Exam Seat No. _____

THAKUR COLLEGE OF SCIENCE & COMMERCE

NAAC Accredited with Grade "A" (3rd Cycle)

THAKUR® TRUSTS

ISO 9001 : 2015 Certified

Degree College

Computer Journal

CERTIFICATE

SEMESTER II UID No. _____

Class FYBSC Roll No. 1758 Year 2019 - 20

This is to certify that the work entered in this journal is the work of Mst. / ~~Ms.~~ CHAUHAN HIMANSHU . K

who has worked for the year 2020 in the Computer Laboratory.

Teacher In-Charge

Head of Department

Date : _____

Examiner

★★ INDEX ★★

No.	Title	Page No.	Date	Staff Member's Signature
1.	To search a number from the list using linear unsorted	37-38	21/12/19	
2.	To search a number from the list using linear sorted method	39-40	21/12/19	
3.	To search a number from the given sorted list using binary search.	41-42	21/12/19	
4.	Impetus reading		23/1/20	
5	To demonstrate the use of stack	42-44	18/1/20	
6.	To demonstrate the Queue add and delete	44-46	18/1/20	
7.	To demonstrate the use of circular queue in data structure	46-48	18/1/20	

★★ INDEX ★★

No.	Title	Page No.	Date	Staff Member's Signature
8	To demonstrate the use of linked list in data structure	48-50	25/1/20	
9	To evaluate postfix expression using stack	50-52	25/1/20	
10	To sort given random data by using bubble sort	53-54	1/2/20	
11	To evaluate, to sort the given data in Quick sort	54-56	1/2/20	
12	To evaluate to sort given random data by using selection sort.	57-58	8/2/20	
13	Binary tree and traversal.	58-60	8/2/20	
14	Merge sort.	61-62	8/2/20	

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
↳ Linear search
↳ 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 check 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. They are arranged in the random manner That is what it calls unsorted linear search

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 by 1 as data is stored from location zero

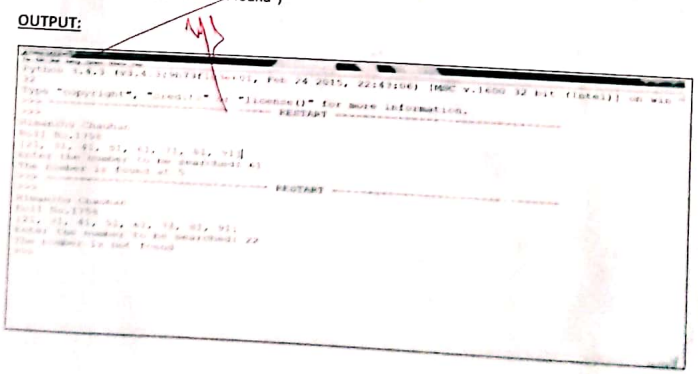
⇒ If all elements are checked one by one and element not found then prompt message number not found

PRACTICAL NO. 1

SOURCE CODE:

```
print("Himanshu Chauhan")
print("Roll No.1758")
found=False
a=[21,31,41,51,61,71,81,91]
print(a)
search=int(input("Enter the number to be searched: "))
for i in range(len(a)):
    if(search==a[i]):
        print("The number is found at" ,i+1)
        found=True
        break;
if(found==False):
        print("The number is not found")
```

OUTPUT:



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 search the data is ~~arranged~~ arranged in ascending to descending or descending to ascending That is all what it meant by searching through 'sorted' that is will 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 updation as, value is stored from location '0'

⇒ If data or element not found print the same.

⇒ In sorted Order list of elements we can check the condition that whether the entered number lies from starting point till the last element if not then without any processing we can say number not in the list.

PRACTICAL NO. 2

SOURCE CODE:

```
print("Himanshu Chauhan")
print("Roll No.1758")
found=False
a=[22,32,52,62,72,82,92]
print(a)
search=int(input("Enter the number to be searched: "))
if(search<a[0]or search>a[len(a)-1]):
    print("Number does not exist")
else:
  for i in range(len(a)):
    if(search==a[i]):
        print("The number is found at" ,i+1)
        found=True
        break;
if(found==False):
        print("The number is not found")
```

OUTPUT:

PRACTICAL - 3

AIM : To search a number from the given sorted 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 the 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 procedure branches into one of two directions.

specifically, the key value is ~~compared~~ to the middle elements of the array.

If the key value is less than or greater than the middle element, the algorithm knows which half of the array to continue searching in because the array is sorted.

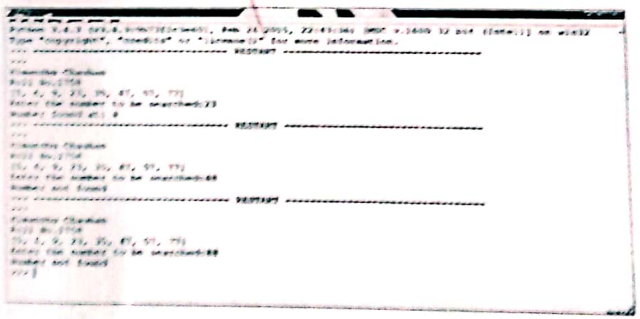
This process is repeated on progressively smaller segments of the array until the value is located.

Because each step in the algorithm divides the array size in half a binary search will complete successfully in Logarithmic time

PRACTICAL NO. 3

```
print("Himanshu Chauhan")
print("Roll No.1758")
a=[5,6,9,23,35,47,57,77]
print(a)
search=int(input("Enter the number to be searched:"))
l=0
r=len(a)-1
while(True):
  m=(l+r)//2
  if(l>r):
    print("Number not found")
    break
  if(search==a[m]):
    print("Number found at:",m+1)
    break
  elif(search<a[m]):
    r=m-1
  else:
    l=m+1
```

OUTPUT :-



SOURCE CODE:-

```
print("HimanshuChauhan")
print("Roll No 1758")
class stack:
 globaltos
 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+=1
   self.l[self.tos]=data
   def pop(self):
   if self.tos<0:
   print("stack empty")
   else:
        k=self.l[self.tos]
   print("data=",k)
   self.tos=self.tos-1
   s=stack()
   s.push(12)
   s.push(13)
   s.push(14)
   s.push(15)
   s.push(16)
   s.push(17)
   s.push(18)
   s.push(19)
```

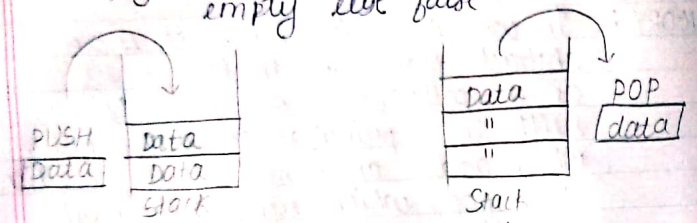

PRACTICAL - 4

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 FIFO (First In last out).
Three Basic operations are performed in the stack.

- PUSH : Adds an item in the stack. If the stack is full it is said to be overflow condition.
- POP : Removes an item from the stack. The items from the are stack 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 elements of stack
- is Empty : Returns true if stack is empty else false.



last_in_First_out

```
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
```

OUTPUT:-

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Himanshu Chauhan
Roll No. 1758
stack is full
data= 18
data= 17
data= 16
data= 15
data= 14
data= 13
data= 12
stack empty
>>>
```

SOURCE CODE:-

```
print("HimanshuChauhan")
print("Roll No.1758")
class Queue:
global r
global f
def __init__(self):
self.r=0
self.f=0
self.l=[0,0,0,0,0,0]
def add(self,data):
    n=len(self.l)
ifself.r<n-1:
self.l[self.r]=data
self.r=self.r+1
else:
print("Queue is full")
def remove(self):
    n=len(self.l)
ifself.f<n-1:
print(self.l[self.f])
self.f=self.f+1
else:
print("Queue is empty")

Q=Queue()
Q.add(10)
Q.add(20)
Q.add(30)
Q.add(40)
```

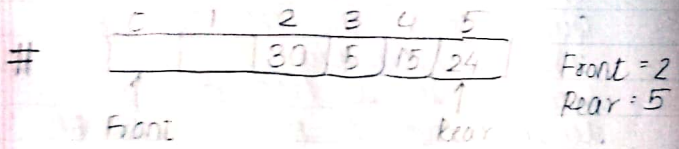

PRACTICAL - 5

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 at both of its ends
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.

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

Rear is used to get the last item from a queue



Q.remove()

Q.remove()

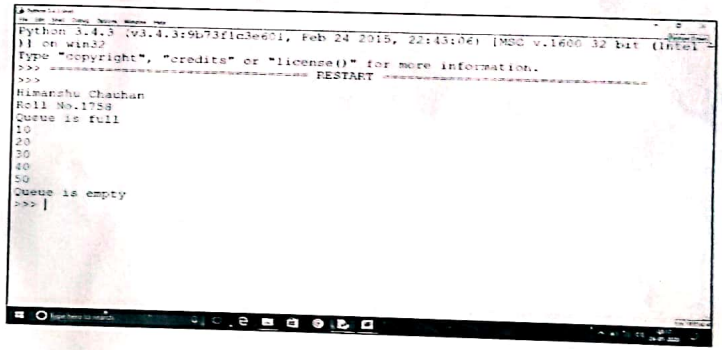
Q.remove()

Q.remove()

Q.remove()

Q.remove()

OUTPUT :-



SOURCE CODE:-

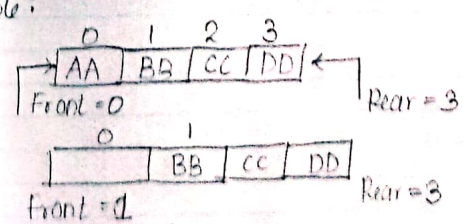
```
print("HimanshuChauhan")
print("Roll No 1758")
class Queue:
global r
global f
def __init__(self):
self.r=0
self.f=0
self.l=[0,0,0,0,0,0]
def add(self,data):
    n=len(self.l)
if self.r<=n-1:
self.l[self.r]=data
print("data added ",data)
self.r=self.r+1
else:
        s=self.r
self.r=0
if self.r<self.f:
self.l[self.r]=data
self.r=self.r+1
else:
self.r=s
print("Queue is full")
def remove(self):
    n=len(self.l)
if self.f<=n-1:
print("data removed:",self.l[self.f])
self.f=self.f+1
```

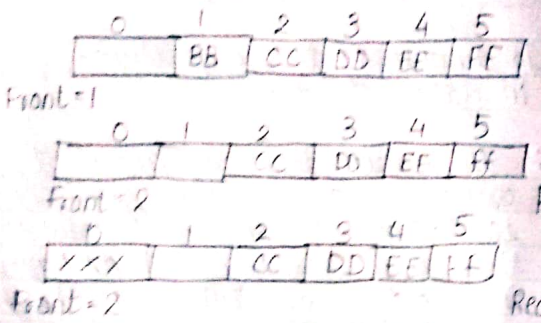

PRACTICAL - 16

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

THEORY : The queue that we implement using an array suffer from one limitation. In that implementation there is a posibility that the queue is reported as full, even though in actuality there might be empty slots 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 and reach the end of the array. The next element is stored in the first slot of the array.

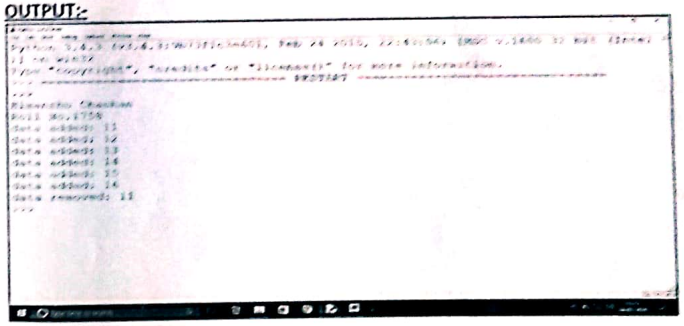
ample :






```
if self.f<=self.r:
print(self.l[self.f])
self.f=self.f+1
else:
print("Queue is empty")
self.fin
Q=Queue()
Q.add(11)
Q.add(12)
Q.add(13)
Q.add(14)
Q.add(15)
Q.add(16)
Q.remove()
Q.add(13)
```

OUTPUT:-



```
#LINKED LIST
print("HimanshuChauhan\nRoll no.1758")
class node:
 global data
 global next
 def __init__(self,item):
  self.data=item
  self.next=None
 class linkedlist:
  global s
  def __init__(self):
   self.s=None
   defaddl(self,item):
   newnode=node(item)
   if self.s==None:
    self.s=newnode
    else:
     head=self.s
     while head.next!=None:
     head=head.next
     head.next=newnode
     defaddBl(self,item):
     newnode=node(item)
     if self.s==None:
     self.s=newnode
     else:
      newnode.next=self.s
```

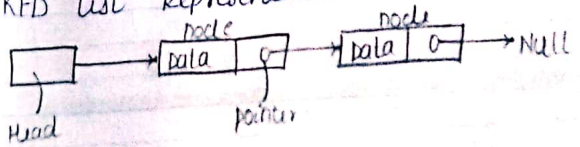

PRACTICAL - 7

AIM : To demonstrate the use of linked list in data structure

THEORY : A linked list is a sequence of data structures linked list is 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 LIST - A linked list contains the connection link to the first link called first.

LINKED list Representation :



Types of linked list :
- Simple
- Doubly
- Circular

Basic Operations
- Insertion
- Deletion
- Display
- Search
- Delete

```
self.s=newnode
def display(self):
head=self.s
whilehead.next!=None:
print(head.data)
head=head.next
print(head.data)
start=linkedlist()
start.addL(50)
start.addL(60)
start.addL(70)
start.addL(80)
start.addB(40)
start.addB(30)
start.addB(20)
start.display()
```

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (
Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Himanshu Chauhan
Roll no.1750
20
30
40
50
60
70
80
>>>
```

#POSTFIX

```
print("Himanshuchauhan\nRoll no.1758")
def evaluate(s):
    k=s.split()
    n=len(k)
  stack=[]
  for i in range(n):
   if k[i].isdigit():
    stack.append(int(k[i]))
   elif k[i]== '+':
                a=stack.pop()
                b=stack.pop()
    stack.append(int(b)+int(a))
   elif k[i]=='-':
                a=stack.pop()
                b=stack.pop()
     stack.append(int(b)-int(a))
    elif k[i]== '*':
                a=stack.pop()
                b=stack.pop()
     stack.append(int(b)*int(a))
    else:
                a=stack.pop()
                b=stack.pop()
     stack.append(int(b)/int(a))
    returnstack.pop()
    s="8 6 9 * +"
```


PRACTICAL- 8

AIM : To evaluate postfix expression using stack.

THEORY : Stack is an (ADT) and works on LIFO (last-in-first-out) i.e PUSH & POP operation
A postfix expression is a collection of operators and operands in which the operator is placed after the operands.

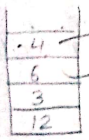
steps to be followed:
1. Read all the symbols one by one from left to right in the given postfix expression.
2. If the reading symbol is operand then push it on to the stack
3. If the reading symbol is operator (+, -, *, /, etc) then perform TWO pop operations and store the two popped operands in two different variables (operand 1 & operand 2) Then perform reading symbol operator using operand 1 & operand 2 and push result back on to the stack.
4. Finally! Perform a pop operation and

display the popped value as final result.

value of postfix expression :

S = 12 3 6 4 (-) + *

start :



b - a = 6 - 4 = 2 // store again in stack

b + a = 3 + 2 = 5 // store result in stack

b * a = 12 * 5 = 60

```
r=evaluate(s)
print("the evaluate value is:",r)
```

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (I
Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Himanshu chauhan
Roll no.1758
the evaluate value is: 62
>>>
```

PRACTICAL - 9

AIM : To sort given random data by using bubble sort

THEORY : Sorting is type in which any random data is sorted i.e arranged in ascending or descending order.
BUBBLE sort sometimes referred to as sinking sort.
Is a simple sorting 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 checks if the condition fails then only swaps otherwise goes on

Example :

First pass

(51428) ⟶ (15428) Here algorithm compares the first two elements and swaps since 5>1

(15428) ⟶ (14528) Swap since 5>4

(14528) ⟶ (14258) Swap since 5>2

(14258) ⟶ (14258) Now since these elements are already in order (8>5) algorithm does not swap them.

Second pass :

(14258) ⟶ (14258)

(14258) ⟶ (12458) Swap sin (4>2

(12458) ⟶ (12458)

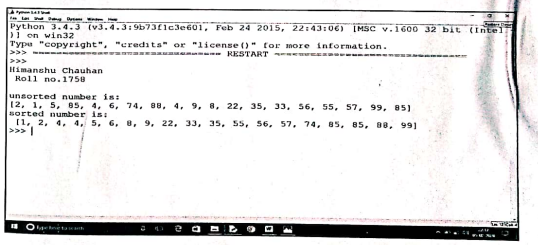
Third pass :

(12458) It checks and gives the data in sorted order.

#Bubble Sort

```
print("HimanshuChauhan\n Roll no.1758")

a=[2,1,5,85,4,6,74,88,4,9,8,22,35,33,56,55,57,99,85]

print("\nunsorted number is:")

print(a)

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

for j in range (len(a)-1-i):

if (a[j]>a[j+1]):

        t=a[j]

a[j]=a[j+1]

a[j+1]=t

print("sorted number is:\n",a)
```

OUTPUT:



#Quick Sort

```
print("Himanshu Chauhan\n Roll no. 1758")
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]
```


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 picked pivot. There are many different versions of quick sort that pick pivot in different ways.

1) Always pick first element as pivot.
2) Always pick last element as pivot
3) pick a random element as pivot
4) pick median as pivot

The key process in quicksort is partition
(). Target of partitions is given an array and an element X of array as pivot, put X at its correct 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.

alist[rightmark]=temp

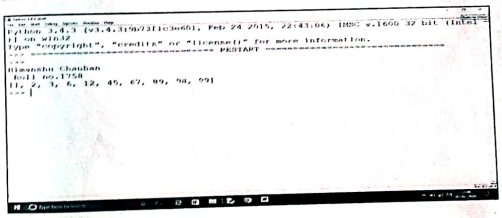
returnrightmark

alist=[1,45,6,98,89,67,3,12,2,99]

quicksort(alist)

print(alist)

OUTPUT:



PRACTICAL - 11

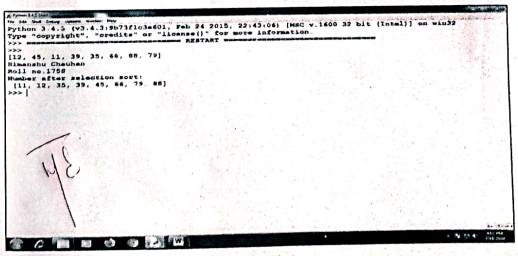
AIM : To evaluate i.e to sort given random data by using selection sort.

THEORY : Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison based algorithm in which the list is divided into two parts the sorted part at the left end and. Initially the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the left most 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(n2), where n is the number of items.

#Selection Sort:

```
a=[12,45,11,39,35,66,88,79]
print(a)
print("HimanshuChauhan\nRoll no.1758")
for i in range(len(a)-1):
for j in range(len(a)-1):
if(a[j]>a[i+1]):
        t=a[j]
a[j]=a[i+1]
a[i+1]=t
print("Number after selection sort:\n",a)
```

OUTPUT:-



#Binary SearchTree

```
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
```


PRACTICAL - 12

AIM : Binary Tree and Traversal.

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 : Nodes which are non-leaf nodes.

Traversing can be defined as a process of visiting every node of the tree exactly once.

BINARY TREE : A tree whose elements have atmost 2 children is called a binary tree. Since each element in a binary tree have only 2 children named as left and right child. It is represented by a pointer to the topmost node in tree. If the tree is empty then value of root is NULL.

A tree node contains following parts:
1. Data
2. Pointer to left child
3. Pointer to right child

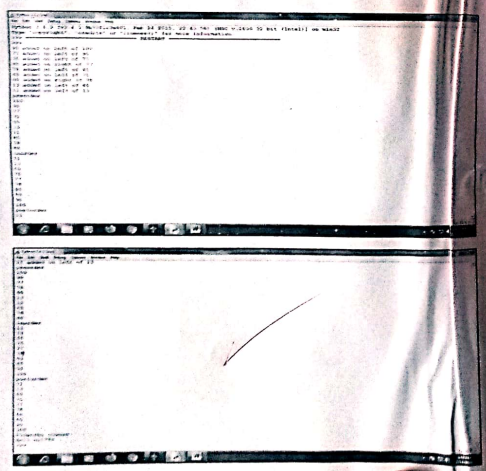
Preorder: i) visit the root node.
(ii) Traverse the left subtree. The left subtree in turn might have left and right subtree.
(iii) Traverse the right subtree. The right subtree in turn might have left and right subtree.

Inorder: i) Traverse the left subtree. The left subtree in turn might have left and right subtree.
(ii) Traverse the right subtree. →
iii) Visit the root node.
The right subtree have left and right subtrees.

```
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)
self.preorder(start.r)
definorder(self,start):
if start!=None:
self.inorder(start.l)
print(start.data)
self.inorder(start.r)
defpostorder(self,start):
if start!=None:
self.inorder(start.l)
self.inorder(start.r)
print(start.data)
T=Tree()
T.add(100)
T.add(90)
T.add(77)
T.add(75)
T.add(85)
T.add(78)
T.add(65)
T.add(80)
T.add(13)
T.add(11)
```

```
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
print("Himanshuchauan\nRoll no1758")
```

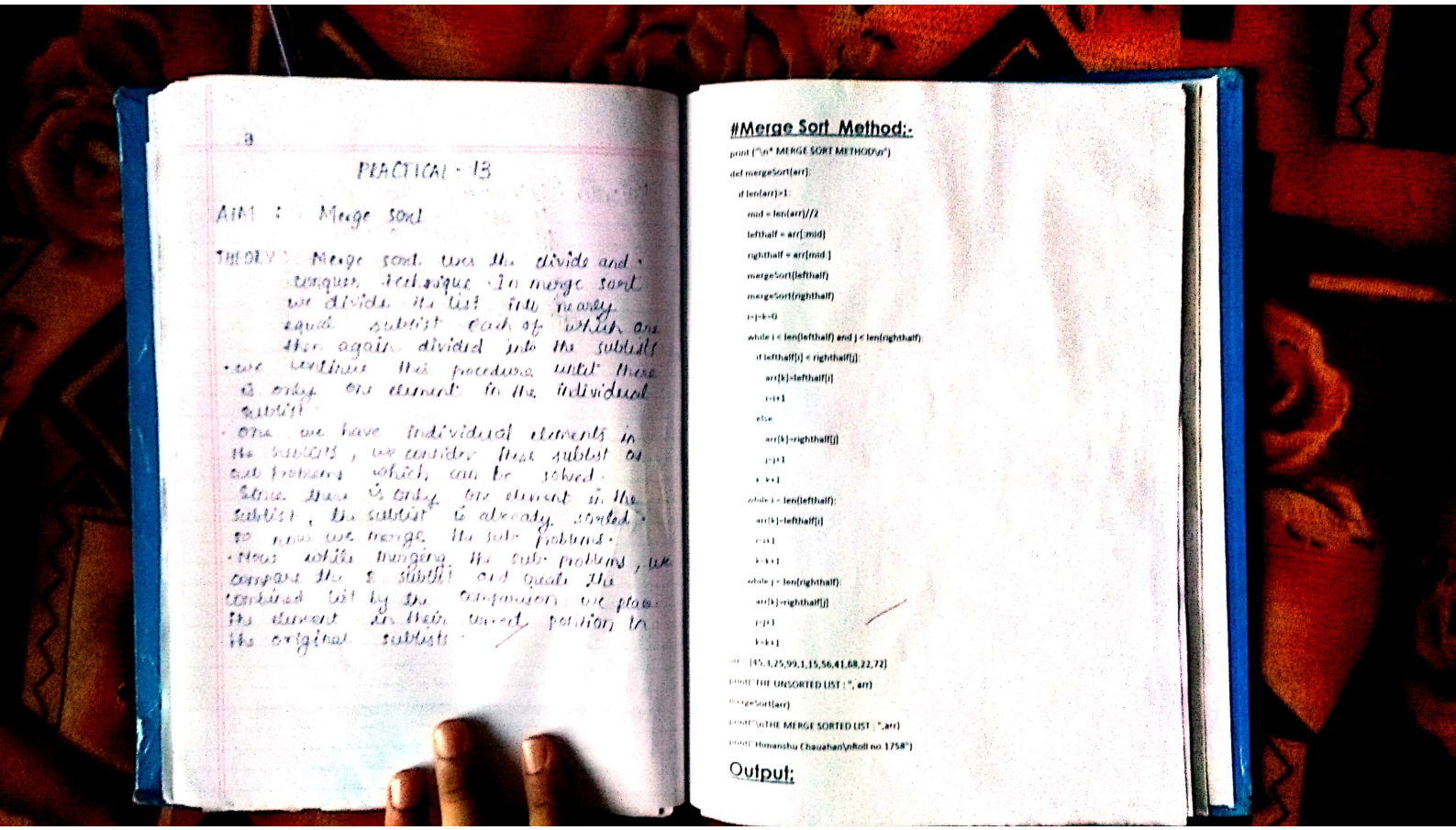
OUTPUT:-



Postorder : i) Traverse the left subtree. The left subtree in turn might have left and right subtrees.

ii) Traverse the right subtree. The right subtree in turn might have left and right subtrees.

iii) visit the root node.



3

PRACTICAL - 13

AIM : 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 then again divided into the 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 our problems which can be solved. Since there is only one element in the sublist, the sublist is already sorted. So now we merge the sub problems.
- Now while merging the sub-problems, we compare the 2 sublist and create the combined list by the comparison we place the element in their correct position in the original sublists.

#Merge Sort Method:-

```
print ("\n* MERGE SORT METHOD\n")
def mergeSort(arr):
  if len(arr)>1:
    mid = len(arr)//2
    lefthalf = arr[:mid]
    righthalf = arr[mid:]
    mergeSort(lefthalf)
    mergeSort(righthalf)
    i=j=k=0
    while i < len(lefthalf) and j < len(righthalf):
      if lefthalf[i] < righthalf[j]:
        arr[k]=lefthalf[i]
        i=i+1
      else:
        arr[k]=righthalf[j]
        j=j+1
      k +=1
    while i < len(lefthalf):
      arr[k]=lefthalf[i]
      i=i+1
      k=k+1
    while j < len(righthalf):
      arr[k]=righthalf[j]
      j=j+1
      k=k+1
arr = [45,1,25,99,1,15,56,41,68,22,72]
print("THE UNSORTED LIST : ", arr)
mergeSort(arr)
print("\nTHE MERGE SORTED LIST : ",arr)
print(" Himanshu Chauahan\nRoll no 1758")
```

Output: