Savitribai Phule Pune University

S.Y.B.Sc.(Cyber and Digital Science)

SEMESTER-III

(CDS-233)Lab Course on Data Structure using Python

WorkBook

Student Name: _____

College Name: _____

Roll No.:_____Division:_____Seat No: _____

Academic Year: _____

CERTIFICATE

This is to certify that Mr./Ms._____

Seat Number _____ of S.Y.B.Sc. (Cyber and Digital Science) Semester III has successfully completed Laboratory course

On Data structure using Python. He/She has scored_mark out of 10 (For Lab Book).

Subject Teacher **H.O.D./Coordinator**

Internal Examiner **External Examiner**

Introduction

1. **About the Workbook:**
   This workbook is intended to be used by S.Y.B.Sc.(Cyber and Digital Science) Semester III students for Data structure using Python. This workbook is designed by considering all the practical concepts / topics mentioned in syllabus.

2. **The objectives of this Workbook are:**
   1) Defining the scope of the course.
   2) To bring the uniformity in the practical conduction and implementation in all colleges affiliated to SPPU.
   3) To have continuous assessment of the course and students.
   4) Providing ready reference for the students during practical implementation.
   5) Provide more options to students so that they can have good practice before facing the examination.
   6) Catering to the demand of slow and fast learners and accordingly providing the practice assignments to them.

3. **Instructions to the students:**
   Please read the following instructions carefully and follow them.

- Students are expected to carry this workbook every time they come to the lab for practical.
- Students should prepare for the assignment by reading the relevant material which is mentioned in ready reference.
- Instructor will specify which problems to solve in the lab during the allotted slot and student should complete them and get verified by the instructor. However, student should spend additional hours in Lab and at home to cover all workbook assignments if needed.

- Students will be assessed for each assignment on a scale from 0 to 5

Not done	0
Incomplete	1
Late Complete	2
Needs improvement	3
Complete	4
Well Done	5

4. Instruction to the Instructors:

Make sure that students should follow above instructions.

Explain the assignment and related concepts using white board if required or by demonstrating the software.
Give specific input to fill the blanks in queries which can vary from student to student.
Evaluate each assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.
The value should also be entered on assignment completion page of the respective Lab course.

5. **Instructions to the Lab administrator:**

You have to ensure appropriate hardware and software is made available to each student.

Editors:

Mr. Satyavan kunjir — Dr. D. Y. Patil ACS College, Pimpri,
Dr. Sonali Nemade — Dr. D. Y. Patil ACS College, Pimpri, Pune
Mr. Yogesh Ingale — Dr. D. Y. Patil ACS College, Pimpri, Pune.

Reviewed By:

Dr. Ranjit D. Patil — Principal Dr. D. Y. Patil ACS College, Pimpri, Pune
Dr. Sujata Patil — Dr. D. Y. Patil ACS College, Pimpri, Pune

Assignment Completion Sheet

Sr. No.	Assignment Name	Marks (Out of 5)	Teacher's Sign
1	Arrays		
2	Link List		
3	Stack		
4	Queue		
5	Introduction to Tree		
6	Graph		
Total (Out of 30)			
Total (Out of 10)			

Signature of Instructor:

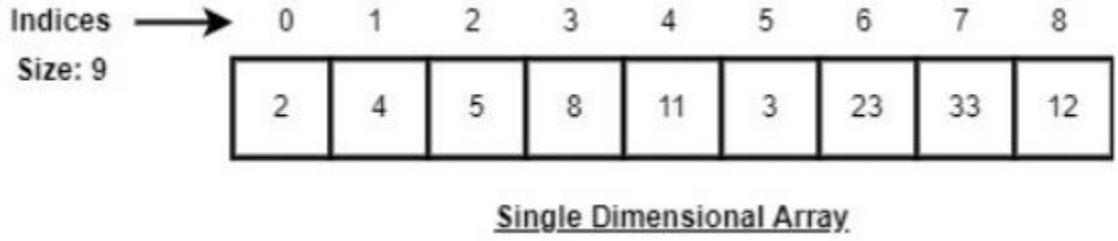
Exercise 1: Arrays

Introduction:

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element**− Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation:



Single Dimensional Array

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

- Index starts with 0.
- Array length is 09 which mean it can store 09 elements.

Basic Operations:

Following are the basic operations supported by an array.

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

Array is created in Python by importing array module to the python program. Then the array is declared as shown blow.

```
from array import *
arrayName = array(typecode, [Initializers])
```

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are:

Typecode	Value
b	Represents signed integer of size 1 byte/td>

B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Before looking at various array operations lets create and print an array using python.
The below code creates an array named array1.

```
from array import *
array1 = array('i', [10,20,30,40,50])
for x in array1:
 print(x)
```

Accessing Array Element:

We can access each element of an array using the index of the element. The below code shows how

```
from array import *
array1 = array('i', [10,20,30,40,50])
print (array1[0])
print (array1[2])
```

Insertion Operation:

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

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
array1.insert(1,60)
for x in array1:
 print(x)
```

Deletion Operation:

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *
```

```
array1 = array('i', [10,20,30,40,50])
array1.remove(40)
for x in array1:
 print(x)
```

Search Operation:

You can perform a search for an array element based on its value or its index.

Here, we search a data element using the python in-built index() method.

```
from array import *
array1 = array('i', [10,20,30,40,50])
print (array1.index(40))
```

Update Operation:

Update operation refers to updating an existing element from the array at a given index.

Here, we simply reassign a new value to the desired index we want to update.

```
from array import *
array1 = array ('i', [10,20, 30,40,50])
array1[2] = 80
for x in array1:
 print(x)
```

Searching Techniques:

Linear Search: Linear Search is a searching technique to search some particular value from an array. This is the simplest searching technique.

In this searching technique,

- The value to be searched is compared with all the elements in the array.
- If the value is found, the index of the element is returned.
- If the particular element is not present throughout the array, then return -1 or some relevant string message.

```
def linearSearch(arr,value):
  for i in range(len(arr)):
    if(arr[i]==value):
      return i
  return -1
array=[1,2,3,4,5,6,7,8,9,10]
value=5
```

```
a=linearSearch(array,value)
if(a==-1):
  print("Element not present")
else:
  print("Element present at index",a)
```

Binary Search:

Problem statement − We will be given a sorted list and we need to find an element with the help of a binary search.

Algorithm

- Compare x with the middle element.
- If x matches with the middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for the right half.
- Else (x is smaller) recur for the left half

```
def binarySearchAppr (arr, start, end, x):
  if end >= start: # check condition
    mid = start + (end- start)   # If element is present at the middle
    if arr[mid] == x:
    return mid  # If element is smaller than mid
    elif arr[mid] > x:
    return binarySearchAppr(arr, start, mid-1, x)
    # Else the element greator than mid
    else:
    return binarySearchAppr(arr, mid+1, end, x)
  else:
    # Element is not found in the array
    return -1
arr = sorted(['t','u','t','o','r','i','a','l'])
  x ='r'
  result = binarySearchAppr(arr, 0, len(arr)-1, x)
if result != -1:
  print ("Element is present at index "+str(result))
else:
```

```
print ("Element is not present in array")
```

Sorting Methods:

Bubble Sort:

Approach

- Starting with the first element(index = 0), compare the current element with the next element of the array.
- If the current element is greater than the next element of the array, swap them.
- If the current element is less than the next element, move to the next element.
- Repeat above steps n-1 times

```
def bubbleSort(ar):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
    # Last i elements are already in correct position
    for j in range(0, n-i-1):
        # Swap if the element found is greater than the next element
        if ar[j] > ar[j+1] :
            ar[j], ar[j+1] = ar[j+1], ar[j]
# Driver code to test above
ar = ['t','u','t','o','r','i','a','l']
bubbleSort(ar)
print ("Sorted array is:")
for i in range(len(ar)):
    print (ar[i])
```

Insertion Sort:

Insertion sort is the simple method of sorting an array. In this technique, the array is virtually split into the sorted and unsorted part. An element from unsorted part is picked and is placed at correct position in the sorted part.

- The array elements are traversed from 1 to n.
- If the array element at position i is greater than its predecessor, it does not need to be moved.

- If the array element at position i is less than its predecessor, it needs to be shifted towards left until we find a predecessor smaller than it or if we reach at the leftmost position in the array.

```
def insertionSort(arr):
  for i in range(1, len(arr)):
    key = arr[i]
    # Move elements of arr[0..i-1], that are greater than key,
    # to one position ahead of their current position
    j = i-1
    while j >=0 and key < arr[j] :
      arr[j+1] = arr[j]
      j -= 1
     arr[j+1] = key
# main
arr = ['t','u','t','o','r','i','a','l']
insertionSort(arr)
print ("The sorted array is:")
for i in range(len(arr)):
  print (arr[i])
```

Selection Sort:

In the **selection sort** algorithm, an array is sorted by recursively finding the minimum element from the unsorted part and inserting it at the beginning. Two subarrays are formed during the execution of Selection sort on a given array.

- The subarray, which is already sorted
- The subarray, which is unsorted.

During every iteration of selection sort, the minimum element from the unsorted subarray is popped and inserted into the sorted subarray.

```
A = ['t','u','t','o','r','i','a','l']
for i in range(len(A)):
  min_= i
  for j in range(i+1, len(A)):
```

```
    if A[min_] > A[j]:
      min_ = j
  #swap
A[i], A[min_] = A[min_], A[i]
# main
for i in range(len(A)):
  print(A[i])
```

Quick Sort (using list comprehension):

Quick Sort using list comprehension is a recursive algorithm for sorting an array of elements. It works by selecting a pivot element and partitioning the array around the pivot, such that all elements less than the pivot are moved to its left and all elements greater than the pivot are moved to its right. Then, it recursively applies the same process to the left and right sub-arrays until the entire array is sorted.

ALGORITHM:

1. If the input array has length 0 or 1, return the array as it is already sorted
2. Choose the first element of the array as the pivot element.
3. Create two empty lists, left and right.
4. For each element in the array except for the pivot:
5. If the element is smaller than the pivot, add it to the left list.
6. If the element is greater than or equal to the pivot, add it to the right list.
7. Recursively call quicksort on the left and right lists.
8. Concatenate the sorted left list, the pivot element, and the sorted right list.
9. Return the concatenated list.

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = [x for x in arr[1:] if x < pivot]
        right = [x for x in arr[1:] if x >= pivot]
        return quicksort(left) + [pivot] + quicksort(right)

# Example usage
arr = [1, 7, 4, 1, 10, 9, -2]
sorted_arr = quicksort(arr)
print("Sorted Array in Ascending Order:")
print(sorted_arr)
```

Merge Sort:

Merge sort is an algorithm that follows the Divide and Conquers paradigm. It continuously divides the array into two equal halves. Later it starts sorting the lists having a single element each and continuously merges the sorted lists to form the complete sorted list.

```
def merge(a1,a2):
  c=[]
  x=0
  y=0
  while(x<len(a1) and y<len(a2)):
    if(a1[x]<a2[y]):
      c.append(a1[x])
      x+=1
    else:
      c.append(a2[y])
      y+=1
  while(x<len(a1)):
    c.append(a1[x])
    x+=1
  while(y<len(a2)):
    c.append(a2[y])
    y+=1
  return c

def mergesort(array):
  if(len(array)==1):
    return array
  mid=(len(array))//2
  a1=mergesort(array[:mid])
  a2=mergesort(array[mid:])
  return merge(a1,a2)
array=[2,3,1,5,4,6,8,10,7,9]
print(mergesort(array))
```

Exercise:

Set A:

1. Write a Python program to append a new item to the end of the array.
2. Write a Python program to reverse the order of the items in the array.

3. Write a Python program to get the number of occurrences of a specified element in an array.
4. Write a Python program to calculate sum of all odd numbers in an array.

Set B:

1. Write a python program for linear search.
2. Write a Python program for binary search of an ordered list.
3. Write a Python program to sort a list of elements using the bubble sort algorithm.
4. Write a Python program to sort a list of elements using the merge sort algorithm.
5. Write a Python program to sort a list of elements using the insertion sort algorithm.

Set C:

1. Write a Python program to sort a list of elements using selection sort algorithm

Exercise 2: Linked List

Introduction:

A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer. Python does not have linked lists in its standard library. We implement the concept of linked lists using the concept of nodes as discussed in the previous chapter.

We have already seen how we create a node class and how to traverse the elements of a node. In this chapter we are going to study the types of linked lists known as singly linked lists. In this type of data structure there is only one link between any two data elements. We create such a list and create additional methods to insert, update and remove elements from the list.

Creation of Linked list:

A linked list is created by using the node class we studied in the last chapter. We create a Node object and create another class to use this ode object. We pass the appropriate values through the node object to point the to the next data elements. The below program creates the linked list with three data elements. In the next section we will see how to traverse the linked list.

```
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.

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

```
    self.nextval = None

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

  def listprint(self):
    printval = self.headval
    while printval is not None:
      print (printval.dataval)
      printval = printval.nextval

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

# Link first Node to second node
list.headval.nextval = e2

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

list.listprint()
```

Output

When the above code is executed, it produces the following result −

```
Mon
Tue
Wed
```

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.

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

```
class SLinkedList:
   def __init__(self):
      self.headval = None
# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval
   def AtBegining(self,newdata):
      NewNode                                                = Node(newdata)

# Update the new nodes next val to existing node
   NewNode.nextval = self.headval
   self.headval = NewNode

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtBegining("Sun")
list.listprint()
```

Output

When the above code is executed, it produces the following result −

```
Sun
Mon
Tue
Wed
```

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.

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

```
      self.headval = None
# Function to add newnode
   def AtEnd(self, newdata):
      NewNode = Node(newdata)
      if self.headval is None:
         self.headval = NewNode
         return
      laste = self.headval
      while(laste.nextval):
         laste = laste.nextval
      laste.nextval=NewNode
# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtEnd("Thu")

list.listprint()
```

Output

When the above code is executed, it produces the following result −

```
Mon
Tue
Wed
Thu
```

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.

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

# Function to add node
   def Inbetween(self,middle_node,newdata):
      if middle_node is None:
         print("The mentioned node is absent")
         return

      NewNode = Node(newdata)
      NewNode.nextval = middle_node.nextval
      middle_node.nextval = NewNode

# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Thu")

list.headval.nextval = e2
e2.nextval = e3

list.Inbetween(list.headval.nextval,"Fri")

list.listprint()
```

Output

When the above code is executed, it produces the following result −

```
Mon
Tue
Fri
Thu
```

Removing an Item:

We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted.Then, point the next pointer of this node to the next node of the node to be deleted.

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

  def Atbegining(self, data_in):
    NewNode = Node(data_in)
    NewNode.next = self.head
    self.head = NewNode

# Function to remove node
  def RemoveNode(self, Removekey):
    HeadVal = self.head

    if (HeadVal is not None):
      if (HeadVal.data == Removekey):
        self.head = HeadVal.next
        HeadVal = None
        return
    while (HeadVal is not None):
      if HeadVal.data == Removekey:
        break
      prev = HeadVal
      HeadVal = HeadVal.next

    if (HeadVal == None):
      return

    prev.next = HeadVal.next
    HeadVal = None

  def LListprint(self):
    printval = self.head
    while (printval):
      print(printval.data),
      printval = printval.next

llist = SLinkedList()
```

```
llist.Atbegining("Mon")
llist.Atbegining("Tue")
llist.Atbegining("Wed")
llist.Atbegining("Thu")
llist.RemoveNode("Tue")
llist.LListprint()
```

Output

When the above code is executed, it produces the following result −

```
Thu
Wed
Mon
```

Doubly Linked List:

In this chapter we see another type of linked list in which it is possible to travel both forward and backward. Such a linked list is called Doubly Linked List. Following is the features of doubly linked list.

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

Creating Doubly linked list

We create a Doubly Linked list by using the Node class. Now we use the same approach as used in the Singly Linked List but the head and next pointers will be used for proper assignation to create two links in each of the nodes in addition to the data present in the node.

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

class doubly_linked_list:
   def __init__(self):
      self.head = None

# Adding data elements
   def push(self, NewVal):
      NewNode = Node(NewVal)
      NewNode.next = self.head
      if self.head is not None:
         self.head.prev = NewNode
```

```
    self.head = NewNode

# Print the Doubly Linked list
  def listprint(self, node):
    while (node is not None):
      print(node.data),
      last = node
      node = node.next

dllist = doubly_linked_list()
dllist.push(12)
dllist.push(8)
dllist.push(62)
dllist.listprint(dllist.head)
```

Output

When the above code is executed, it produces the following result −

62 8 12

Inserting into Doubly Linked List:

Here, we are going to see how to insert a node to the Doubly Link List using the following program. The program uses a method named insert which inserts the new node at the third position from the head of the doubly linked list.

```
# Create the Node class
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None
    self.prev = None

# Create the doubly linked list
class doubly_linked_list:
  def __init__(self):
    self.head = None

# Define the push method to add elements
  def push(self, NewVal):
    NewNode = Node(NewVal)
    NewNode.next = self.head
    if self.head is not None:
      self.head.prev = NewNode
    self.head = NewNode

# Define the insert method to insert the element
```

```
  def insert(self, prev_node, NewVal):
    if prev_node is None:
      return
    NewNode = Node(NewVal)
    NewNode.next = prev_node.next
    prev_node.next = NewNode
    NewNode.prev = prev_node
    if NewNode.next is not None:
      NewNode.next.prev = NewNode

# Define the method to print the linked list
  def listprint(self, node):
    while (node is not None):
      print(node.data),
      last = node
      node = node.next

dllist = doubly_linked_list()
dllist.push(12)
dllist.push(8)
dllist.push(62)
dllist.insert(dllist.head.next, 13)
dllist.listprint(dllist.head)
```

Output

When the above code is executed, it produces the following result −

62 8 13 12

Appending to a Doubly linked list:

Appending to a doubly linked list will add the element at the end.

```
# Create the node class
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None
    self.prev = None
# Create the doubly linked list class
class doubly_linked_list:
  def __init__(self):
    self.head = None

# Define the push method to add elements at the begining
  def push(self, NewVal):
    NewNode = Node(NewVal)
    NewNode.next = self.head
```

```
    if self.head is not None:
      self.head.prev = NewNode
    self.head = NewNode

# Define the append method to add elements at the end
  def append(self, NewVal):
    NewNode = Node(NewVal)
    NewNode.next = None
    if self.head is None:
      NewNode.prev = None
      self.head = NewNode
      return
    last = self.head
    while (last.next is not None):
      last = last.next
    last.next = NewNode
    NewNode.prev = last
    return

# Define the method to print
  def listprint(self, node):
    while (node is not None):
      print(node.data),
      last = node
      node = node.next

dllist = doubly_linked_list()
dllist.push(12)
dllist.append(9)
dllist.push(8)
dllist.push(62)
dllist.append(45)
dllist.listprint(dllist.head)
```

Output

When the above code is executed, it produces the following result −

62 8 12 9 45

Exercise:

Set A:

1. Write a Python program to create a singly linked list, append some items and iterate through the list.

2. Write a Python program to search a specific item in a singly linked list and return true if the item is found otherwise return false.
3. Write a Python program to delete the first item from a singly linked list.
4. Write a Python program to delete the last item from a singly linked list.

Set B:

1. Write a Python program to create a doubly linked list, append some items and iterate through the list (print forward).
2. Write a Python program to create a doubly linked list and print nodes from current position to first node.
3. Write a Python program to count the number of items of a given doubly linked list.
4. Write a Python program to insert an item in front of a given doubly linked list.
5. Write a Python program to search a specific item in a given doubly linked list and return true if the item is found otherwise return false.
6. Write a Python program to delete a specific item from a given doubly linked list.

Set C:

1. Write a Python program to print a given doubly linked list in reverse order.

Exercise 3: Stack

Introduction:
In the context of data structures, a stack is an abstract data type that follows the Last-In-First-Out (LIFO) principle. It is a collection of elements with two primary operations: push and pop.
In Python, you can implement a stack using various data structures such as lists or collections.dequeue.

1) **Stack implementation using a list:**

```
class Stack:
   def _ _init_ _(self):
self.stack = []

   def push(self, item):
self.stack.append(item)

   def pop(self):
      if not self.is_empty():
         return self.stack.pop()
      else:
print("Stack is empty.")  # raise IndexError("pop from an empty stack")
         return None

   def is_empty(self):
      return len(self.stack) == 0

   def peek(self):
      if not self.is_empty():
         return self.stack[-1]
      else:
print("Stack is empty.")
         return None

   def size(self):
      return len(self.stack)
```

In this implementation, the Stack class has an underlying list (self.stack) to store the elements.
Let's break down the methods:

1) **_ _init_ _:** Initializes an empty list to store the elements of the stack.
2) **is_empty:** Returns True if the stack is empty; otherwise, returns False
3) **push:** Adds an item to the top of the stack.
4) **pop:** Removes and returns the top item from the stack. Raises an IndexError if the stack is empty
5) **peep:** Returns the top item from the stack without removing it. Raises an IndexError if the stack is empty.

6) **size:** Returns the number of elements currently in the stack.

You can use this Stack class to perform standard stack operations in Python. For example:

```
my_stack = Stack()

my_stack.push(1)
my_stack.push(2)
my_stack.push(3)
print("Top item:", my_stack.peek())  # Output: Top item: 3
popped_item = my_stack.pop()
print("Popped item:", popped_item)   # Output: Popped item: 3
print("Is stack empty?", my_stack.is_empty())  # Output: Is stack empty? False
print("Size of stack:", my_stack.size())  # Output: Size of stack: 2
```

In the example above, we create a stack (my_stack), push three elements onto it, and then demonstrate various stack operations.

2) Stack implementation using a Collection:

You can implement a stack using the deque class from the collections module in Python. The deque class provides the necessary methods to perform stack operations efficiently. Here's an example of implementing a stack using deque:

```
from collections import deque
class Stack:
    def __init__(self):
self.stack = deque()

    def push(self, item):
self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        return None

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

    def top(self):
        if not self.is_empty():
            return self.stack[-1]
        return None
```

```
# Test the stack implementation
stack = Stack()

stack.push(1)
stack.push(2)
stack.push(3)

print("Stack size:", stack.size())
print("Top element:", stack.top())

while not stack.is_empty():
print("Popped element:", stack.pop())
```

Output
Stack size: 3
Top element: 3
Popped element: 3
Popped element: 2
Popped element: 1

In this implementation, the Stack class uses the deque object to store the stack elements. The push method appends an item to the end of the deque, simulating the behavior of pushing onto a stack. The pop method removes and returns the last item from the deque, representing the behavior of popping from a stack. The is_empty method checks if the deque is empty, and the size method returns the current size of the stack. The top method returns the top element of the stack without removing it.

You can use the Stack class just like a regular stack, pushing items onto it, popping items from it, checking if it's empty, getting the top element, and retrieving its size.

Static Implementation of Stack

If you want to implement a static stack with a fixed size in Python, you can modify the previous implementation by adding a maximum size parameter. Here's an example:

```
class StaticStack:
    def __init__(self, max_size):
self.max_size = max_size
self.stack = [None] * max_size
self.top = -1

    def is_empty(self):
        return self.top == -1

    def is_full(self):
        return self.top == self.max_size - 1
```

```
    def push(self, item):
        if self.is_full():
            raise IndexError("push to a full stack")
self.top += 1
self.stack[self.top] = item

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        item = self.stack[self.top]
self.stack[self.top] = None
self.top -= 1
        return item

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from an empty stack")
        return self.stack[self.top]

    def size(self):
        return self.top + 1
```

In this modified implementation, we add the max_size parameter to the __init__ method, which specifies the maximum number of elements the stack can hold. We create a list with max_size None values to represent the stack.
We also add two new methods: is_full and size. The is_full method checks if the stack is full, and the size method returns the current number of elements in the stack

Here's an example of how to use the StaticStack class:

```
my_stack = StaticStack(3)
my_stack.push(1)
my_stack.push(2)
my_stack.push(3)
print("Top item:", my_stack.peek())  # Output: Top item: 3
popped_item = my_stack.pop()
print("Popped item:", popped_item)   # Output: Popped item: 3
print("Is stack empty?", my_stack.is_empty())  # Output: Is stack empty? False
print("Is stack full?", my_stack.is_full())    # Output: Is stack full? False
print("Size of stack:", my_stack.size())  # Output: Size of stack: 2
```

Dynamic Implementation of Stack
In a dynamic implementation of a stack, the stack can grow or shrink as needed. We can achieve this by using a dynamic data structure like a Python list, which can automatically resize itself as elements are added or removed. Here's an example of a dynamic stack implementation in Python:

```
class Dynamicstack:
   def __init__(self):
self.stack = []
self.top = -1

   def is_empty(self):
      return self.top == -1

   def push(self, item):
self.stack.append(item)
self.top += 1

   def pop(self):
      if self.is_empty():
         raise indexerror("pop from an empty stack")
      item = self.stack.pop()
self.top -= 1
      return item

   def peek(self):
      if self.is_empty():
         raise indexerror("peek from an empty stack")
      return self.stack[self.top]

   def size(self):
      return self.top + 1
```

In this dynamic implementation, we use a Python list as the underlying data structure for the stack. The list can grow and shrink automatically as elements are added or removed.

Here's an example of how to use the Dynamic Stack class

```
my_stack = Dynamicstack()
my_stack.push(1)
my_stack.push(2)
my_stack.push(3)
print("top item:", my_stack.peek())  # output: top item: 3
popped_item = my_stack.pop()
print("popped item:", popped_item)   # output: popped item: 3
print("is stack empty?", my_stack.is_empty())  # output: is stack empty? false
print("size of stack:", my_stack.size())  # output: size of stack: 2
```

Reversing each word of the String

To reverse a string using a stack data structure in Python, you can follow these steps:

1) Define an empty stack.
2) Iterate through each character in the string.

3) Push each character onto the stack.
4) Create an empty string to store the reversed string.
5) Pop each character from the stack and append it to the reversed string.
6) Return the reversed string.

Evaluate Postfix Expression

To evaluate a postfix expression using a stack in Python, you can follow these steps:

1) Create an empty stack.
2) Iterate through each character in the postfix expression.
3) If the character is an operand (number), push it onto the stack.
4) If the character is an operator, pop the top two elements from the stack.
5) Perform the operation on the two popped elements using the operator.
6) Push the result back onto the stack.
7) After iterating through all the characters, the final result will be the only element left in the stack.
8) Return the result.

Check whether the string is Palindrome or not using stack

To check whether a string is a palindrome or not using a stack, you can follow these steps:

1) Create an empty stack.
2) Iterate through each character in the string and push them onto the stack.
3) Create an empty string to store the reversed string.
4) Pop each character from the stack and append it to the reversed string.
5) Compare the reversed string with the original string.
6) If they are the same, the string is a palindrome. Otherwise, it is not.

SET A:

1) Write a Python program to implement Static implementation of stack of integers. (using list/ collection.dequeue)
2) Write a Python program to implement Dynamic implementation of stack of integers
3) Write a Pyhton program to reverse each word of the string by using static and dynamic implementation of stack.

   Example: Input - This is a program in python
   Output – sihTsi a margorpninohtyp

SET B:

1) Write a Python program to read a postfix expression, evaluate it and display the result. (Use Static/Dynamic implementation of Stack).
2) Write a Python program to accept an infix expression, convert it into its equivalent postfix expression and display the result.(Use Static/Dynamic implementation of Stack).
3) Write a Python program which accepts the string and check whether the string is Palindrome or not using stack. (Use Static/Dynamic implementation of Stack).

SET C:

1) Write a Python program to check whether the contents of two stacks are identical.
2) Write a Python program that copies the contents of one stack into another. The order of two stacks must be identical.(Hint: Use a temporary stack to preserve the order).
3) Write a Python program to accept an infix expression, convert it into its equivalent prefix expression and display the result. (Use Static/Dynamic implementation of Stack).

Exercise 4: Queue

Introduction queue

Queues are a fundamental data structure that follows the First-In-First-Out (FIFO) principle. In a queue, elements are added at the rear and removed from the front, resembling a real-life queue or line.

In Python, a queue using built-in data structures or by creating a custom class.

1. **Queue Implementation using Lists:**

   Python lists provide a simple way to implement a queue. The append() function can be used to enqueue elements at the rear of the list, while the pop(0) function removes the front element.

   Example:

```
# Create an empty queue
    queue = []

# Enqueue elements
    queue.append(10)
    queue.append(20)
    queue.append(30)

# Dequeue elements
    front = queue.pop(0)
    print(front)

 # Output: 10
```

   However, using pop(0) to dequeue elements from the front of the list is not efficient because it requires shifting all subsequent elements. A more efficient approach is to use the collections.deque class.

2. **Queue Implementation using collections.deque:**

   The collections module in Python provides a deque class, which stands for "double-ended queue." It allows for efficient appending and popping of elements from both ends.

   Example:

```
from collections import deque

# Create an empty queue
```

```
queue = deque()

# Enqueue elements
queue.append(10)
queue.append(20)
queue.append(30)

# Dequeue elements
front = queue.popleft()
print(front)

# Output: 10
```

The popleft() function removes and returns the element from the left side of the deque, corresponding to the front of the queue.

3. **Custom Queue Class:**

If you prefer a more customized implementation, you can create a Queue class using a linked list or any other underlying data structure. Here's a basic example using a linked list:

Example:

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

class Queue:
    def __init__(self):
    self.front = None
    self.rear = None

defis_empty(self):
    returnself.front is None

def enqueue(self, data):
    new_node = Node(data)
ifself.is_empty():
self.front = new_node
self.rear = new_node
else:
self.rear.next = new_node
self.rear = new_node
```

```
defdequeue(self):
ifself.is_empty():
return None
front_data = self.front.data
        self.front = self.front.next
ifself.front is None:
self.rear = None
        returnfront_data
```

This implementation maintains two pointers, front and rear, which point to the first and last nodes of the linked list, respectively.

Static Implementation of Queue:

In a static implementation of a queue, the size of the queue is fixed and cannot be changed once it is defined.

Example:

```
class Queue:
    def __init__(self, capacity):
    self.capacity = capacity
    self.queue = [None] * capacity
    self.front = 0
    self.rear = -1
    self.size = 0

    defis_empty(self):
        returnself.size == 0

    defis_full(self):
        returnself.size == self.capacity

    defenqueue(self, item):
    ifself.is_full():
    print("Queue is full. Cannot enqueue item.")
    return
    self.rear = (self.rear + 1) % self.capacity
    self.queue[self.rear] = item
    self.size += 1

    defdequeue(self):
        ifself.is_empty():
        print("Queue is empty. Cannot dequeue item.")
```

```
            return None
            item = self.queue[self.front]
        self.queue[self.front] = None
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return item

    defget_front(self):
        ifself.is_empty():
        return None
            returnself.queue[self.front]

    defget_rear(self):
        ifself.is_empty():
        return None
            returnself.queue[self.rear]
```

In this implementation, the Queue class is initialized with a capacity parameter that determines the maximum number of elements the queue can hold. The queue itself is represented as an array (queue) with the specified capacity.

1. The front and rear variables keep track of the indices of the front and rear elements, respectively. The size variable tracks the current number of elements in the queue.
2. The is_empty() and is_full() methods check if the queue is empty or full, respectively.
3. The enqueue() method adds an item to the rear of the queue. It first checks if the queue is full to avoid overflow. If the queue is not full, it updates the rear index, inserts the item at that position in the array, and increments the size variable.
4. The dequeue() method removes and returns the item from the front of the queue. It checks if the queue is empty to avoid underflow. If the queue is not empty, it retrieves the item at the front index, sets that position in the array to None, updates the front index, decrements the size variable, and returns the item.
5. The get_front() and get_rear() methods return the front and rear elements of the queue, respectively, without removing them.

Dynamic Implementation Queue:

In a dynamic implementation of a queue, the size of the queue can grow or shrink dynamically as elements are added or removed. One common way to achieve this in Python is by using the list data structure.

Example:

```
class Queue:
    def __init__(self):
```

```
        self.queue = []

    defis_empty(self):
        returnlen(self.queue) == 0

    defenqueue(self, item):
        self.queue.append(item)

    defdequeue(self):
        ifself.is_empty():
print("Queue is empty. Cannot dequeue item.")
        return None
            returnself.queue.pop(0)

    defget_front(self):
        ifself.is_empty():
        return None
            returnself.queue[0]

    defget_rear(self):
        ifself.is_empty():
        return None
            returnself.queue[-1]
```

In this implementation, the Queue class is initialized with an empty list queue. The is_empty() method checks if the queue is empty by examining the length of the list.

1. The enqueue() method adds an item to the rear of the queue using the append() function of the list.
2. The dequeue() method removes and returns the item from the front of the queue using the pop(0) function. This operation has a time complexity of O(n) since it requires shifting all subsequent elements. If you frequently need to dequeue items from the front, you may consider using the collections.deque class instead for better performance.
3. The get_front() and get_rear() methods return the front and rear elements of the queue, respectively, without removing them.

Operations of Queue:

1. Enqueue: This operation adds an element to the rear of the queue.

Example:

```
queue = []  # Create an empty queue
queue.append(item)  # Enqueue item to the rear
```

2. Dequeue: This operation removes and returns the element from the front of the queue.

Example:

```
front = queue.pop(0)  # Dequeue item from the front
```

3. Front: This operation returns the element at the front of the queue without removing it.

Example:

```
front = queue[0]  # Get the front item
```

4. Rear: This operation returns the element at the rear of the queue without removing it.

Example:

```
rear = queue[-1]  # Get the rear item
```

5. Size: This operation returns the number of elements currently in the queue.

Example:

```
size = len(queue)  # Get the size of the queue
```

6. Is Empty: This operation checks if the queue is empty.

Example:

```
is_empty = len(queue) == 0  # Check if the queue is empty
```

Types of queue:

1. **Simple Queue:**

   This is the most basic type of queue where elements are inserted at the rear and removed from the front in a First-In-First-Out (FIFO) manner. It follows a straightforward implementation where enqueue and dequeue operations are performed.

   Example:

```
Implementing a simple queue in Python using a list:
class Queue:
        def __init__(self):
                self.queue = []

        defis_empty(self):
                returnlen(self.queue) == 0
```

```
        defenqueue(self, item):
            self.queue.append(item)

    defdequeue(self):
        ifself.is_empty():
        print("Queue is empty. Cannot dequeue item.")
            return None
            returnself.queue.pop(0)

    defget_front(self):
        ifself.is_empty():
        return None
            returnself.queue[0]

    def size(self):
        returnlen(self.queue)
```

In this implementation:

1. The Queue class is initialized with an empty list queue to hold the elements.
2. The is_empty() method checks if the queue is empty by examining the length of the list.
3. The enqueue() method adds an item to the rear of the queue using the append() function of the list.
4. The dequeue() method removes and returns the item from the front of the queue using the pop(0) function. If the queue is empty, an error message is printed, and None is returned.
5. The get_front() method returns the element at the front of the queue without removing it. If the queue is empty, None is returned.
6. The size() method returns the current number of elements in the queue by returning the length of the list.

You can create an instance of the Queue class and use its methods to perform enqueueing, dequeueing, and other queue operations:

```
queue = Queue()  # Create a new queue
queue.enqueue(10)  # Enqueue elements
queue.enqueue(20)
queue.enqueue(30)
print(queue.get_front())  # Output: 10
item = queue.dequeue()  # Dequeue an element
print(item)  # Output: 10

print(queue.size())
# Output: 2
```

2. **Circular Queue:**

A circular queue is a variation of the simple queue where the rear and front pointers wrap around to the beginning of the queue when they reach the end. This circular structure allows efficient utilization of memory and avoids the need to shift elements when performing enqueue and dequeue operations.

Example:

Implementing a circular queue in Python using a list:

```
classCircularQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0
        self.rear = 0
        self.size = 0

    defis_empty(self):
        returnself.size == 0

    defis_full(self):
        returnself.size == self.capacity

    defenqueue(self, item):
    ifself.is_full():
    print("Queue is full. Cannot enqueue item.")
    return
        self.queue[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity
        self.size += 1

    defdequeue(self):
        ifself.is_empty():
        print("Queue is empty. Cannot dequeue item.")
        return None
            item = self.queue[self.front]
            self.queue[self.front] = None
            self.front = (self.front + 1) % self.capacity
            self.size -= 1
            return item

    defget_front(self):
```

```
        ifself.is_empty():
        return None
            returnself.queue[self.front]

    defget_rear(self):
        ifself.is_empty():
        return None
            returnself.queue[(self.rear - 1) % self.capacity]
```

In this implementation:

1. The CircularQueue class is initialized with a capacity parameter that determines the maximum number of elements the circular queue can hold. The queue list is created with the specified capacity, and the front and rear indices are initially set to 0.
2. The is_empty() and is_full() methods check if the circular queue is empty or full, respectively, based on the size variable.
3. The enqueue() method adds an item to the rear of the circular queue. If the queue is full, an error message is printed, and the item is not added. Otherwise, the item is inserted at the rear index, and the rear index is updated by incrementing it modulo the capacity. The size variable is incremented.
4. The dequeue() method removes and returns the item from the front of the circular queue. If the queue is empty, an error message is printed, and None is returned. Otherwise, the item at the front index is retrieved, set to None, the front index is updated by incrementing it modulo the capacity, and the size variable is decremented.
5. The get_front() method returns the element at the front of the circular queue without removing it. If the queue is empty, None is returned.
6. The get_rear() method returns the element at the rear of the circular queue without removing it. If the queue is empty, None is returned.

You can create an instance of the CircularQueue class and use its methods to perform enqueueing, dequeueing, and other circular queue operations:

```
queue = CircularQueue(5)  # Create a new circular queue with capacity 5

queue.enqueue(10)  # Enqueue elements
queue.enqueue(20)
queue.enqueue(30)

print(queue.get_front())  # Output: 10
print(queue.get_rear())  # Output: 30

item = queue.dequeue()  # Dequeue an element
print(item)  # Output: 10

queue.enqueue(40)  # Enqueue another element
print(queue.get_rear())  # Output: 40
```

```
print(queue.is_empty())  # Output: False
print(queue.is_full())  # Output: False
```

SET A:

1. Write a python program to Implement Static implementation of Queue of integers
2. Write a python program to Implement Dynamic implementation of Queue of integers
3. Program to find the maximum element in a queue using list.

SET B:

1. Write a program to reverse the elements of a queue (Use Static &Dynamic implementation of Queue)
2. Write a python program to Implement circular queue of integers using list.

SET C:

1. Write a python Program to find the size of a queue without using the len() function.
2. Write a python program to Implement a circular queue using a list or create a custom CircularQueue class.

Exercise 5: Introduction to Tree

Definition

A binary search tree is a binary tree, which is either empty or in which each node contains a key that satisfies following conditions:

1) For every node X, in the tree the values of all the keys in its left subtree are smaller than the key value in X.
2) For every node X, in the tree the values of all the keys in its right subtreeare larger than the key value in X.

Representation of Binary Tree:

1. **Static Representation of Binary Tree:**

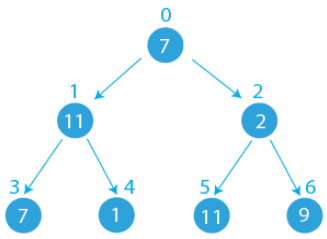
One of the way to represent binary tree using array is to store nodes level by level, starting from the zero level where the root is present. Such representation needs sequential numbering of the nodes starting with nodes on level zero, then those on level 1 and so on. A complete binary tree of height h has $(2^{h+1} - 1)$ nodes in it. The nodes can be stored in one dimensional array, TREE, with the node numbered at location TREE(i). A array of size $(2^{h+1} - 1)$ or $2^d - 1$ (where d = no. of levels) is required.

The root node is stored in the first memory location as the first element in the array. Following rules can be used to decide the location of any i^{th} node of a tree:

For any node with index i, $1 \leq i \leq n$;

a) PARENT (i) = $(i - 1)/2$ if $i \neq 1$ If i = 0 then it is root which has no parent
b) LCHILD (i) = (2 * i + 1)
c) RCHILD(i) = (2 * i + 2)

Example: Consider the given Binary tree:



The representation of the above binary tree using array is as followed:

2. Linked Representation of Binary Tree

Another way to represent a binary tree is linked list, which is more memory efficient than the array representation. All nodes should be allocated dynamically. Each node with data and link fields. The root pointer points to the tree in memory. Each node consists of three fields, Lchild, Data and Rchild

Data that represents value stored in the node.
Lchild that represents the pointer to the left child.
Rchild that represents the pointer to the right child.

Implementation

1) Create a Node

```
class Node:
   def __init__(self,data):
      #Assign data to the new node, set left and right children to None

      self.data = data;
      self.left = None;
      self.right = None;
```

2) create & Insert Binary search tree

```
classBinaryTree:
        def createNode(self,data):
                return Node(data)
        def insert(self,node,data):
                if node is None:
                        self.createNode(data)
                if data <node.data :
                        node.left = self.insert(node.left,data)
                else:
                        node.right = self.insert(node.right,data)
                return node
```

3) Inorder Traversing

```
        def traverse_Inorder(self, root):
                if root ==  None:
                        return
```

```
        self.traverse_Inorder (root.left)
        print(root.data)
        self.traverse_Inorder (root.right)
```

4) Preorder Traversing

```
    def traverse_preorder(self, root):
        if root ==  None:
            return
        print(root.data)
        self.traverse_preorder (root.left)
        self.traverse_preorder (root.right)
```

5) Postorder Traversing

```
    defraverse_postorder(self, root):
        if root ==  None:
            return
        self.traverse_postorder (root.left)
        self.traverse_postorder (root.right)
        print(root.data)
```

Set A

a) Implement a Binary search tree (BST) with operations – create, search, insert, inorder, preorder and postorder.
b) Create a Binary tree. Check whether it is a BST or not.
c) Create a Binary Search tree for string data.
d) Implement a Binary search tree (BST) with operations – create, count the total nodes and total leaf nodes in the tree.

Set B

a) Write a Python program which uses Binary search tree library and implements following function with recursion:
   def copy(root) : – create another BST which is exact copy of BST which is passed as parameter.
   def compare(T1, T2): – compares two binary search trees and returns 1 if they are equal and 0 otherwise.
b) Write a function to delete an element from BST.
c) Write a function print all paths from the root node to every leaf node of BST.

Set C

a) Write a Python program which uses Binary search tree library and implements following two functions:

defsumodd(T) – returnssum of all odd numbers from BST
defsumeven(T) – returnssum of all even numbers from BST
defmirror(T) – converts given tree into its mirror image.

b) What modifications are required in search function to count the number of comparisons required?

Assignment Evaluation

0:NotDone		1:Incomplete		2:LateComplete	
3:NeedsImprovement		4:Complete		5:WellDone	

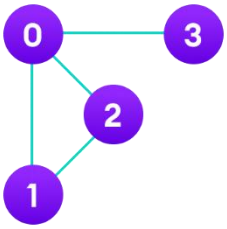
Practical Incharge :
Date :.

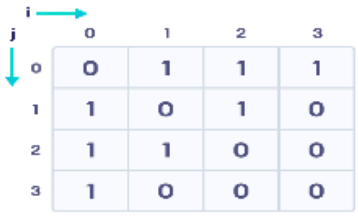
Exercise 6: Graph

Introduction:

A graph consists of a set of vertices and a set of edges. We can write a graph as G=(V,E), where V is a set of nodes (vertices) and E is a set of edges (arcs).

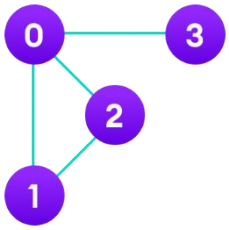
The one way of representing graphs is adjacency matrix representation. In adjacency matrix representation of a Graph with n vertices and e edges, a two dimensional n x n array , say a , is used , with the property that a[i,j] equals 1 if there is an edge from i to j and a[i,j] equals 0 if there is no edge from i to j.

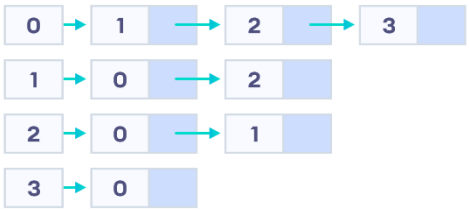




The usual operations on graph are:
Indegree(i) – returns the indegree (the number of edges ending on) of the i^{th}vertex
Outdegree(i) – returns the outdegree(the number of edges moving out) of the i^{th} vertex)
displayAdjMatrix – displays the adjacency matrix for the graph

Another way of representing graphs is adjacency list representation. In adjacency list representation of a graph with n vertices and e edges, there are n linked lists, one list for each vertex in the graph.





To implement graph as an adjacency list, we create an array of lists.
The usual operations on graph are:
Indegree(i) – returns the indegree (the number of edges ending on) of the i^{th} vertex
Outdegree(i) – returns the outdegree(the number of edges moving out) of the i^{th}vertex)
displayAdjList – displays the adjacency list for the graph

Adjacency Matrix representation in Python	Adjacency List representation in Python

class Graph(object):
  # Initialize the matrix
def __init__(self, size):
self.adjMatrix = []
  for i in range(size):
self.adjMatrix.append([0 for i in range(size)])
self.size = size
  # Add edges
defadd_edge(self, v1, v2):
  if v1 == v2:
  print("Same vertex %d and %d" % (v1, v2))
self.adjMatrix[v1][v2] = 1
self.adjMatrix[v2][v1] = 1
  # Remove edges
defremove_edge(self, v1, v2):
  if self.adjMatrix[v1][v2] == 0:
  print("No edge between %d and %d" % (v1, v2))
  return
self.adjMatrix[v1][v2] = 0
self.adjMatrix[v2][v1] = 0
def __len__(self):
  return self.size
  # Print the matrix
defprint_matrix(self):
  for row in self.adjMatrix:
  for val in row:
  print('{:4}'.format(val)),
  print
def main():
  g = Graph(5)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.print_matrix()
if __name__ == '__main__':
  main()
	class AdjNode:
	def __init__(self, value):
	self.vertex = value
	self.next = None
	class Graph:
	def __init__(self, num):
	self.V = num
	self.graph = [None] * self.V
	  # Add edges
	defadd_edge(self, s, d):
	  node = AdjNode(d)
	node.next = self.graph[s]
	self.graph[s] = node
	  node = AdjNode(s)
	node.next = self.graph[d]
	self.graph[d] = node
	  # Print the graph
	defprint_agraph(self):
	  for i in range(self.V):
	  print("Vertex " + str(i) + ":", end="")
	  temp = self.graph[i]
	  while temp:
	  print(" -> {}".format(temp.vertex), end="")
	  temp = temp.next
	  print(" \n")
	if __name__ == "__main__":
	  V = 5
	  # Create graph and edges
	  graph = Graph(V)
	graph.add_edge(0, 1)
	graph.add_edge(0, 2)
	graph.add_edge(0, 3)
	graph.add_edge(1, 2)
	graph.print_agraph()

Set A

a) Write a Python program that accepts the vertices and edges of a graph and stores it as an adjacency matrix. Display the adjacency matrix.

b) Write a Python program that accepts the vertices and edges of a graph and store it as an adjacency matrix. Implement functions to print indegree, outdegree and total degree of all vertices of graph.

c) Write a Python program that accepts the vertices and edges of a graph. Create adjacency list and display the adjacency list.

d) Write a Python program that accepts the vertices and edges of a graph. Create adjacency list. Implement functions to print indegree, outdegree and total degree of all vertex of graph.

Set B

a) Write a Python program that accepts the vertices and edges of a graph and store it as an adjacency matrix. Implement function to traverse the graph using Breadth First Search (BFS) traversal.

b) Write a Python program that accepts the vertices and edges of a graph and store it as an adjacency matrix. Implement function to traverse the graph using Depth First Search (DFS) traversal.

c) Write a Python program that accepts the vertices and edges of a graph and store it as an adjacency list. Implement function to traverse the graph using Breadth First Search (BFS) traversal.

d) Write a Python program that accepts the vertices and edges of a graph and store it as an adjacency list. Implement function to traverse the graph using Depth First Search (DFS) traversal.

Set C

a) Write a Python program for the implementation of Topological sorting.

b) Write a Python program for the Implementation of Prim's Minimum spanning tree algorithm

c) Write a Python program for the Implementation of Kruskal's Minimum spanning tree algorithm.

Assignment Evaluation

0:NotDone		1:Incomplete		2:LateComplete	
3:NeedsImprovement		4:Complete		5:WellDone	

Practical Incharge :
Date :