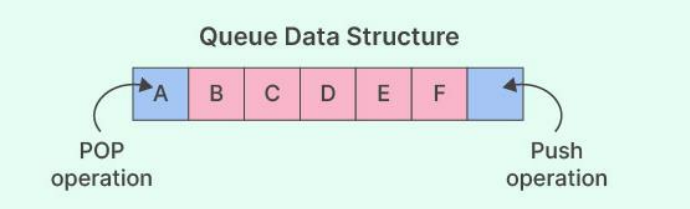
b. APPLICATIONS OF STACK

Applications of Stack Data Structure

- To reverse a word - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
- In compilers - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.
- In browsers - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.
- Backtracking Algorithms - Backtracking is a method of solving problems where you move forward only if there is no obstacle in the solution path, and if you encounter an obstacle, you move backwards and try another path. Stacks are used extensively in backtracking problems to keep track of the visited paths and to backtrack when no solution is found on the current path.
- Parsing - In compilers, stacks are used in the parsing phase to check the correctness of expressions and their syntax. For example, they can be used to check whether the opening and closing parentheses in an expression are balanced.
- Memory Management - Stacks play a vital role in memory management. When a function is called, the memory for its variables is allocated on the stack, and when the function returns, this memory is freed up.

5. QUEUES: OVERVIEW OF QUEUE

Queue data structure stores the data in a linear sequence. Queue data structure follows the FIFO rule, which means first-in-first-out. It is similar to the stack data structure, but it has two ends. In the queue, we can perform insertion operations from the rear using the Enqueue method and deletion operations from the front using the deque method. Escalator is one of the best real-life examples of the queue.



The primary queue operations are as follows:

- Enqueue: It adds an element to the end of the queue. When the queue reaches its total capacity, it reaches an overflow condition. The time complexity of enqueueing is O:1.
- Dequeue: This operation removes an element from the queue. Since it bases the queue on a FIFO manner, it releases the items in the order of their additions. When the queue becomes empty, it reaches an underflow condition. The time complexity is O:1.
- Front: It gives you the first item from the queue. The time complexity is O:1.
- Rare: It gives you the last item from the queue. The time complexity is O:1.

a. IMPLEMENTATION OF QUEUE (LIST)

Python list is used as a way of implementing queues. The list's append() and pop() methods can insert and delete elements from the queue. However, while using this method, shift all the other elements of the list by one to maintain the FIFO manner. This results in requiring O(n) time complexity.

Example:

```
# Initialize a queue
queue_exm = []
# Adding elements to the queue
queue_exm.append('x')
queue_exm.append('y')
queue_exm.append('z')
print("Queue before any operations")
print(queue_exm)
```

```
# Removing elements from the queue
print("\nDequeuing items")
print(queue_exm.pop(0))
print(queue_exm.pop(0))
print(queue_exm.pop(0))
print("\nQueue after deque operations")
print(queue_exm)
```

Output:
Queue before any operation ['x' , 'y', 'z']
Dequeue items x , y, z

Adding Elements to Queue:

You can add elements to a Python queue from the rear end. The process of adding elements is known as enqueuing. In the below example, you will create a Queue class and use the insert method to implement a FIFO queue.

```
# Creating the queue class
class Queue:
 def __init__(self):
    self.queue = list()
 def element_add_exm(self,data):
# Using the insert method
    if data not in self.queue:
       self.queue.insert(0,data)
       return True
    return False
 def leng(self):
    return len(self.queue)
Queue_add = Queue()
Queue_add.element_add_exm("Mercedes Benz")
Queue_add.element_add_exm("BMW")
Queue_add.element_add_exm("Maserati")
Queue_add.element_add_exm("Ferrari")
Queue_add.element_add_exm("Lamborghini")
print("Queue's Length: ",Queue_add.leng())
```

Output:
Queue length is 5

b. APPLICATIONS OF QUEUES

A queue data structure is generally used in scenarios where the FIFO approach (First In First Out) has to be implemented. The following are some of the most common queue applications in data structure:

- Managing requests on a single shared resource such as CPU scheduling and disk scheduling
- Handling hardware or real-time systems interrupts
- Handling website traffic
- Routers and switches in networking
- Maintaining the playlist in media players
- Queues can be used to manage and allocate resources, such as printers or CPU processing time.
- Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
- Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.

c. PRIORITY QUEUES

Priority queue in Python is a special type of queue that is executed based on priorities. It is not a completely FIFO queue as it sorts and dequeues an element based on priority and not based on when they were added. It calculates the priorities based on the ordering of their key pairs. They are most useful in scheduling tasks where priority is of importance. For instance, an operating system executes and completes a task based on priority; hence, a priority queue can be used here.

There are multiple ways to implement a priority queue. The two standard ways are through:

- Manually sorted list
- queue.PriorityQueue Class

Example: Implementing Priority Queue in Python with a Manually Sorted List

The manually sorted list can help identify and dequeue smaller and largest items. However,

inserting new elements can be challenging as it follows an O(n) time complexity. Hence, the best use of a manually sorted list can be when the number of insertions is minimal. In the code below, you will manually sort a list to implement a priority queue in Python.

```
priority_queu = []
priority_queu.append((3, 'Huindai'))
priority_queu.append((4, 'Mahindra'))
priority_queu.append((1, 'Maruthi'))
priority_queu.append((2, 'TATA'))
# Resort everytime a new element is added
priority_queu.sort(reverse=True)
while priority_queu:
  nxt_itm = priority_queu.pop()
  print(nxt_itm)
```

Output:
(1, 'Maruthi')
(2, TATA')
(3, Huindai')
(4, 'Mahindra')

Example: Implementing Priority Queue in Python with the queue.PriorityQueue Class

The queue.PriorityQueue class is a preferred option compared to a manually sorted list as it shares common time complexity with a standard queue. It also uses heapq (Heap Queue Algorithm) to perform quick operations.

The primary difference between a standard queue and a queue.PriorityQueue is that the latter offers coordination and locking semantics to handle multiple concurrent events. Here's an example of implementing a priority queue in Python.

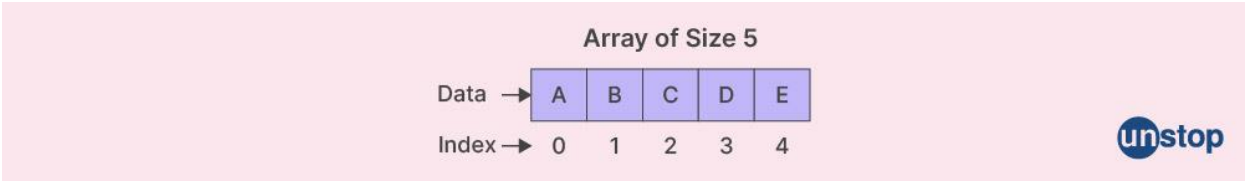
from queue import PriorityQueue

```
priority_queu = PriorityQueue()
priority_queu.put((3, 'Huindai'))
priority_queu.put((4, 'Mahindra'))
priority_queu.put((1, 'Maruthi'))
priority_queu.put((2, 'TATA'))
while not priority_queu.empty():
  nxt_itm = priority_queu.get()
  print(nxt_itm)
```

Output:
(1, 'Maruthi')
(2, TATA')
(3, Huindai')
(4, 'Mahindra')

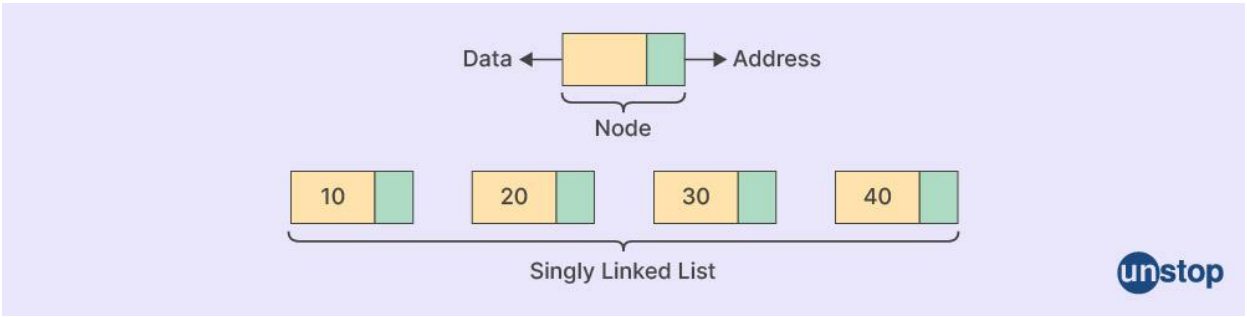
6. ARRAYS

The array is the most used Linear data type. The array stores the objects of the same data type in a linear fashion. Users can use an array to construct all linear or non-linear data structures. For example, Inside the car management software to store the car names array of the strings is useful. We can access the element of the array by the index of elements. In an array, the index always starts at 0. To prevent memory wastage, users should create an array of dynamic sizes.



7. LINKED LISTS

LinkedList data structure stores the data in the form of a node. Every linked list node contains the element value and address pointer. The address pointer of the LinkedList consists of the address of the next node. It can store unique or duplicate elements.

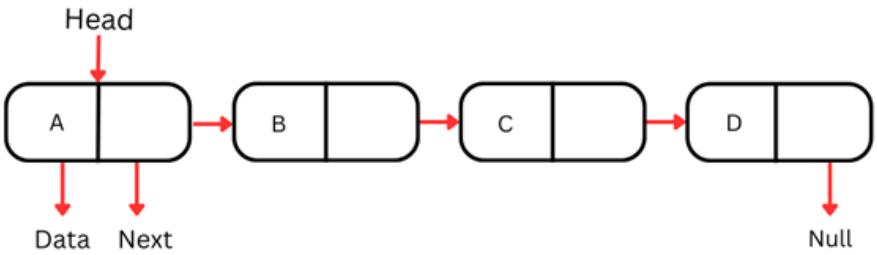


a. IMPLEMENTATION OF SINGLY LINKED LISTS

A singly-linked list is the simplest type of linked list, where each node contains some data and a reference to the next node in the sequence. They can only be traversed in a single direction - from the head (the first node) to the tail (the last node).

Each node in a singly-linked list typically consists of two parts:

- **Data:** The actual information stored in the node.
- **Next Pointer**: A reference to the next node. The last node's next pointer is usually set to null.



Since these data structures can only be traversed in a single direction, accessing a specific element by value or index requires starting from the head and sequentially moving through the nodes until the desired node is found. This operation has a time complexity of O(n), making it less efficient for large lists.

Inserting and deleting a node at the beginning of a singly-linked list is highly efficient with a time complexity of O(1). However, insertion and deletion in the middle or at the end requires traversing the list until that point, leading to an O(n) time complexity.

The design of singly-linked lists makes them a useful data structure when performing operations that occur at the beginning of the list.

The below program creates the linked list with three data elements.

```
class Node:
    def __init__(self, dataval=None):
    self.dataval = dataval
    self.nextval = None

class SLinkedList:
  def __init__(self):
    self.headval = None

list1 = SLinkedList()
list1.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
# Link first Node to second node
list1.headval.nextval = e2

# Link second Node to third node
e2.nextval = e3
```

Traversing a Linked List

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

Insertion in a Linked List

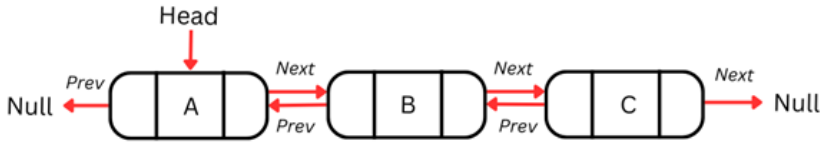
Inserting element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list, we have the below scenarios.

- **Inserting at the Beginning -** This involves pointing the next pointer of the new data node to the current head of the linked list. So the current head of the linked list becomes the second data element and the new node becomes the head of the linked list.
- **Inserting at the End -** This involves pointing the next pointer of the the current last node of the linked list to the new data node. So the current last node of the linked list becomes the second last data node and the new node becomes the last node of the linked list.
- **Inserting in between two Data Nodes -** This involves changing the pointer of a specific node to point to the new node. That is possible by passing in both the new node and the existing node after which the new node will be inserted. So we define an additional class which will change the next pointer of the new node to the next pointer of middle node. Then assign the new node to next pointer of the middle node.

b. DOUBLY LINKED LISTS

One disadvantage of singly-linked lists is that we can only traverse them in a single direction and cannot iterate back to the previous node if required. This constraint limits our ability to perform operations that require bidirectional navigation.

Doubly-linked lists solve this problem by incorporating an additional pointer within each node, ensuring that the list can be traversed in both directions. Each node in a doubly linked list contains three elements: the data, a pointer to the next node, and a pointer to the previous node.



Algorithm:

- Define a Node class which represents a node in the list. It will have three properties: data, previous which will point to the previous node and next which will point to the next node.
- Define another class for creating a doubly linked list, and it has two nodes: head and tail. Initially, head and tail will point to null.
- addNode() will add node to the list:
  - It first checks whether the head is null, then it will insert the node as the head.
  - Both head and tail will point to a newly added node.
  - Head's previous pointer will point to null and tail's next pointer will point to null.
  - If the head is not null, the new node will be inserted at the end of the list such that new node's previous pointer will point to tail.
  - The new node will become the new tail. Tail's next pointer will point to null.
- display() will show all the nodes present in the list.
  - Define a new node 'current' that will point to the head.
  - Print current.data till current points to null.
  - Current will point to the next node in the list in each iteration.

```
#Represent a node of doubly linked list
class Node:
   def __init__(self,data):
      self.data = data;
      self.previous = None;
      self.next = None;
    class DoublyLinkedList:
   #Represent the head and tail of the doubly linked list
   def __init__(self):
      self.head = None;
      self.tail = None;
```

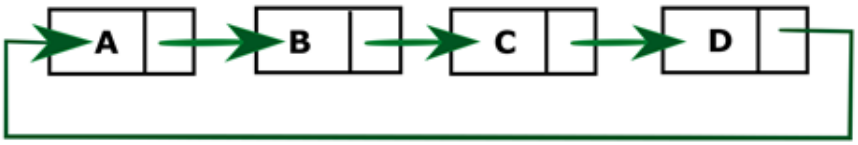
```
    #addNode() will add a node to the list
    def addNode(self, data):
        #Create a new node
        newNode = Node(data);

        #If list is empty
        if(self.head == None):
            #Both head and tail will point to newNode
            self.head = self.tail = newNode;
            #head's previous will point to None
            self.head.previous = None;
            #tail's next will point to None, as it is the last node of the list
            self.tail.next = None;
        else:
            #newNode will be added after tail such that tail's next will point to
newNode
            self.tail.next = newNode;
            #newNode's previous will point to tail
            newNode.previous = self.tail;
            #newNode will become new tail
            self.tail = newNode;
            #As it is last node, tail's next will point to None
            self.tail.next = None;

    #display() will print out the nodes of the list
    def display(self):
        #Node current will point to head
        current = self.head;
        if(self.head == None):
            print("List is empty");
            return;
        print("Nodes of doubly linked list: ");
        while(current != None):
            #Prints each node by incrementing pointer.
            print(current.data),;
            current = current.next;

dList = DoublyLinkedList();
#Add nodes to the list
dList.addNode(1);
dList.addNode(2);
dList.addNode(3);
dList.addNode(4);
dList.addNode(5);

#Displays the nodes present in the list
dList.display();
```

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

c. CIRCULAR LINKED LISTS

Circular linked lists are a specialized form of linked list where the last node points back to the first node, creating a circular structure. This means that, unlike the singly and doubly linked lists we've seen so far, the circular linked list does not end; instead, it loops around.

The cyclical nature of circular linked lists makes them ideal for scenarios that need to be looped through continuously, such as board games that loop back from the last player to the first, or in computing algorithms such as round-robin scheduling.



ALGORITHM:

- Define a Node class which represents a node in the list. It has two properties data and next which will point to the next node.
- Define another class for creating the circular linked list, and it has two nodes: head and tail. It has two methods: add() and display() .
- add() will add the node to the list:
    - It first checks whether the head is null, then it will insert the node as the head.
    - Both head and tail will point to the newly added node.
    - If the head is not null, the new node will be the new tail, and the new tail will point to the head as it is a circular linked list.
- display() will show all the nodes present in the list.
    - Define a new node 'current' that will point to the head.
    - Print current.data till current will points to head
    - Current will point to the next node in the list in each iteration

Program:

```
#Represents the node of list.
class Node:
  def __init__(self,data):
    self.data = data;
    self.next = None;
```

```
class CreateList:
 #Declaring head and tail pointer as null.
 def __init__(self):
  self.head = Node(None);
  self.tail = Node(None);
  self.head.next = self.tail;
  self.tail.next = self.head;

 #This function will add the new node at the end of the list.
 def add(self,data):
  newNode = Node(data);
  #Checks if the list is empty.
  if self.head.data is None:
   #If list is empty, both head and tail would point to new node.
   self.head = newNode;
   self.tail = newNode;
   newNode.next = self.head;
  else:
   #tail will point to new node.
   self.tail.next = newNode;
   #New node will become new tail.
   self.tail = newNode;
   #Since, it is circular linked list tail will point to head.
   self.tail.next = self.head;

 #Displays all the nodes in the list
 def display(self):
  current = self.head;
  if self.head is None:
   print("List is empty");
   return;
  else:
     print("Nodes of the circular linked list: ");
     #Prints each node by incrementing pointer.
     print(current.data),
     while(current.next != self.head):
        current = current.next;
        print(current.data),


class CircularLinkedList:
 cl = CreateList();
 #Adds data to the list
 cl.add(1);
 cl.add(2);
 cl.add(3);
 cl.add(4);
```

```
#Displays all the nodes present in the list
cl.display();
```

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

UNIT -IV
DICTIONARIES

1. INTRODUCTION

Python provides another composite data type called a dictionary, which is similar to a list in that it is a collection of objects. Dictionaries are Python's implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces ({}). A colon (:) separates each key from its associated value:

```
d = {
   <key>: <value>,
   <key>: <value>,
     .
     .
     .
   <key>: <value>
}
```

2. LINEAR LIST REPRESENTATION

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

Skip list in Data structure

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

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

Skip list structure

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

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

Complexity table of the Skip list

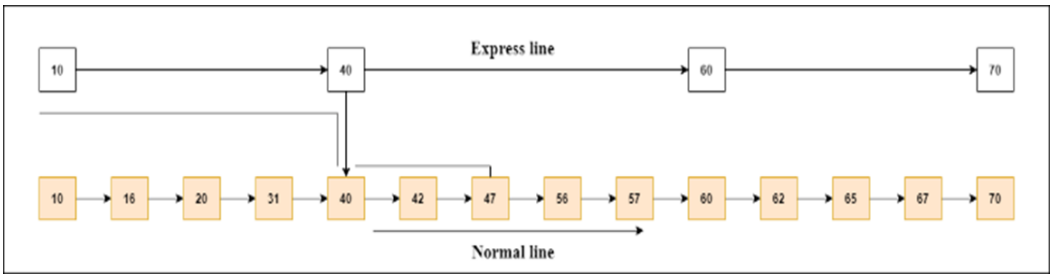
S. No	Complexity	Average case	Worst case
1.	Access complexity	O(logn)	O(n)
2.	Search complexity	O(logn)	O(n)
3.	Delete complexity	O(logn)	O(n)
4.	Insert complexity	O(logn)	O(n)
5.	Space complexity	-	O(nlogn)

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

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

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

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



Skip List Basic Operations

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

- Insertion operation: It is used to add a new node to a particular location in a specific situation.
- Deletion operation: It is used to delete a node in a specific situation.

- Search Operation: The search operation is used to search a particular node in a skip list.

Algorithm of the insertion operation

```
Insertion (L, Key)
local update [0...Max_Level + 1]
a = L → header
for i = L → level down to 0 do.
while a → forward[i] → key forward[i]
update[i] = a 7.
a = a → forward[0]
lvl = random_Level()
if lvl > L → level then
for i = L → level + 1 to lvl do
update[i] = L → header
L → level = lvl 14.
a = makeNode(lvl, Key, value)

for i = 0 to level do
a → forward[i] = update[i] → forward[i]
update[i] → forward[i] = a
```

Algorithm of deletion operation

```
Deletion (L, Key)
local update [0... Max_Level + 1]
a = L → header
for i = L → level down to 0 do.
while a → forward[i] → key forward[i]
update[i] = a
a = a → forward[0]
if a → key = Key then
for i = 0 to L → level do
if update[i] → forward[i] ? a then break
update[i] → forward[i] = a → forward[i]
free(a)
while L → level > 0 and L → header → forward[L → level] = NIL do
L → level = L → level – 1
```

Algorithm of searching operation

```
Searching (L, SKey)

a = L → header
loop invariant: a → key level down to 0 do.
while a → forward[i] → key forward[i]
```

a = a → forward[0]
if a → key = SKey then return a → value
else return failure

Advantages of the Skip list

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

Disadvantages of the Skip list

- It requires more memory than the balanced tree.
- Reverse searching is not allowed.
- The skip list searches the node much slower than the linked list.

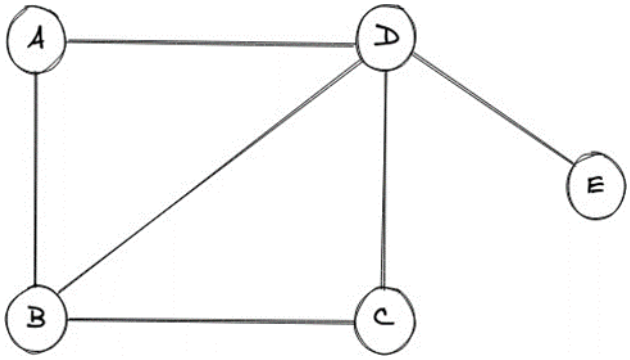
Applications of the Skip list

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

3. GRAPHS

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

In the figure below, we have a simple graph where there are five nodes in total and six edges.

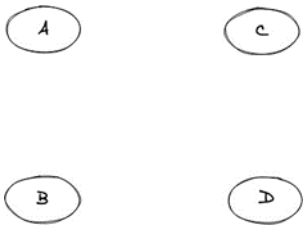


The nodes in any graph can be referred to as entities and the edges that connect different nodes define the relationships between these entities. In the above graph we have a set of nodes {V} = {A, B, C, D, E} and a set of edges, {E} = {A-B, A-D, B-C, B-D, C-D, D-E} respectively

Types of Graphs

A) Null Graphs

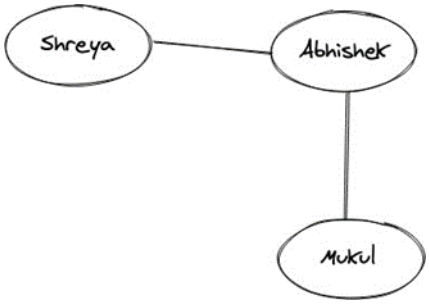
A graph is said to be null if there are no edges in that graph. A pictorial representation of the null graph is given below:



B) Undirected Graphs

If we take a look at the pictorial representation that we had in the Real-world example above, we can clearly see that different nodes are connected by a link (i.e. edge) and that edge doesn't have any kind of direction associated with it. For example, the edge between "Anuj" and "Deepak" is bi-directional and hence the relationship between them is two ways, which turns out to be that "Anuj" knows "Deepak" and "Deepak" also knows about "Anuj". These types of graphs where the relation is bi-directional or there is not a single direction, are known as Undirected graphs.

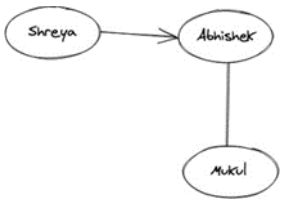
A pictorial representation of another undirected graph is given below:



C) Directed Graphs

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

A pictorial representation of the graph is given below:

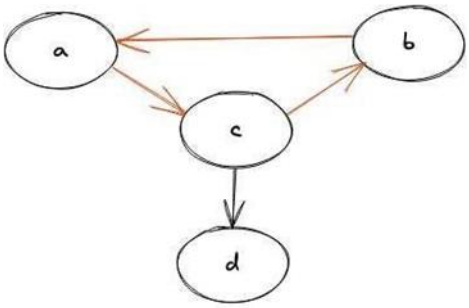


It can also be noted that the edge from "Shreya" to "Abhishek" is an outgoing edge for "Shreya" and an incoming edge for "Abhishek".

D) Cyclic Graph

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

A pictorial representation of a cyclic graph is given below:

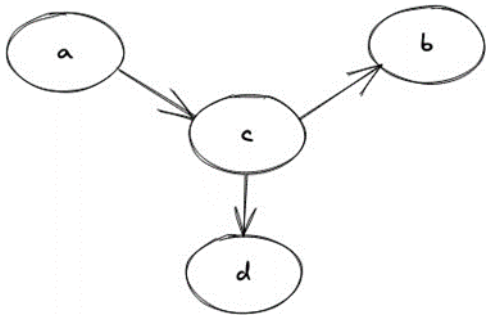


It can be easily seen that there exists a cycle between the nodes (a, b, c) and hence it is a cyclic graph.

E) Acyclic Graph

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

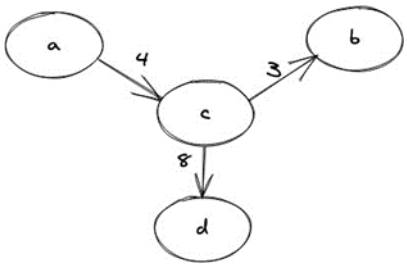
A pictorial representation of an acyclic graph is given below:



F) **Weighted Graph**

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

A pictorial representation of the weighted graph is given below:

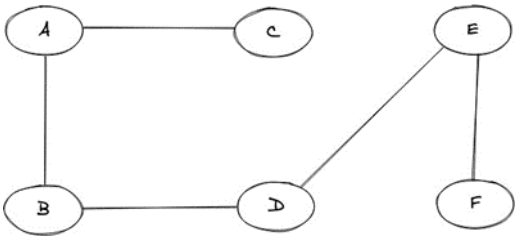


It can also be noted that if any graph doesn't have any weight associated with it, we simply call it an unweighted graph.

G) **Connected Graph**

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

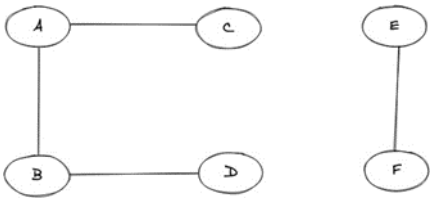
A pictorial representation of the connected graph is given below:



H) Disconnected Graph

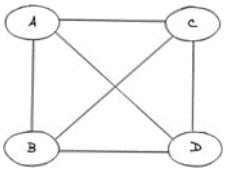
A graph that is not connected is simply known as a disconnected graph. In a disconnected graph, we will not be able to find a path from between every two nodes of the graph.

A pictorial representation of the disconnected graph is given below:



I) Complete Graph

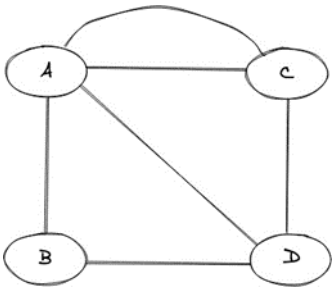
A graph is said to be a complete graph if there exists an edge for every pair of vertices(nodes) of that graph. A pictorial representation of the complete graph is given below:



J) Multigraph

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

A pictorial representation of the multigraph is given below:



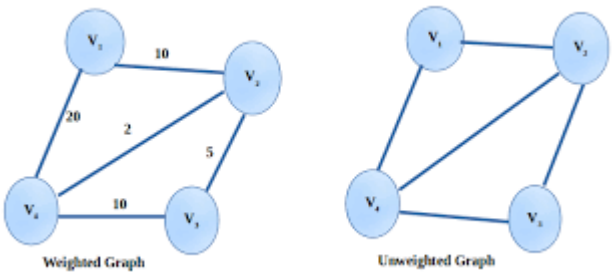
Commonly Used Graph Terminologies

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

4. WEIGHTED VS UNWEIGHTED GRAPHS

A weighted graph is defined as a special type of graph in which the edges are assigned some weights which represent cost, distance, and many other relative measuring units. A weight is a numerical value attached to each individual edge in the graph. Weighted Graph will contains weight on each edge where as unweighted does not. An unweighted graph is a graph in which the edges do not have weights or costs associated with them. Instead, they simply represent the presence of a connection between two vertices. When you doesn't have weight, all edges are considered equal. Shortest distance means less number of nodes you travel.

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



Real-Time Applications of Weighted Graph:

- Transportation networks: Using weighted graphs, we can figure things out like the path that takes the least time, or the path with the least overall distance. This is a simplification of how weighted graphs can be used for more complex things like a GPS system. Graphs are used to study traffic patterns, traffic light timings and much more by many big tech companies such as OLA, UBER, RAPIDO, etc. Graph networks are used by many map programs such as Google Maps, Bing Maps, etc.
- Document link graphs: Link weighted graphs are used to analyze relevance of web pages, the best sources of information, and good link sites by taking the count of the number of views as weights in the graph.
- Epidemiology: Weighted graphs can be used to find the maximum distance transmission from an infectious to a healthy person.
- Graphs in quantum field theory: Vertices represent states of a quantum system and the edges represent transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude. Research to find the maximum frequency along a path can be done using weighted graphs.
- Social network graphs: We can find which all users are connected in a network both directly(direct connection) and indirectly(indirect connection). But now weighted graphs are also used in social media for many purposes, for example, In recent times Instagram is using features like close friends which is not the same as all friends these features are being implemented using weighted graphs.
- Network packet traffic graphs: Network packet traffic graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

Applications of Unweighted Graph:

- Unweighted graphs are used to represent data that are not related in terms of magnitude.
- Unweighted graphs are used to represent computation flow.
- Representation of image segmentation, where pixels are represented as nodes and edges represent adjacency relationships.

- Representation of state spaces in decision-making processes and problem-solving in AI.
- Representation of information networks, such as the World Wide Web.

5. REPRESENTATIONS

Breadth-first search and Depth-first search in python are algorithms used to traverse a graph or a tree.

a. BREADTH FIRST SEARCH

Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

As breadth-first search is the process of traversing each node of the graph, a standard BFS algorithm traverses each vertex of the graph into two parts: 1) Visited 2) Not Visited. So, the purpose of the algorithm is to visit all the vertex while avoiding cycles.

BFS starts from a node, then it checks all the nodes at distance one from the beginning node, then it checks all the nodes at distance two, and so on. So as to recollect the nodes to be visited, BFS uses a queue.

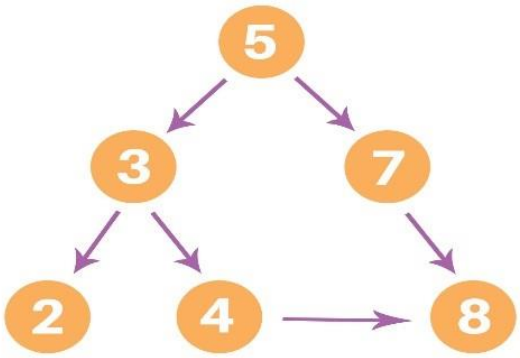
The steps of the algorithm work as follow:

- Start by putting any one of the graph's vertices at the back of the queue.
- Now take the front item of the queue and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
- Keep continuing steps two and three till the queue is empty.

The pseudocode for BFS in python goes as below:

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
   remove the head u of Q
   mark and enqueue all (unvisited) neighbors of u
```

Consider the following graph which is implemented in the code below:

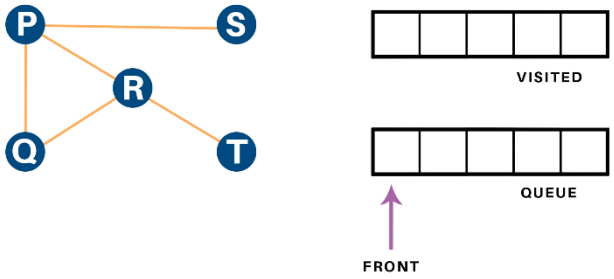



```
graph = {
 '5' : ['3','7'],
 '3' : ['2', '4'],
 '7' : ['8'],
 '2' : [],
 '4' : ['8'],
 '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
 visited.append(node)
 queue.append(node)

 while queue:          # Creating loop to visit each node
   m = queue.pop(0)
   print (m, end = " ")

   for neighbour in graph[m]:
     if neighbour not in visited:
       visited.append(neighbour)
       queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

The output of the above code will be as follow:

```
Following is the Breadth-First Search
5 3 7 2 4 8
```

Ex: Here, we will use an undirected graph with 5 vertices.



NOTE: Undirected graph with 5 vertices

We begin from the vertex P, the BFS algorithmic program starts by putting it within the Visited list and puts all its adjacent vertices within the stack.

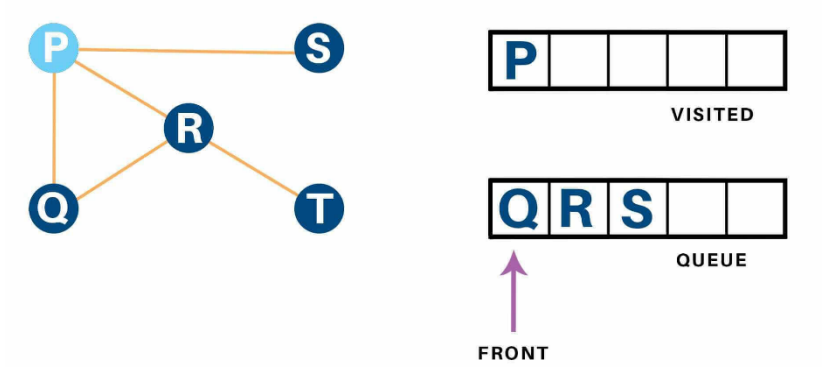


Fig 2: Visit starting verted and add its adjacent vertices to queue

Next, we have a tendency to visit the part at the front of the queue i.e. Q and visit its adjacent nodes. Since P has already been visited, we have a tendency to visit R instead.

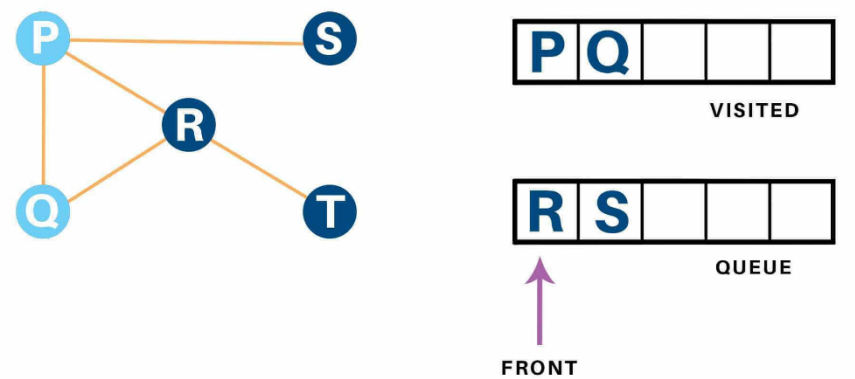


Fig 3: Visit the first neighbor of node p which is Q

Vertex R has an unvisited adjacent vertex in T, thus we have a tendency to add that to the rear

of the queue and visit S, which is at the front of the queue.

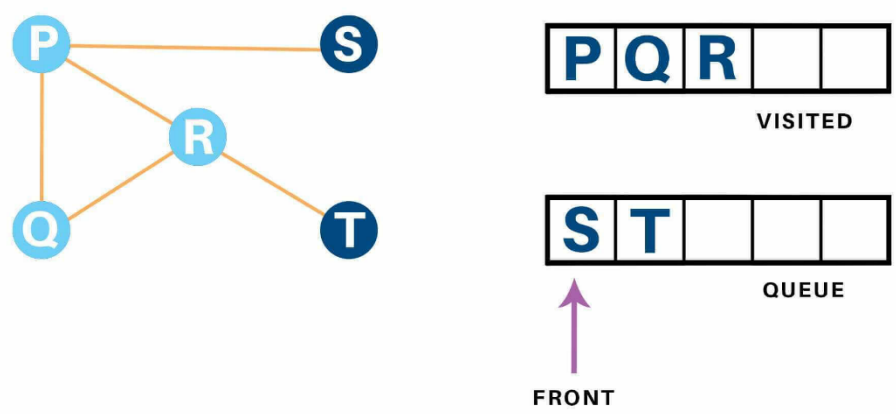


Fig 4: Visit R which was added to queue earlier to add its neighbor

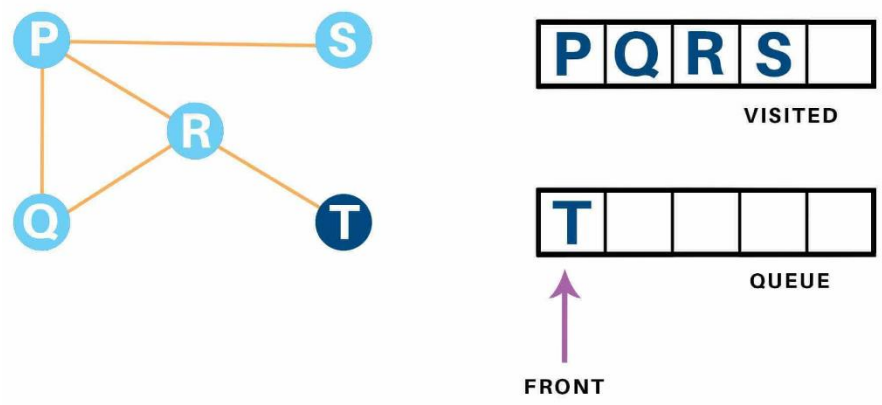


Fig 5: T remianing in queue

Now, only T remains within the queue since the only adjacent node of S i.e. P is already visited.

We have a tendency to visit it.

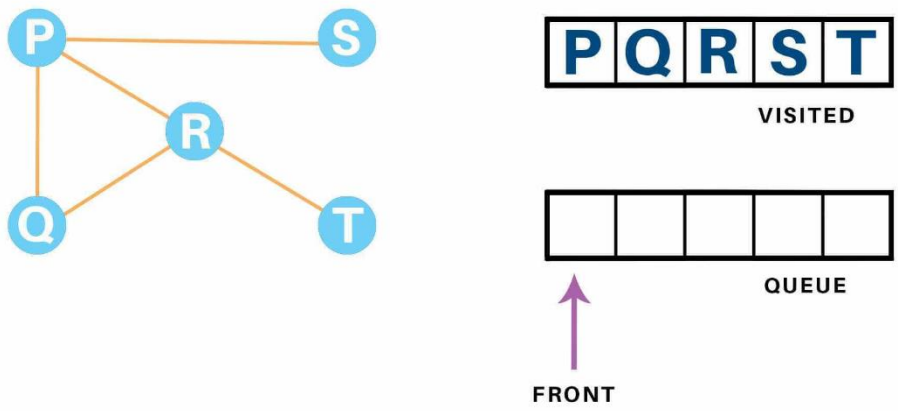


Fig 6: Visited all nodes hence list visited is full and queue is empty

Since the queue is empty, we've completed the Traversal of the graph.

The time complexity of the Breadth first Search algorithm is in the form of O(V+E), where V is

the representation of the number of nodes and E is the number of edges. Also, the space complexity of the BFS algorithm is O(V).

Applications of BFS Algorithm

- Breadth-first Search Algorithm has a wide range of applications in the real-world. Some of them are as discussed below:
- In GPS navigation, it helps in finding the shortest path available from one point to another.
- In pathfinding algorithms
- Cycle detection in an undirected graph
- In minimum spanning tree
- To build index by search index
- In Ford-Fulkerson algorithm to find maximum flow in a network.

b. DEPTH FIRST SEARCH.

Depth-first traversal or Depth-first Search is an algorithm to look at all the vertices of a graph or tree data structure.

The Depth-First Search is a recursive algorithm that uses the concept of backtracking. It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward on an equivalent path to seek out nodes to traverse. All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

A standard Depth-First Search implementation puts every vertex of the graph into one in all 2 categories: 1) Visited 2) Not Visited. The only purpose of this algorithm is to visit all the vertex of the graph avoiding cycles.

The DSF algorithm follows as:

- We will start by putting any one of the graph's vertex on top of the stack.
- After that take the top item of the stack and add it to the visited list of the vertex.
- Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
- Lastly, keep repeating steps 2 and 3 until the stack is empty.

The pseudocode for Depth-First Search in python goes as below: In the init() function, notice that we run the DFS function on every node because many times, a graph may contain two different disconnected part and therefore to make sure that we have visited every vertex, we can also run the DFS algorithm at every node.

```
DFS(G, u)
   u.visited = true
   for each v ∈G.Adj[u]
      if v.visited == false
         DFS(G,v)
init() {
   For each u ∈G
      u.visited = false
    For each u ∈G
      DFS(G, u)
}
```

Example

Let us see how the DFS algorithm works with an example. Here, we will use an undirected graph with 5 vertices.

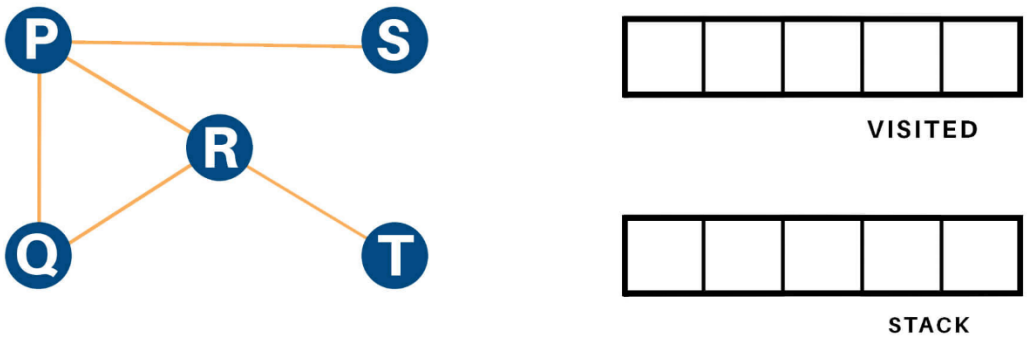


Fig 1 : Undirected graph with 5 vertices

We begin from the vertex P, the DFS rule starts by putting it within the Visited list and putting all its adjacent vertices within the stack.

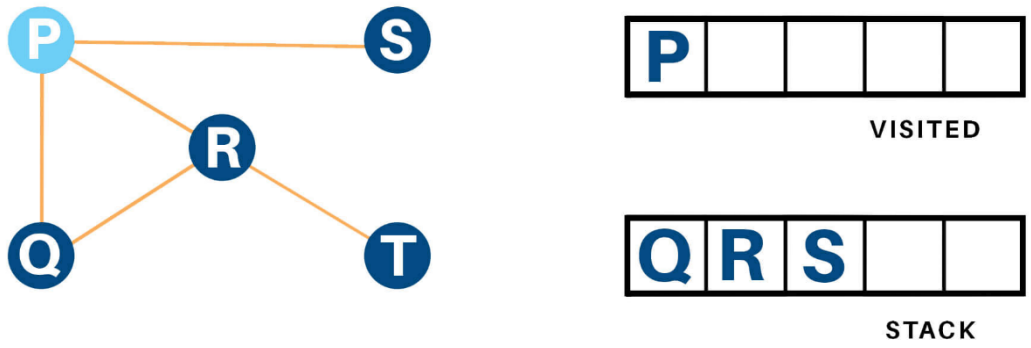


Fig 2: Visit starting vertex and add its adjacent vertices to stack

Next, we tend to visit the part at the highest of the stack i.e. Q, and head to its adjacent nodes. Since P has already been visited, we tend to visit R instead.

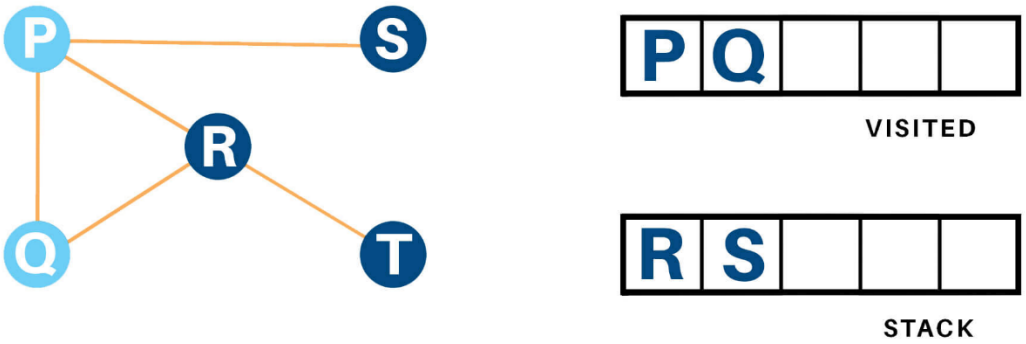


Fig 3: Visit the element at the top

Vertex R has the unvisited adjacent vertex in T, therefore we will be adding that to the highest of the stack and visit it.

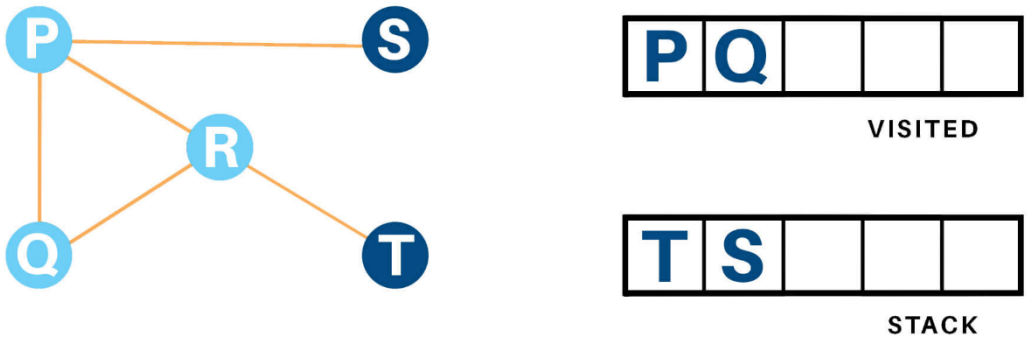


Fig 4: Vertex R has unvisited node T so we include it to the top of the stack and then visit it.

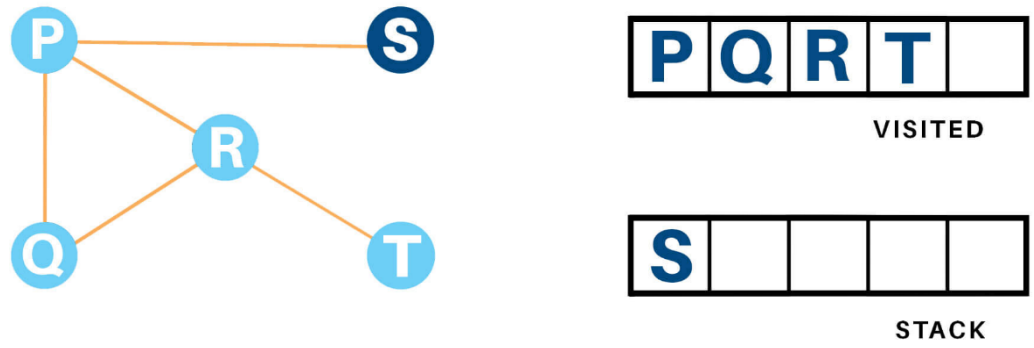


Fig 5: Vertex R has an unvisited node T, so we include it to the top of stack and then visit it.

At last, we will visit the last component S, it does not have any unvisited adjacent nodes, thus we've completed the Depth First Traversal of the graph.

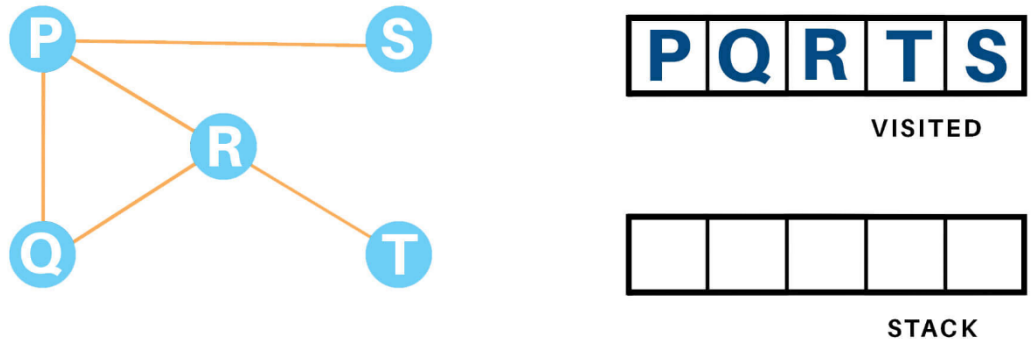


Fig 6: After visiting last node S, it doesn't have any adjacent node

The time complexity of the Depth-First Search algorithm is represented within the sort of O(V + E), where V is that the number of nodes and E is that the number of edges. The space complexity of the algorithm is O(V).

Applications

Depth-First Search Algorithm has a wide range of applications for practical purposes. Some of them are as discussed below:

- For finding the strongly connected components of the graph
- For finding the path
- To test if the graph is bipartite
- For detecting cycles in a graph
- Topological Sorting

- Solving the puzzle with only one solution.
- Network Analysis
- Mapping Routes
- Scheduling a problem

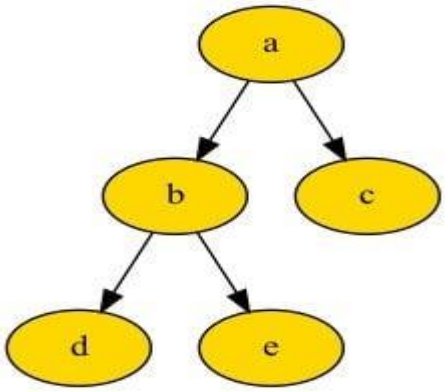
UNIT –V

TREES

1. OVERVIEW OF TREES

Trees are non-linear data structures that store data hierarchically and are made up of nodes connected by edges. For example, in a family tree, a node would represent a person, and an edge would represent the relationship between two nodes. Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

Example of tree:



a. TREE TERMINOLOGY

- **Root:** A node that doesn't have any parent, and the entire tree originates from it. In Fig. 1, the root is node a.
- **Leaf**: Nodes that don't have any child. In Fig. 1, leaf nodes are nodes c, d, and e.
- **Parent node:** Immediate predecessor of a node. In Fig. 1, node a is the parent node of nodes b and c.
- **Child node:** Immediate successor of a node. In Fig. 1, Node b is the child node of Node a.
- **Ancestors:** All predecessors of a node. In Fig. 1, nodes a and b are ancestors of node d.
- **Descendants**: All successors of a node. In Fig. 1, nodes d and e are descendants of node b.
- **Siblings:** Nodes that have the same parent. In Fig. 1, node d and e are siblings.

- **Left sibling:** Sibling to the left of the node. In Fig. 1, node d is the left sibling of node e.
- **Right sibling:** Sibling to the right of the node. In Fig. 1, node e is the right sibling of node d.
- **Depth:** Length of the path from node to root. In Fig. 1, the depth of node b is 2, and Node d is 3.
- **Height/Max depth:** Maximum depth of root to a leaf node. In Fig. 1, the height of the tree is 3.

Common Operations on Trees:

- **Traversal:** Traversing a tree allows you to visit each node in a specific order. Common traversal algorithms include depth-first traversal (pre-order, in-order, post-order) and breadth-first traversal.
- **Searching:** Searching for a specific node or value in a tree can be done using various search algorithms like depth-first search (DFS) or breadth-first search (BFS).
- **Insertion and Deletion:** Adding or removing nodes from a tree while maintaining its properties.
- **Height and Depth Calculation**: Calculating the height and depth of a tree or a specific node.
- **Balancing**: Balancing a tree to ensure efficient operations, especially in search trees like AVL trees or Red-Black trees.

Types of TREE Data Structure:

Tree data structure can be classified into three types based upon the number of children each node of the tree can have. The types are:

- **Binary tree:** In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
- **Ternary Tree:** A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
- **N-ary Tree or Generic Tree:** Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

Advantages of Tree Data Structure:

- Tree offer Efficient Searching Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
- Trees provide a hierarchical representation of data, making it easy to organize and navigate large amounts of information.
- The recursive nature of trees makes them easy to traverse and manipulate using recursive algorithms.

Disadvantages of Tree Data Structure:

- Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to inefficient search times.
- Trees demand more memory space requirements than some other data structures like arrays and linked lists, especially if the tree is very large.
- The implementation and manipulation of trees can be complex and require a good understanding of the algorithms.

2. BINARY TREES

Binary Tree is a form of a tree whose nodes cannot have more than two children. Each node of the binary tree has two pointers associated with it, one points to the left child, and the other points to the right child of the node. It is an unordered tree having no fixed organized structure for the arrangement of nodes. Binary Tree is slow for the searching, insertion, or deletion of the data because of its unordered structure. The time complexity of these operations is (□).

Key characteristics of a binary tree include:

- **Root Node:** The topmost node of the tree, serving as the starting point for traversing the tree.
- **Parent and Child Nodes:** Each node in a binary tree (except the root) has a parent node and may have up to two children nodes. The parent node is the immediate node above the child node.
- **Left and Right Children:** Each node can have at most two children, referred to as the left child and the right child. These children nodes are positioned to the left and right of the parent node, respectively.

- **Leaf Nodes:** Nodes that do not have any children are called leaf nodes or terminal nodes. They are the nodes at the bottom-most level of the tree.
- **Internal Nodes:** Nodes that have at least one child are called internal nodes. They are not leaf nodes and are located somewhere between the root node and the leaf nodes.
- **Depth and Height:** The depth of a node is the length of the path from the root node to that node. The height of a node is the length of the longest path from that node to a leaf node. The height of the binary tree is the height of the root node.
- **Binary Search Property:** In a binary search tree (a specific type of binary tree), the values stored in the left subtree of a node are less than the value of the node, and the values stored in the right subtree are greater than the value of the node. This property allows for efficient searching, insertion, and deletion operations.

a. IMPLEMENTATION

Crete the Node class

```
class Node:
    """
    A node class representing a single element in the binary tree.
    """
    def __init__(self, data):
        self.data = data
        self.left = None  # Left child
        self.right = None  # Right child
```

The insert function

```
def insert(root, data):
    """
    Inserts a new node with the given data into the binary tree.
    """
    if root is None:
        return Node(data)
    if data < root.data:
        root.left = insert(root.left, data)
    else:
        root.right = insert(root.right, data)
    return root
```

In-order traversal of the binary tree, printing data

```
def inorder_traversal(root):
    """
    Performs an in-order traversal of the binary tree, printing data.
    """
```

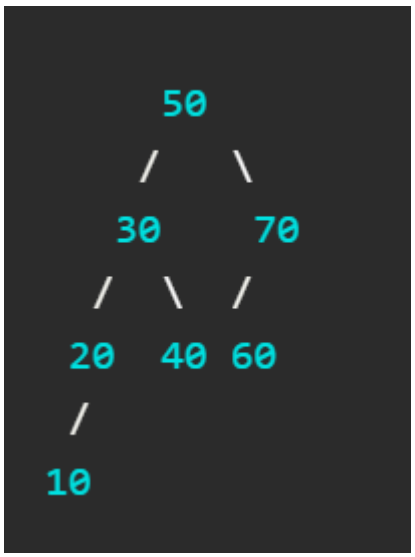
```
    if root is not None:
        inorder_traversal(root.left)
        print(root.data, end=" ")
        inorder_traversal(root.right)
```

example usage - add values

```
# Example usage
root = None
root = insert(root, 50)
insert(root, 30)
insert(root, 20)
insert(root, 40)
insert(root, 70)
insert(root, 60)
```

output

```
print("Inorder traversal: ")
inorder_traversal(root)
After inserting 50, 30, 20, 40, 70, 60
```



b. APPLICATIONS OF BINARY TREE

- File explorer.
- Used as the basic data structure in Microsoft Excel and spreadsheets.
- Editor tool: Microsoft Excel and spreadsheets.
- Evaluate an expression
- Routing Algorithms

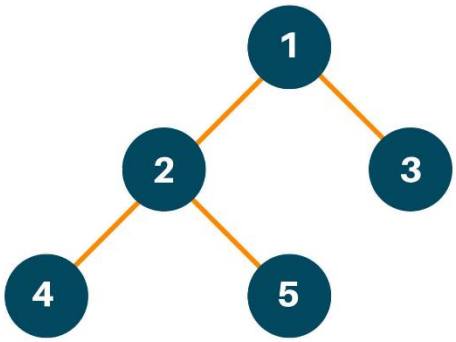
c. TREE TRAVERSALS

There are three types of tree traversal techniques, These traversal in trees are types of depth first search.

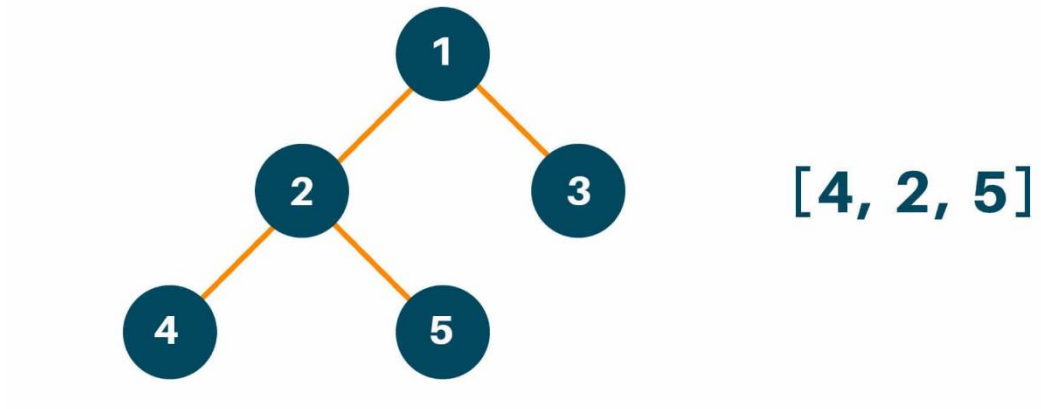
- **Inorder Traversal in Tree**
  - In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

```
PREORDER(n)
 if(n != null)
 print(n.data) // visiting root PREORDER(n.left) // visiting left subtree
PREORDER(n.right) // visiting right subtree
```

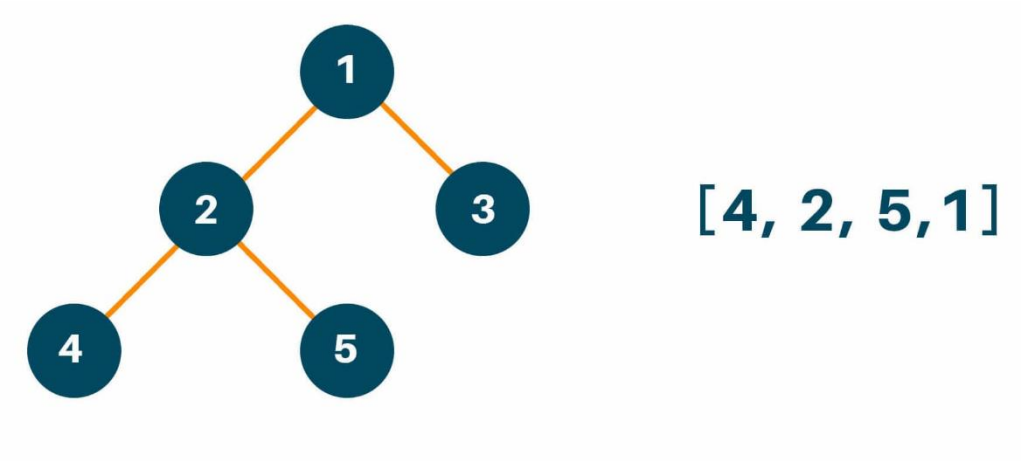
Let us consider the following binary tree:



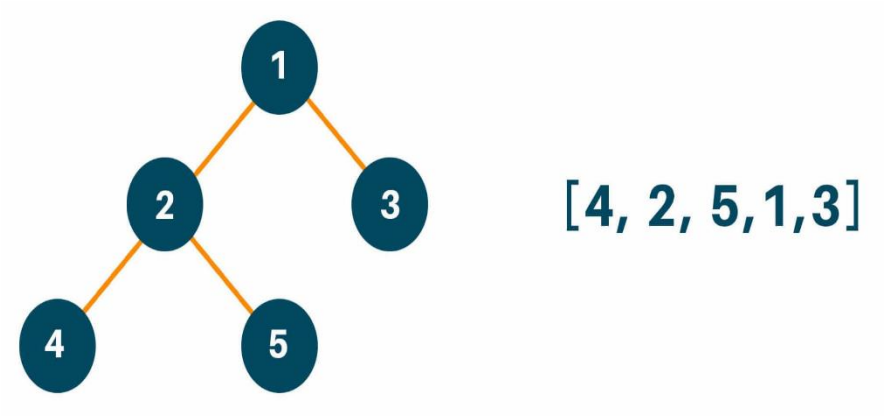
Now according to the inorder tree traversal method, we first traverse the left subtree of the original tree. Remember, even while traversing the left subtree, we will follow the same process i.e. left -> root -> right if the left child node of the original tree has furthermore child nodes. After traversing the left subtree, we will add the result to the array as shown below.



After following the first step, we will traverse the root node of the original tree as shown below.



Lastly, we will traverse the right subtree following the same process i.e. left -> root -> right if the right child node of the original tree has more than one child node.

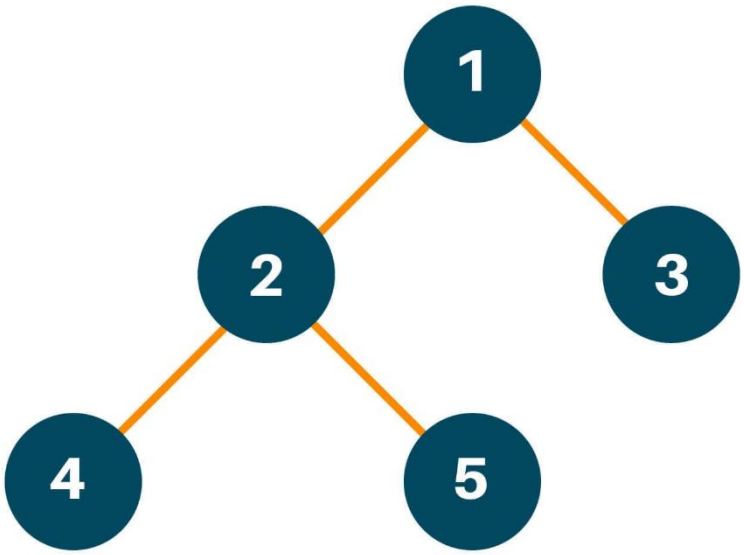


- **Preorder Traversal in Tree**

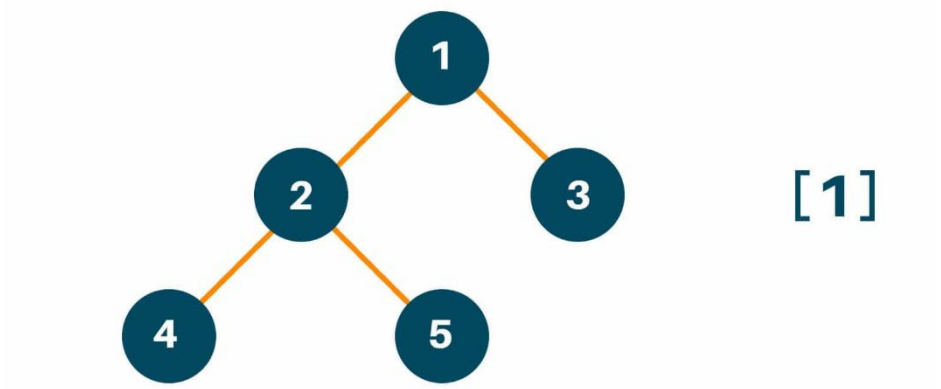
Using the preorder traversal method, we first visit the root of the original tree. Then we will traverse the left subtree of the original tree and lastly the right subtree of the original tree. Below is the Python code for Preorder Tree Traversal with recursion:

- Visit the root node
- Calling preorder (left-subtree)
- Calling preorder (right subtree)

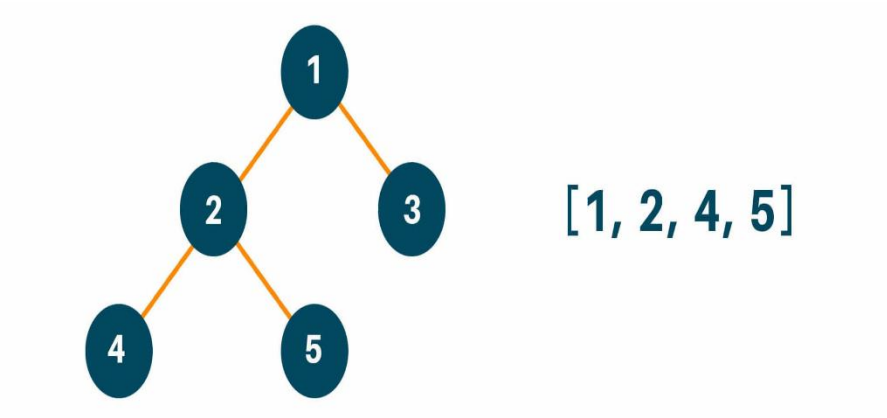
Let us consider the following example tree:



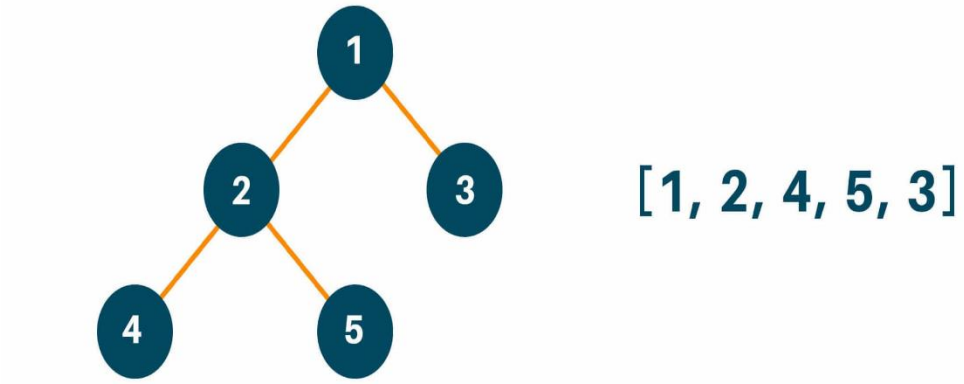
According to the preorder traversal method, we will first traverse the root node of the original tree and put it in the result array as shown in the figure below.



Then, we will traverse the left subtree of the original tree by calling the preorder traversal method. Here we will recursively call the function preorder to maintain the same process of traversal i.e root -> left -> right if the left child of the original tree has more than one child node and add the answer in the array as shown in the figure below.



Lastly, we will traverse the right subtree of the original tree similarly like we did with the left subtree and put the result in the answer array as shown below.

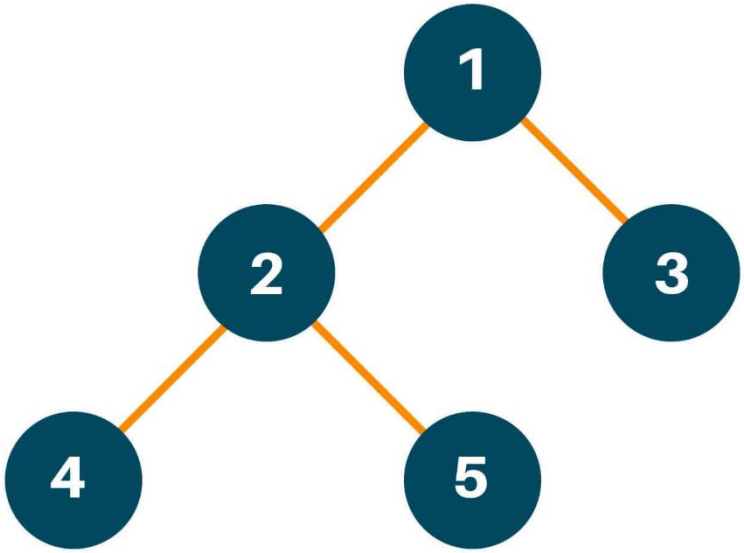


- **Postorder Traversal in Tree**

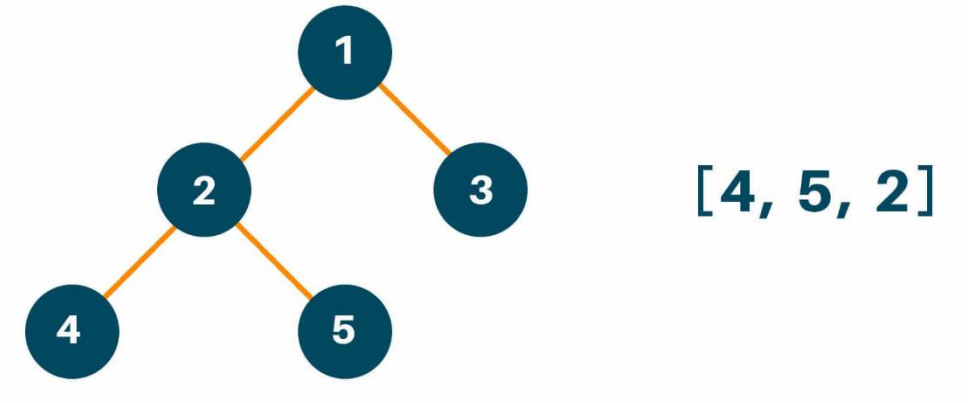
Using the postorder tree traversal method we first visit the left subtree of the original tree followed by the right subtree and lastly the root node of the original tree. Below is the Python code for Postorder Tree Traversal with recursion:

- Calling left-subtree
- Calling right-subtree
- Visit root node

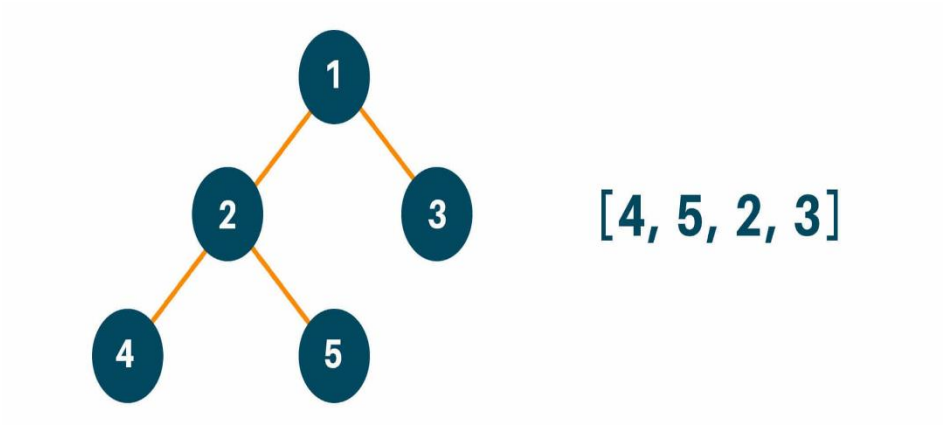
Let us consider the following example tree:



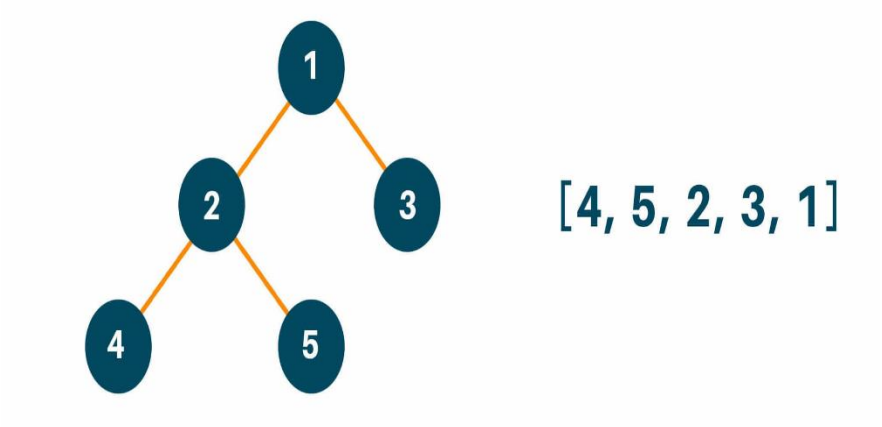
Using the postorder traversal method, we will first traverse the left subtree of the original tree. Remember that we will follow the same process of traversing of left subtree i.e left -> right -> root if the left subtree has more than one child node and then put the result in the answer array as shown in the figure.



Later we will traverse the right subtree of the original tree similarly like we did with the left subtree and add the answer in the result array as shown below.



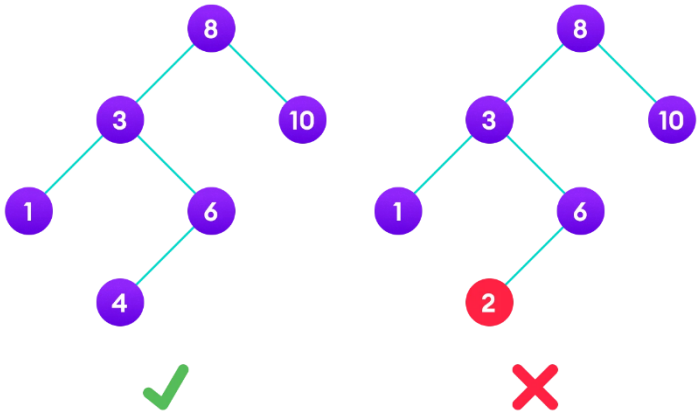
And lastly, we will traverse the root node of the original tree and finish our traversal method as shown in the figure below.



3. BINARY SEARCH TREES

A Binary Search Tree Python is a special type of tree, i.e., a nonlinear data structure with special properties like

- Each tree node can have a maximum of 2 children nodes, i.e., left node and right node.
- In each tree node, the value of the left child is less than or equal to its parent node, and the value of the right child will always be greater than or equal to its parent node value.
- Left_Subtree(node) <= node <= Right_Subtree(node)



Binary trees come in a variety of forms, including:

- **Complete binary tree:** The root key has a sub-tree with two or no nodes, and all levels of the tree are filled.
- **Balanced binary tree:** A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.
- **Full binary tree:** A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes. Conversely, there is no node in a full binary tree, which has one child node.
- Searching in a binary search tree has the worst-case complexity of O(n).

a. IMPLEMENTATION

How to Insert a Node in the Binary Search Tree?

In a Binary Search Tree Python, while inserting a node, we need to take care of its property, i.e., left <= root <= right. In this, we try to insert a new key value in an existing Binary Search Tree while retaining its properties and all the existing keys and values.

Algorithm:

- Start from the root node.
- When inserting an element, compare it to the root; if it is smaller than the root, call the left subtree recursively; otherwise, call the right subtree recursively.
- Simply insert the node at the left (if less than current) or the right (if not) after reaching the end.

Searching for a value

In a BST is very similar to how we found a value using Binary Search on an array. For Binary Search to work, the array must be sorted already, and searching for a value in an array can then be done really fast. Similarly, searching for a value in a BST can also be done really fast because of how the nodes are placed.

Algorithm:

- Start at the root node.
- If this is the value we are looking for, return.
- If the value we are looking for is higher, continue searching in the right subtree.
- If the value we are looking for is lower, continue searching in the left subtree.
- If the subtree we want to search does not exist, depending on the programming language, return None, or NULL, or something similar, to indicate that the value is not inside the BST.

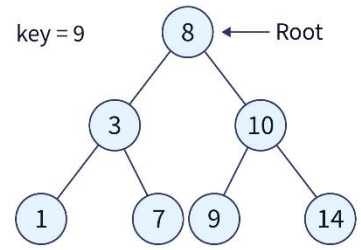
How to Delete a Node in the Binary Search Tree?

Deleting is one of the most important tasks of a Binary Search Tree Python, i.e., removing a node from the existing node without affecting the structure of the tree, other keys, or the property of a Binary Search Tree. It has three cases for deleting a node in a Binary Search Tree.

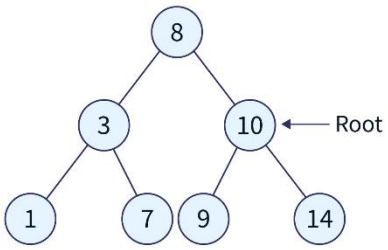
- Case 1: The leaf node in the first scenario is the node that has to be eliminated. Simply remove the node from the tree in this situation.
- Case 2: The node that has to be eliminated in the second scenario has just one child node. Do the following actions in this situation:
    - Put its descendant node in its place.
    - The child node should be moved from its starting location.
- Case 3: In the third scenario, the removed node has two offspring. Do the following actions in this situation:
    - Obtain that node's in-order successor.
    - Put the in-order successor in place of the node.
    - The in-order successor should be moved from its initial location.

Time Complexity

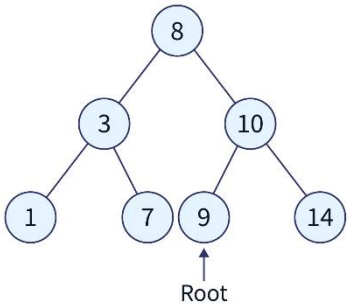
- Average cases : O(logn)
- Worst case : O(n) , when the tree is unbalanced.
- Space Complexity : O(n)



root.key < key Therefore move Right



root.key > key Therefore move Left



Now root.key == key
Therefore answer = root

4. AVL TREES

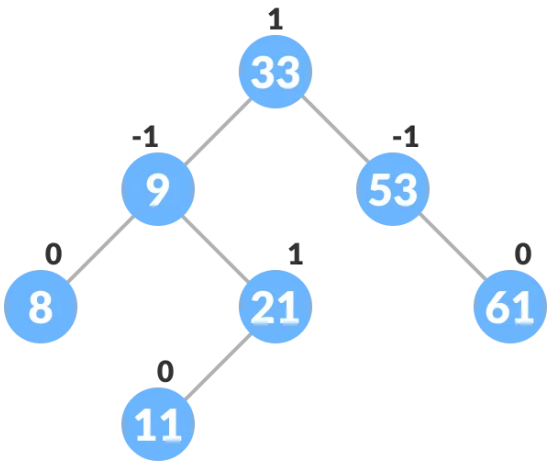
AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as balance factor. The AVL tree was introduced in the year 1962 by G.M. Adelson Velsky and E.M. Landis.

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 = height Of Left Subtree – height Of Right Subtree

The below tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.



AVL Tree Rotations

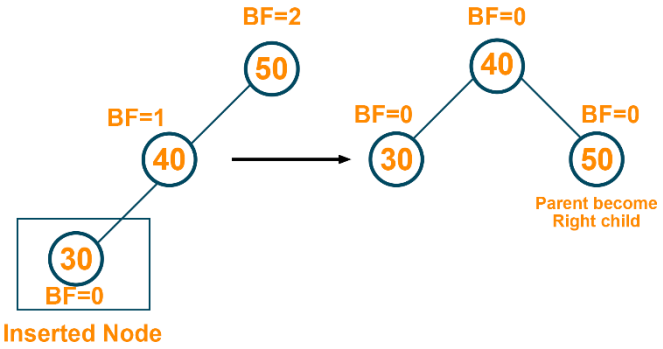
In AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition, then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation, we use rotation operations to make the tree balanced.

Rotation operations are used to make the tree balanced. Rotation is the process of moving nodes either to left or to right to make the tree balanced. There are four rotations and they are classified into two types.

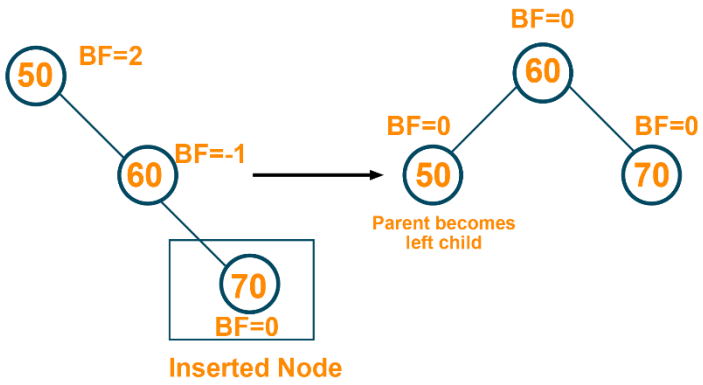
Single Rotation

Single rotation switches the roles of the parent and child while maintaining the search order. We rotate the node and its child, the child becomes a parent.

- **Single LL(Left Left) Rotation -** Here, every node of the tree moves toward the left from its current position. Therefore, a parent becomes the right child in the LL rotation. Let us see the below examples:



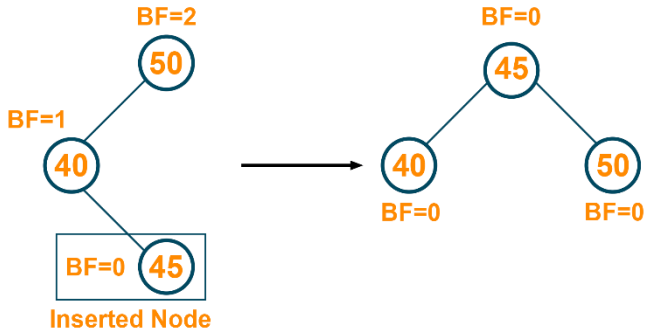
- **Single RR(Right Right) Rotation -** Here, every node of the tree moves toward the right from the current position. Therefore, the parent becomes a left child in RR rotation. Let us see the below example:



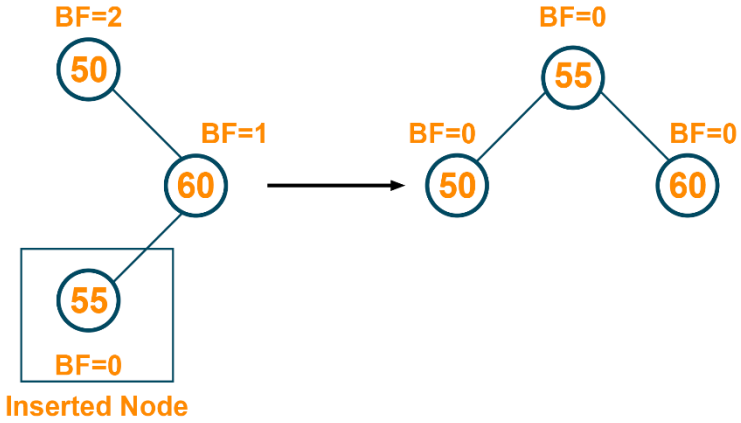
Double Rotation

Single rotation does not fix the LR rotation and RL rotation. For this, we require double rotation involving three nodes. Therefore, double rotation is equivalent to the sequence of two single rotations.

- **LR(Left-Right) Rotation -** The LR rotation is the process where we perform a single left rotation followed by a single right rotation. Therefore, first, every node moves towards the left and then the node of this new tree moves one position towards the right. Let us see the below example:



- **RL (Right-Left) Rotation -** The RL rotation is the process where we perform a single right rotation followed by a single left rotation. Therefore, first, every node moves towards the right and then the node of this new tree moves one position towards the left. Let us see the below example:



Python's standard library does not include an AVL tree implementation. There are, however, a number of third-party libraries available that provide AVL tree functionality.

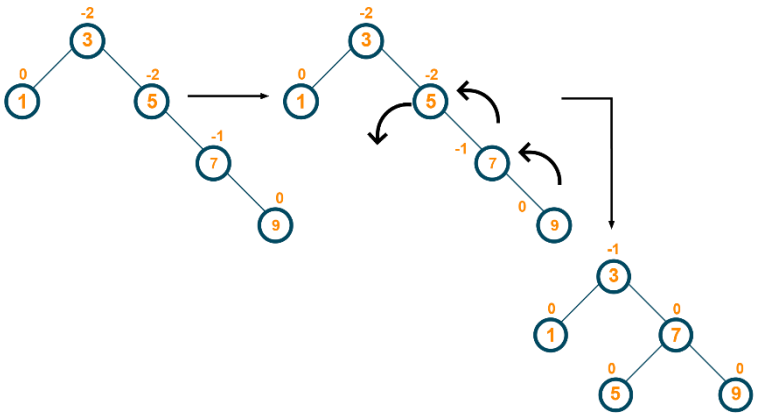
The 'avl tree' is a popular Python library for working with AVL trees, and it can be installed with pip: pip install avl_tree.

INSERTION OPERATION IN AVL TREE

In the AVL tree, the new node is always added as a leaf node. After the insertion of the new node, it is necessary to modify the balance factor of each node in the AVL tree using the rotation operations. The algorithm steps of insertion operation in an AVL tree are:

- Find the appropriate empty subtree where the new value should be added by comparing the values in the tree
- Create a new node at the empty subtree
- The new node is a leaf ad and thus will have a balance factor of zero
- Return to the parent node and adjust the balance factor of each node through the

rotation process and continue it until we are back at the root. Remember that the modification of the balance factor must happen in a bottom-up fashion.

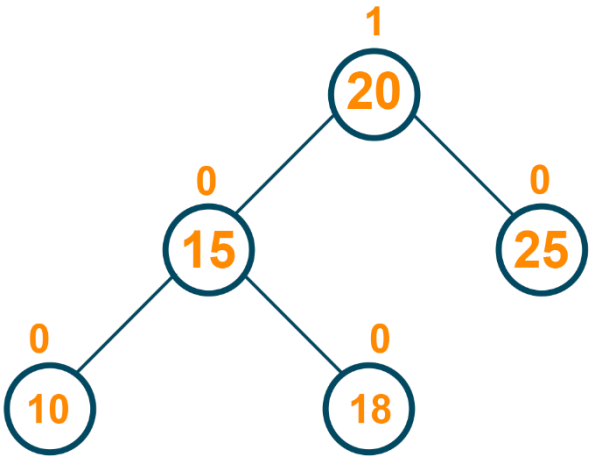


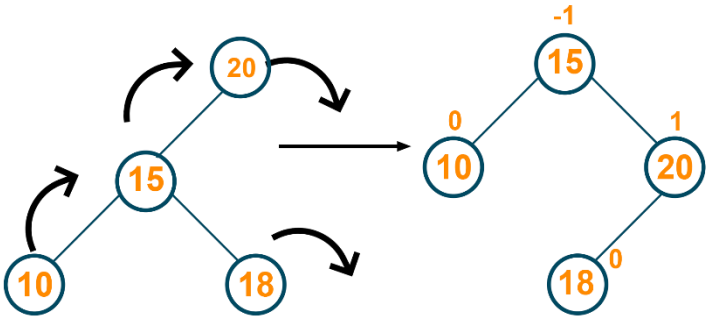
DELETION OPERATION IN AVL

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:

- Locate the node to be deleted
- If the node does not have any child, then remove the node
- If the node has one child node, replace the content of the deletion node with the child node and remove the node
- 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.
- Update the balance factor of the AVL tree

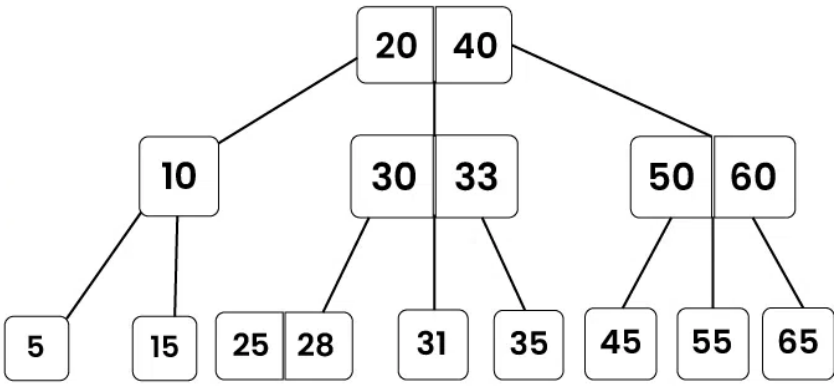
Ex: we have to delete the node '25' from the tree. As the node to be deleted does not have any child node, we will simply remove the node from the tree:





5. B-TREES

A B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.



B-TREE TRAVERSALS

Step-by-step algorithm:

- Create a node with the name ‘BTreeNode‘ to create a default node with a list of keys and a list of children nodes.
- The b-tree class will be created which has two attributes ‘root‘ and ‘t‘, root represents

the root node, and ‘t‘ represents the minimum degree of the B-tree.

- The display function in class ‘Btree‘ will be used to print to nodes of the tree in level-wise format.
- The final tree is created in the main function by inserting keys in the B-tree.

SEARCH OPERATION IN B TREE

B tree makes it convenient for users to search for an element in it like searching for an element in any other binary tree. Let us see how it searches for an element ‘m‘ in the tree.

Step-by-step algorithm:

- m is not found in the root so we will compare it with the key. i.e. compare 10 and 100.
- Since 10<100, we will search in the left part of the tree.
- Now we will compare m with all the elements in the current node i.e. we will compare m with 35 and 65 in order.
- Since m is smaller than both 35 and 65, we will go to the left side of the tree to find the element.
- Now we will compare m with all the keys of the current node i.e. 10 and 20 but the first key of this node is equal to m so we found the element.
- Time Complexity : O(log n), Auxiliary Space: O(n)

INSERT OPERATION IN B TREE

Inserting an element refers to adding the element at a certain position in the tree. Let us see how the B tree allows the insertion of an element.

Step-by-step algorithm:

- For an empty tree, the root node is declared and the element m is directly inserted.
- Use the search operation explained above to find the location where the new element is to be inserted.
- Case 1: If the node has already fulfilled its capacity of several keys
- then the node needs to be split. For this, firstly insert the key normally and then split the keys from the median.
- This middle key will move with the parent to separate the left and right nodes.
- Case 2: If the node has not fulfilled its capacity of several keys

- Simply insert the key at the required position.
- After the element is inserted, ensure the number of keys is updated.

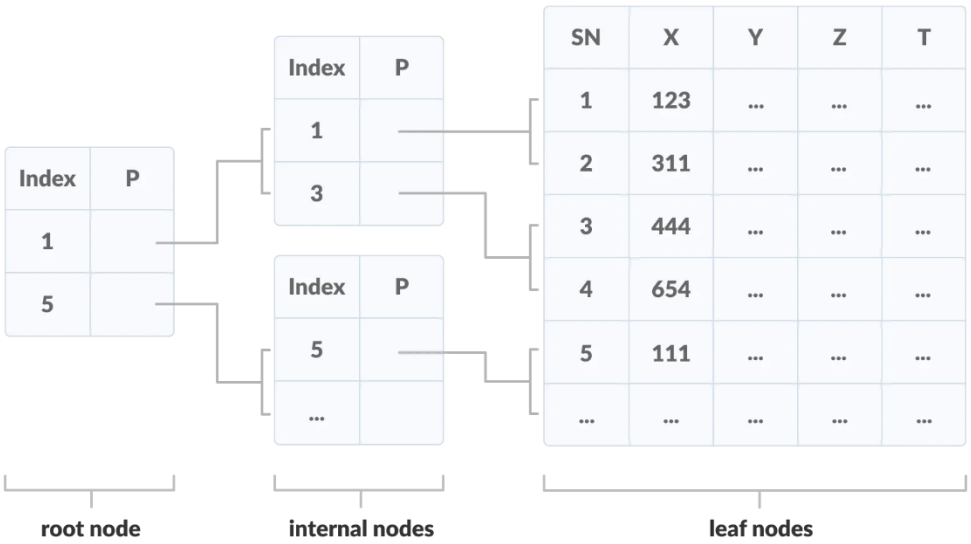
DELETE OPERATION IN B TREE

Deleting elements refers to removing the element from a certain position in the tree. Let us see how the B tree allows the deletion of an element. We need to study 3 different cases for deleting an element.

- **Case 1: Deletion of a leaf node -** We search for the particular element to be deleted. If it is a leaf node then we need to check that it doesn't break the rule of a minimum number of keys of a node. If it does reduce the number of keys than the minimum number of keys, then we will take a key from the neighboring sibling from left to right checking for the key that has more than the minimum number of required keys. If it doesn't violate the number of minimum keys then we can simply join the parent node to the left or right sibling to delete that key.
- **Case 2: Deletion of an internal node -** We search for the particular element to be deleted. This node will be deleted and be replaced by its left child which comes just before it in the inorder predecessor. Note that it is only possible if the left child has more than the minimum number of keys. In case the left node doesn't have more than the minimum number of keys. then the deleted node is replaced by its right child which comes just after it in the inorder successor. Note that it is only possible if the left child has more than the minimum number of keys. If both children have a case where they only have the minimum number of keys, they are merged to replace that deleted node.
- **Case 3: If the height of the tree reduces -** If we witness case 2 repeatedly, then we will have to merge the left and right nodes several times which can shrink the tree. In this case, children are re-arranged in increasing order to maintain the height of the tree.

6. B+ TREES

A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level. An important concept to be understood before learning B+ tree is multilevel indexing. In multilevel indexing, the index of indices is created as in figure below. It makes accessing the data easier and faster.



Properties of a B+ Tree

- All leaves are at the same level.
- The root has at least two children.
- Each node except root can have a maximum of m children and at least m/2 children.
- Each node can contain a maximum of m - 1 keys and a minimum of ⌈ m/2⌉ - 1 keys.

Comparison between a B-tree and a B+ Tree

- The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.
- The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.
- Operations on a B+ tree are faster than on a B-tree.

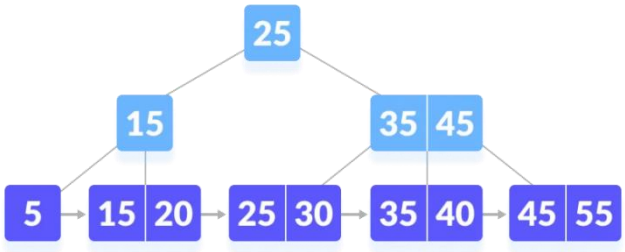
SEARCHING ON A B+ TREE

The following steps are followed to search for data in a B+ Tree of order m. Let the data to be searched be k.

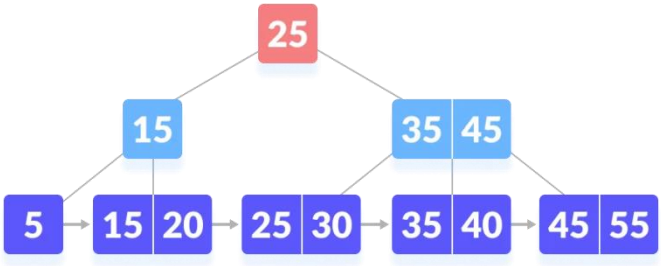
- Start from the root node. Compare k with the keys at the root node [k1, k2, k3,.....km - 1].
- If k < k1, go to the left child of the root node.
- Else if k == k1, compare k2. If k < k2, k lies between k1 and k2. So, search in the left child of k2.
- If k > k2, go for k3, k4,...km-1 as in steps 2 and 3.
- Repeat the above steps until a leaf node is reached.
- If k exists in the leaf node, return true else return false.

EXAMPLE:

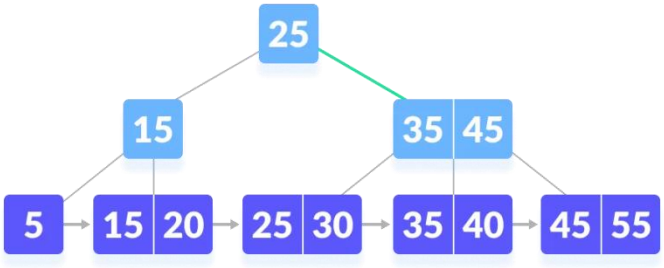
Let us search k = 45 on the following B+ tree.



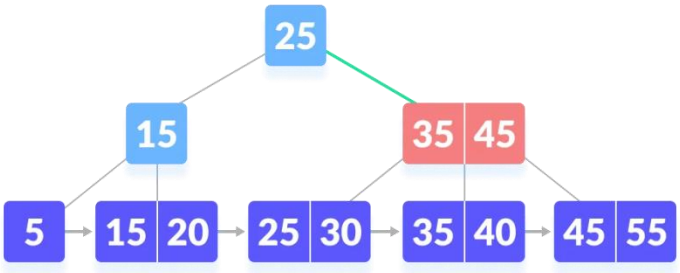
i. Compare k with the root node.k is not found at the root



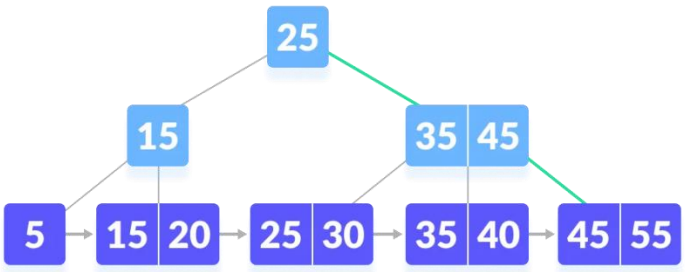
ii. Since k > 25, go to the right child. (**Go to right of the root)**



iii. Compare k with 35. Since k > 30, compare k with 45. **(K Not Found)**



iv. Since k ≥ 45, so go to the right child. (**go to the right)**



v. k is found. **(k is found)**