180 Practical - 6

* Aim: To demonstrate Queue add and delete.

* Theory: Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT.
Front points to the beginning of the queue and rear points to the end of the queue.
Queue follows the FIFO (First-In-First-Out) structure.
According to its FIFO structure, element inserted first will also be removed first.
In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue) because queue is open on both of its end.
Enqueue() can be termed as add() in queue i.e. adding a element in queue.
Dequeue() can be termed as delete or Remove i.e. deleting or removing of element.

035

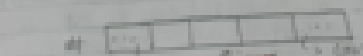
Program

Output

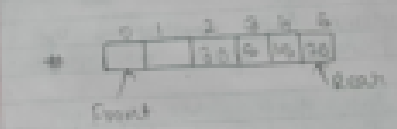
```
Process returned 0 (0x0)   execution time : 0.000 s
Queue is full
10
20
30
40
50
Queue is empty
```

Front is used to get the front data item from a queue.

Rear is used to get the last item from a queue.



On which index have ends.



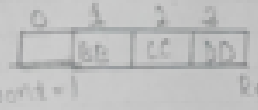
Front = 2
Rear = 5

Practical - 6

* Aim: To demonstrate the use of circular queue in data structure.

* Theory: The queue that we implemented using an array suffers from one limitation. In that implementation there is a possibility that the queue is reported as full (since rear has reached the end of the array) even though it is actually there might be empty slot at the beginning of the queue.
To overcome this limitation we can implement queue as circular queue.
In circular queue we go on adding the element to the queue.

Example:



```
def remove(self):
    [illegible]
    if self.f==[illegible]:
        print("Data removed",self.l[self.f])
        self.f=self.f+1
    else:
        [illegible]
        self.f=[illegible]
        if self.f!=self.r:
            print(self.l[self.f])
            self.f=self.f+1
        else:
            print("Queue is empty")
            self.f=[illegible]
Q=Queue()
Q.add(44)
Q.add(55)
Q.add(66)
Q.add(77)
Q.add(88)
Q.add(99)
Q.remove()
Q.add(66)
```

Output:

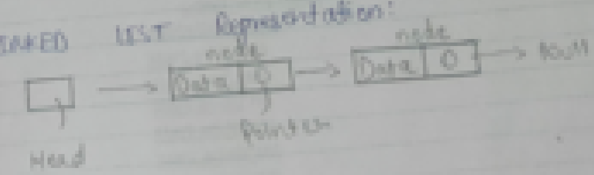
```
==================== RESTART: C:/Users/Sampath/Desktop/Ds/ds4.py ====================
Pranay 1743
data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
data removed: 44
>>>
```

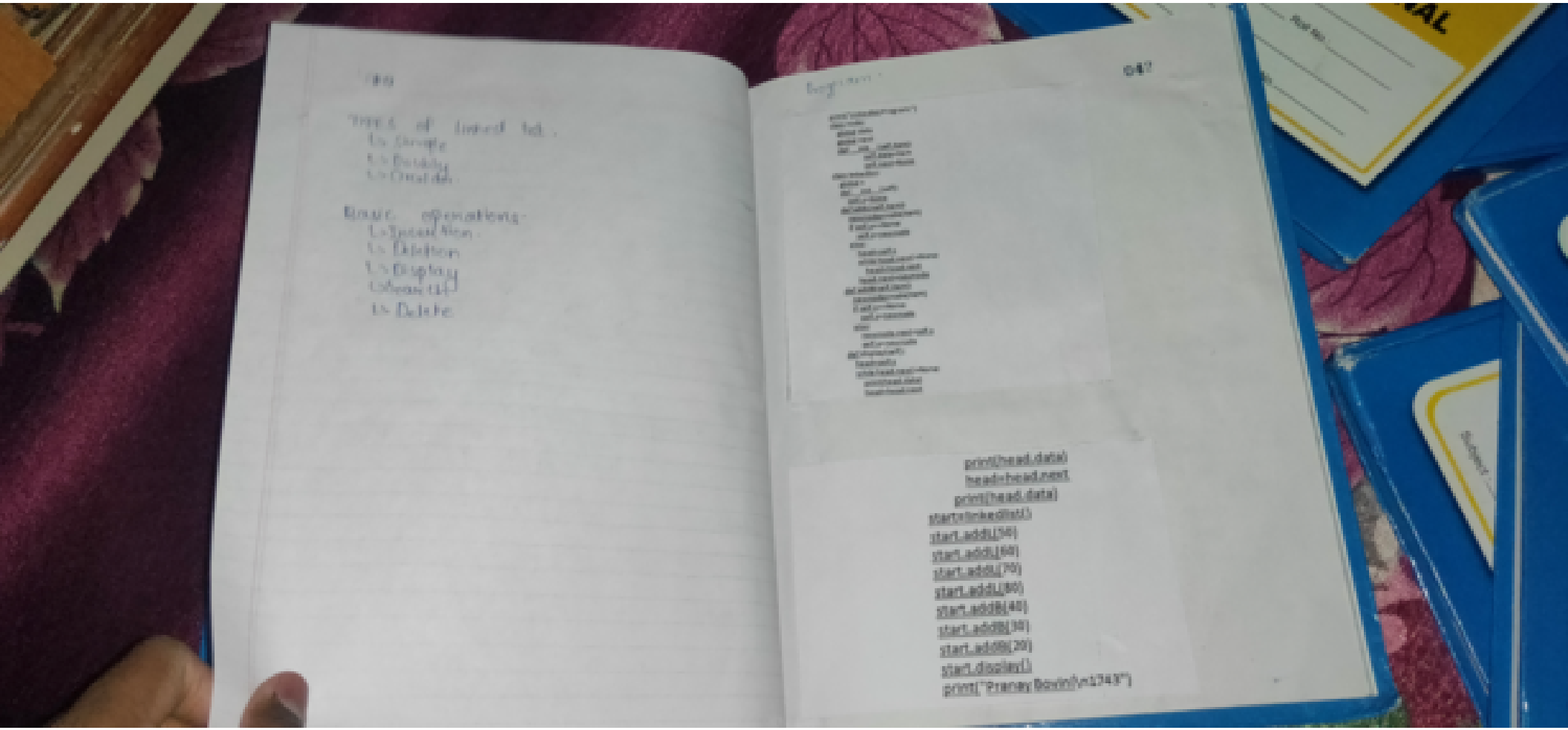

Practical-4

* Aim:- To demonstrate the use of linked list in data structure.

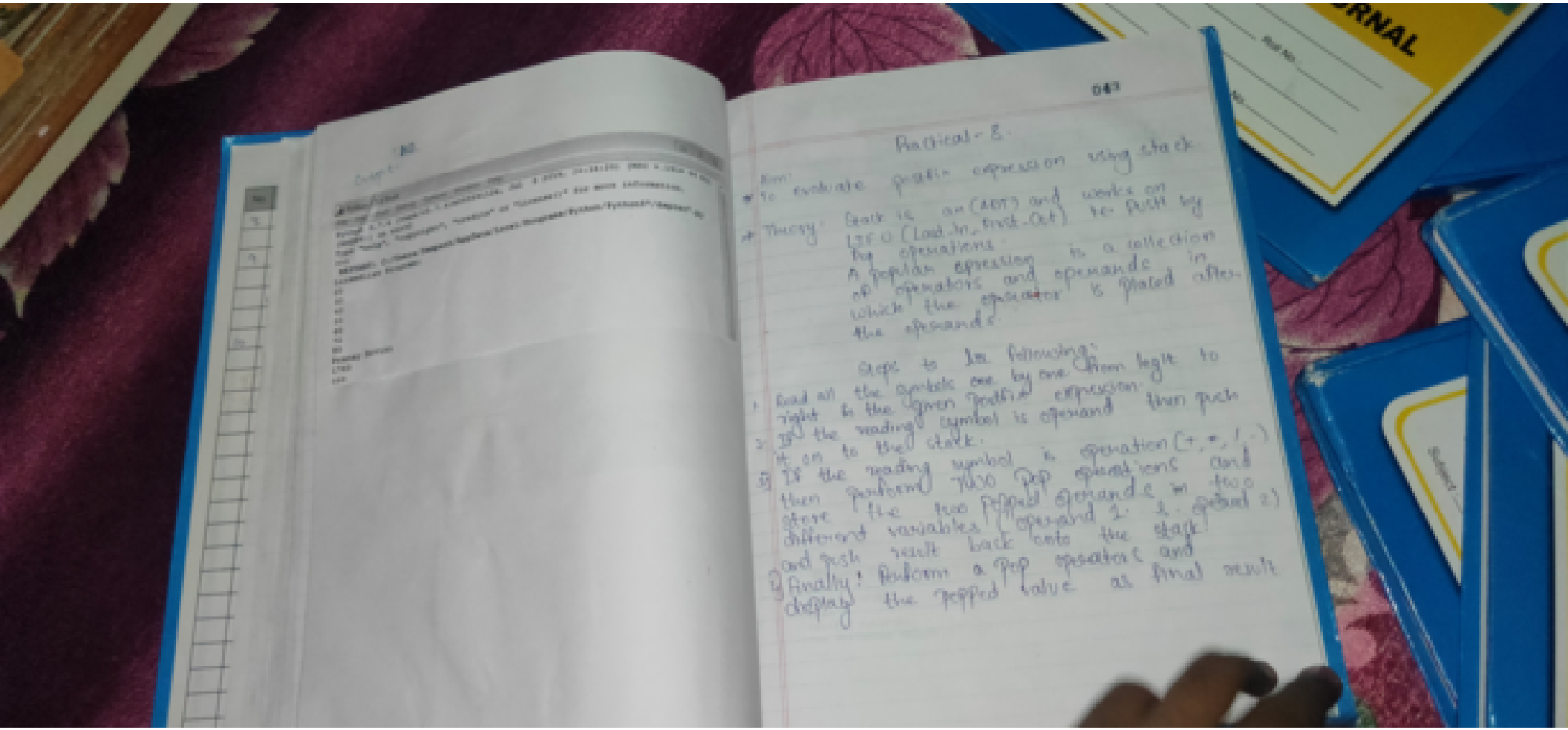
* Theory: A linked data is a sequence of data structures. Linked list is a a sequence of links which contains items. Each link contains a connection to another link.
  - Link - Each link of a linked list can store a data called an element.
  - NEXT - Each link of a linked list contains a link to the next link called NEXT.
  - LINKED - A linked list contains the connection link to the first link called First.

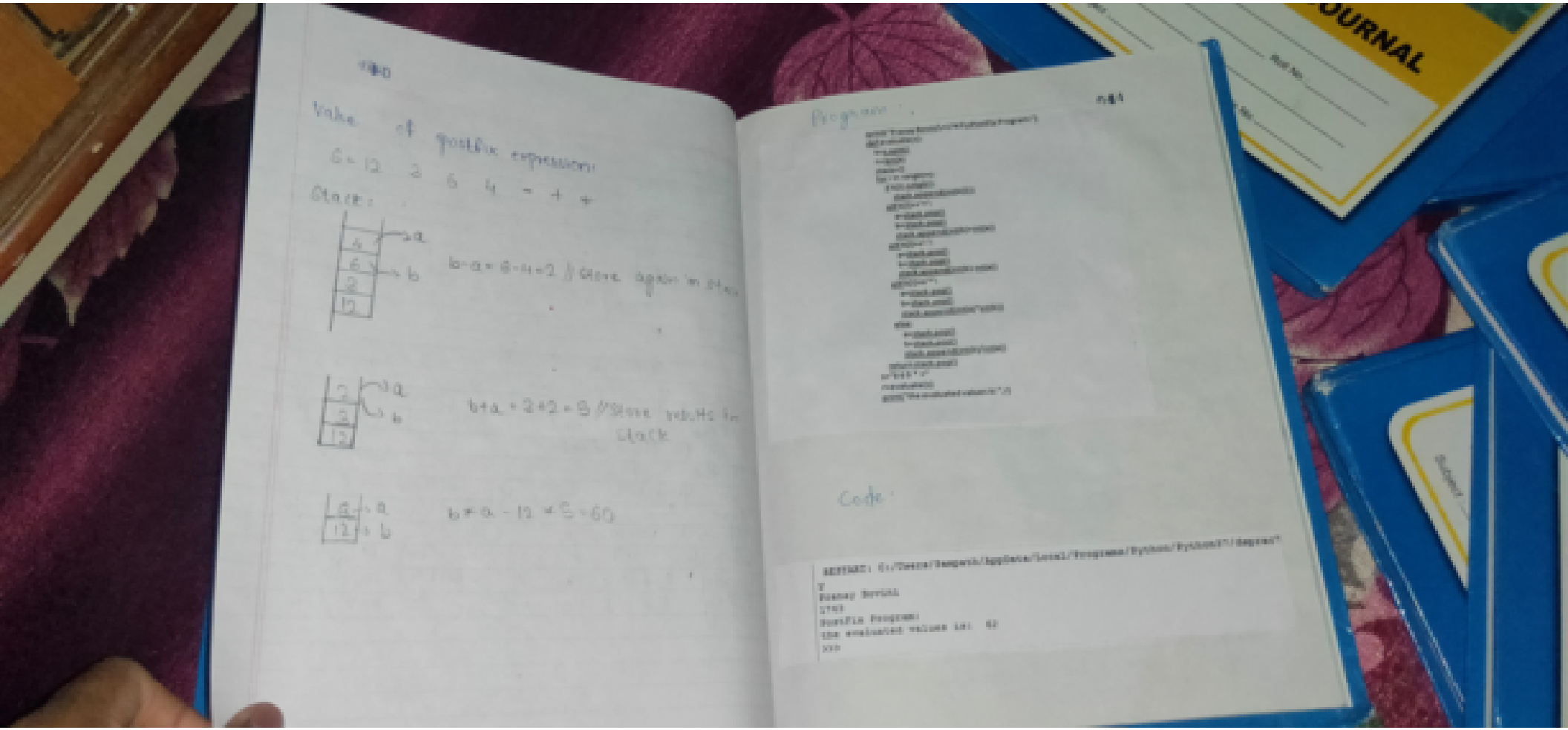
LINKED LIST Representation:





```
        print(head.data)
        head=head.next
    print(head.data)
start=linkedlist()
start.add1(50)
start.add1(60)
start.add1(70)
start.add1(80)
start.add8(40)
start.add8(30)
start.add8(20)
start.display()
print("Pranay Bovin\n1743")
```


043
Practical - 8.
Aim:
To evaluate postfix expression using stack.
Theory:
Stack is an (ADT) and works on
LIFO (Last In, First Out)
the operations.
A postfix expression is a collection
of operators and operands in
which the operator is placed after
the operands.
Steps to be following:
1. Read all the symbols one by one from left to
right in the given postfix expression.
2. If the reading symbol is operand then push
it on to the stack.
3. If the reading symbol is operation (+, *, /, -)
then perform two pop operations and
store the two popped operands in two
different variables (operand 1, operand 2)
and push result back onto the stack.
4. Finally! Perform a pop operation and
display the popped value as final result.


JOURNAL
Value of postfix expression:
Stack:
Program:
Code:

Practical - 9

Aim: To sort given random data by using bubble sort.

Theory: SORTING is type in which any random data is sorted / arranged in ascending or descending order. BUBBLE sort sometimes referred to as sinking sort.
It is simple sinking algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in wrong order. The pass through the list is repeated until the list is sorted. The algorithm which is a comparison sort is named for the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow as it compares one element only checks if condition fails then swaps otherwise goes on.

Example:
First pass
(5 2 6 3 9) -> (2 5 6 3 9)
Here algorithm compares the first two elements and swaps since 5>2
(2 5 6 3 9) -> (2 5 6 3 9)

([illegible] 6 9) → (2 5 3 6 9) Swap since 5>2.

([illegible]) → (2 5 3 6 9) Now since these elements are already in order (6 < 9) algorithm doesn't swap them.

Second pass:

(2 5 3 6 9) → (2 5 3 6 9)

(2 5 3 6 9) → (2 3 5 6 9) ∵ 5>2

(2 3 5 6 9) → (2 3 5 6 9)

Third pass:

(2 3 5 6 9) - It checks and gives the data in sorted order.

* Program :-

```
print("Pranay Bovini\nRoll No:1743")
a=[31,12,17,27,16]
print(a)

for i in range(len(a)-1):
    for j in range(len(a)-1):
        if(a[j]>a[j+1]):
            t=a[j]
            a[j]=a[j+1]
            a[j+1]=t
print(a)
```

* Output :-

```
RESTART: D:/Users/Sampath/AppData/Local/Programs/Python...
...py
Pranay Bovini
Roll No:1743
[31, 12, 17, 27, 16]
[12, 16, 17, 27, 31]
>>>
```

Program:-

```
print("[illegible] Roll No [illegible]")
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)
def quickSortHelper(alist,first,last):
   if first<last:
      splitpoint=partition(alist,first,last)
      quickSortHelper(alist,first,splitpoint-1)
      quickSortHelper(alist,splitpoint+1,last)
def partition(alist,first,last):
   pivotvalue=alist[first]
   leftmark=first+1
   rightmark=last
   done=False
   while not done:
      while leftmark<=rightmark and alist[leftmark]<=pivotvalue:
         leftmark=leftmark+1
      while alist[rightmark]>=pivotvalue and rightmark>=leftmark:
         rightmark=rightmark-1
      if rightmark<leftmark:
         done=True
      else:
         temp=alist[leftmark]
         alist[leftmark]=alist[rightmark]
         alist[rightmark]=temp
   temp=alist[first]
   alist[first]=alist[rightmark]
   alist[rightmark]=temp
   return rightmark
alist=[31,41,45,64,32,76,31,56,80]
quickSort(alist)
print(alist)
```


Practical- 10.

\# Aim: To 'evaluate' i.e. to sort the given data in Quick Sort.

\# Theory: Quicksort is an efficient sorting algorithm. Type of a divide & conquer algorithm. It picks an element as pivot and partitions the given array around the picket point. There are many different versions of quick sort that pick pivot in different ways.

1] Always pick first elements as pivot.
2] Always pick last elements as pivot.
3] Pick a random element as pivot.
4] Pick median as pivot.

The key process in quicksort in partition (). Target of partitions in given an array and an element x of array as pivot, put x at in current position in sorted array and put all smaller elements (smaller than x) before x, & put all greater elements (greater than x) after x. All this should be done in linear time.

* Output :-

```
RESTART: C:/Users/Sampath/AppData/Local/Programs/Python[illegible]
Y-BY
[illegible]
 Roll No:1743
[31, 91, 45, 51, 66, [illegible], [illegible], 56, 80]
>>>
```

Program:-

Output:-

* MERGE SORT METHOD

THE UNSORTED LIST :

THE MERGE SORTED LIST :

Practical 11

Aim:- To sort given data using merge sort.

Theory:- Merge sort uses the divide and conquer technique. In merge sort we divide the list into nearly equal sublist each of which are the again sublist each of which are then again divided into two sublists.

- We continue this procedure until there is only one element in the individual sublist.
- Once we have individual elements in the sublists, we consider these sublist as sub problems which can be solved. Since there is only one element in the sublist, the sublist is already sorted.
- Now while merging the sub problems, we compare the 2 sublist and create the combined list by comparing the element of the sublist. After comparison we place the element in their correct position in the original sublists.

Practical 13

Topic: Binary Tree

Theory: Binary Tree is a tree which supports maximum of 2 children for any node within the tree. Thus any particular node can have either 0 or 1 or 2 children.
Leaf Node: Nodes which do not have any children.
Internal Node: Node which do non-leaf nodes.
Traversing can be defined as a process of visiting every node of the tree exactly once.

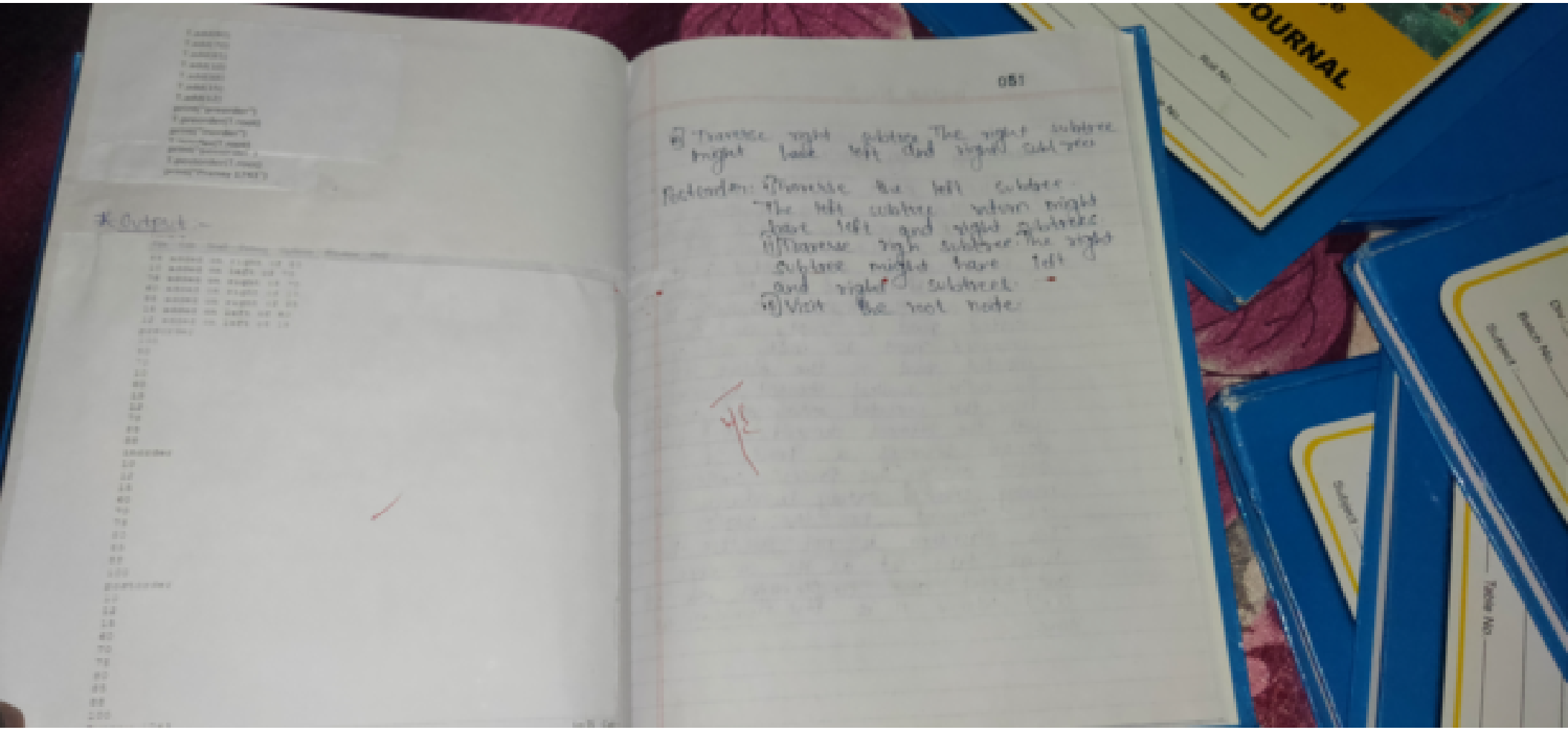
Preorder: i) Visit root node.
ii) Traverse the left subtree. The left subtree in turn might have left and right subtree.
iii) Traverse the right subtree. The right subtree inturn might have left and right subtree.

Inorder: i) Traverse left. The left subtree might have right subtree.
ii) Visit root node.

Program:

```
## Binary Tree and Traversal ##

class Node:
    global r
    global l
    global data
    def __init__(self,l):
        self.l=None
        self.data=l
        self.r=None
class Tree:
    global root
    def __init__(self):
        self.root=None
    def add(self,val):
        if self.root==None:
            self.root=Node(val)
        else:
            newnode=Node(val)
            h=self.root
            while True:
                if newnode.data < h.data:
                    if h.l!=None:
                        h=h.l
                    else:
                        h.l=newnode
                        print(newnode.data,"added on left of",h.data)
                        break
                else:
                    if h.r!=None:
                        h=h.r
                    else:
                        h.r=newnode
                        print(newnode.data,"added on right of",h.data)
                        break
    def preorder(self,start):
        if start!=None:
            print(start.data)
            self.preorder(start.l)
    def inorder(self,start):
        if start!=None:
            self.inorder(start.l)
            print(start.data)
            self.inorder(start.r)
    def postorder(self,start):
        if start!=None:
            self.inorder(start.l)
            self.inorder(start.r)
            print(start.data)
T=Tree()
T.add(100)
```


JOURNAL

Practical No: 13

Aim: To evaluate i.e. to sort given random data by using selection sort.

Theory:- Selection sort is a simple sorting algorithm. This sorting algorithm is an inplace comparison based algorithm in which the list is divided into two part the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is empty and the unsorted part is the entire list. The entire smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes a part of sorted array. This process continues moving unsorted array boundary by one element to the right. This algorithm is not suitable for large data set as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

Input:-

```
print("Pranay Bovini\nRoll No:1743")
a=[22,21,20,19,18,17]
print(a)

for i in range(len(a)-1):
    for j in range(len(a)-1):
        if(a[j]>a[j+1]):
            t=a[j]
            a[j]=a[j+1]
            a[j+1]=t
print(a)
```

Output:-

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 6
4 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/[illegible]/AppData/Local/Programs/Python/Python37/Selectio
nsort.py
Pranay Bovini
Roll No:1743
[22, 21, 20, 19, 18, 17]
[17, 18, 19, 20, 21, 22]
>>>
```

Practical 1

* Aim: To search a number from the list using linear unsorted.

* Theory: The process of identifying or finding a particular record is called searching. There are two types of search
1. Linear search
2. Binary search

The linear search is further classified as
* Sorted * Unsorted.

Here we will look on the UNSORTED Linear Search.

Linear Search, also known as sequential search, is a process that checks every element in the list sequentially until the desired element is found. When the elements to be searched are not specifically arranged in ascending or descending order.

Unsorted linear search.
- The data is entered in random manner.
- User needs to specify the element to be searched in the entered list.
- Check the condition that whether the entered number matches. If it matches then display the location plus increment 1 as data is stored from location zero.

Program


```
print("[illegible]")
[illegible]
a=[[illegible]]
[illegible]=int(input("Enter the number to find"))
for i in range([illegible]):
    if([illegible]):
        print("Number found")
        [illegible]
        break
    if([illegible]):
        print("Number not found")
```

Output:

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 6
4 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/[illegible]/AppData/Local/Programs/Python/Python37/[illegible].py
===
[illegible]
Enter the number to find[illegible]
Number not found
Number not found
Number found
>>>
```

Practical - 2 :

* Aim: To search a number from the list using linear sorted method.

* Theory: SEARCHING and SORTING are different modes or types of data structure.
SORTING:- To basically sort the inputed data in ascending or descending manner.
SEARCHING:- To search elements and to display the same.
In searching that too in LINEAR SORTED In searching the data is arrangements in search the data is [illegible] in ascending to descending or descending to ascending that is all what is "sorted" that meant by searching through sorted [illegible] arranged data.

SORTED LINEAR SEARCH

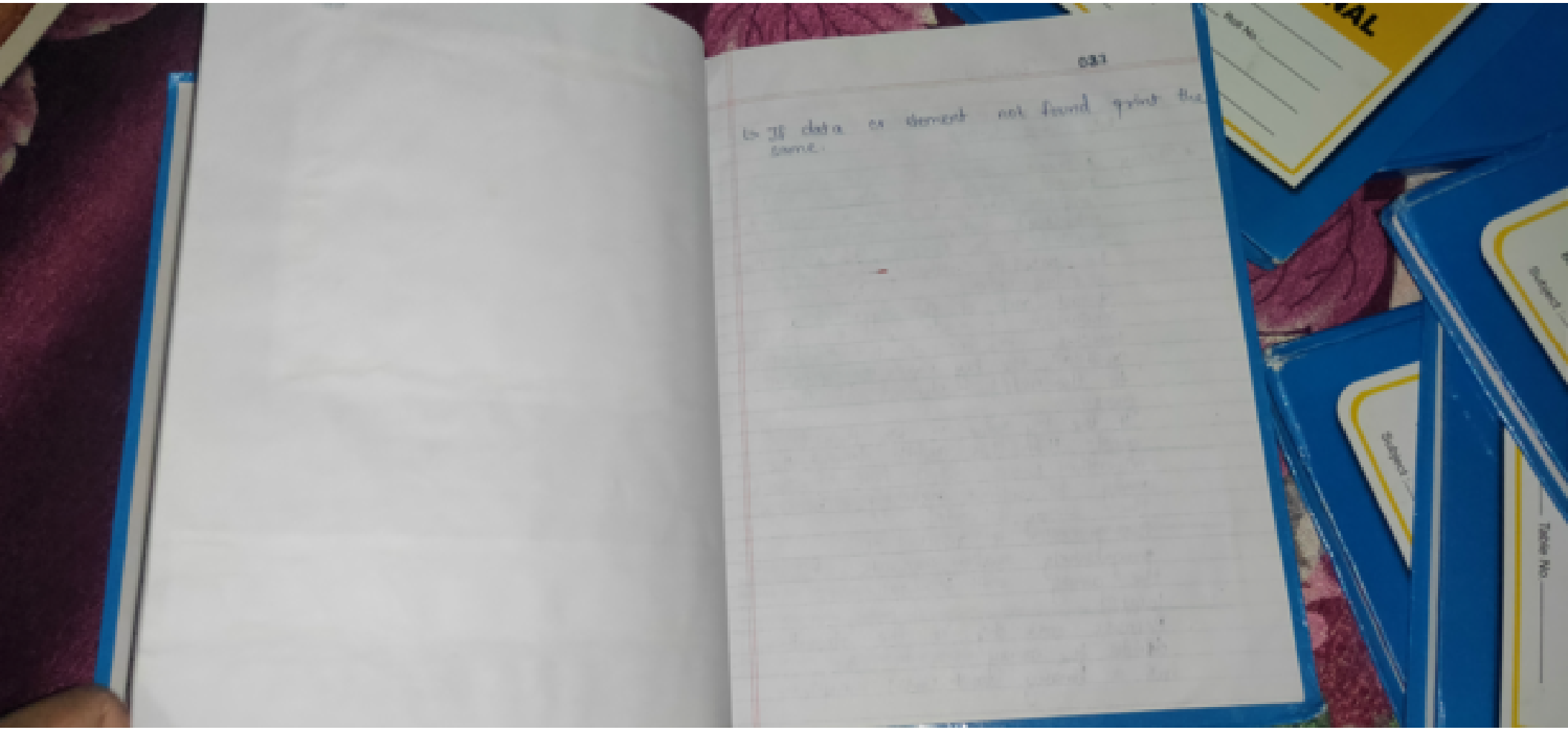
↳ The user is supposed to enter data in sorted manner.
↳ User has to given an element for searching through sorted list.
↳ If element is found display with an updated as value is sorted from label "0".
↳ If sorted order list of elements user can be check the condit" that whether the entered number lies from starting point till the last element.

Program


```
print("Pranay 1743")
f=False
a=[illegible]
n=int(input("Enter the number to find"))
if(n<a[0] or n>a[len(a)-1]):
 print("Number does not exist")
else:
 for i in range(len(a)):
  if(n==a[i]):
   f=True
   print("Number found at index",i)
   break
 if(f==False):
  print("Number is not found")
```

Output.

```
==================== RESTART: C:/Users/[illegible]/Desktop/[illegible].py ====================
===
Pranay  1743
Enter the number to find[illegible]
Number does not exist
>>> 
```


087
Subject
Table No

LEO Practical - 3

* Aim: To search a number from the given array list using binary search.

* Theory: A Binary Search also known as a half-interval search, is an algorithm used in computer science to locate a specified value (key) within an array. For this search to be binary, the array must be sorted in either ascending or descending order.
At each step of the algorithm a comparison is made and the possible location is one of two directions. Specifically, the key value is compared to the middle element of the array.
If the key value is less than or greater than this middle element, the algorithm knows which half of the array to continue searching in because the array is sorted.
This process is repeated on progressively smaller segments of the array until the value is located.
Because each step in the algorithm divides the array size in the half a binary search until complete

034

program:

```
print("Binary LIST")
arr=[1,2,3,4,5,6,7,8,9]
print(arr)
num=int(input("Enter number to be find from list"))
n=0
i=len(arr)-1
print(arr[0],arr[i])
if(num<arr[0] or num>arr[i]):
    print("Number not in range")
if(num==arr[0]):
    print("Number found at location",0+1)
if(num==arr[i]):
    print("Number found at location",i+1)
else:
    while(i>=n):
        if(num==arr[n]):
            print("Number found at location",n+1)
            break
        else:
            n=n+1
            num
            arr[n]==arr[n]
            else:
                print("Number not in given list")
```

output

```
========== RESTART: C:/Users/Swapnil/Desktop/Da/bs1.py ==========
>>>
Binary LIST
Enter number to be find from list 23
Number Found at Location 7
>>>
========== RESTART: C:/Users/Swapnil/Desktop/Da/bs1.py ==========
>>>
Binary LIST
Enter number to be find from list56
Number not in range
>>>
========== RESTART: C:/Users/Swapnil/Desktop/Da/bs1.py ==========
>>>
Binary LIST
Enter number to be find from list23
Number not in given list
>>>
```

MCA Practical-1

* Aim: To demonstrate the use of Stack.

* Theory: In computer science, a stack is an abstract data type that serves as a collection of elements with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed.
The order may be LIFO (Last In First Out) or FILO (First In Last Out).
These basic operations are performed in the stack.

- Push: Adds an item in the stack. If the stack is full said to be over flow condition.
- Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an underflow condition.
- Peek or Top: Returns top element of stack.
- Empty: Returns true if stack is empty else false.

4.4.1 37

```
print("Pranay1743")
class stack:
    global tos
    def __init__(self):
        self.l=[0,0,0,0,0,0,0,0]
        self.tos=-1
    def push(self,data):

        n=len(self.l)
        if self.tos==n-1:
            print("stack is full")
        else:
            self.tos=self.tos+1
            self.l[self.tos]=data
```

Output

```
========== RESTART: C:/Users/Swapnil/Desktop/Da/da4.py ==========
>>>
Enemy 1743
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty
```