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 F4 BSC CS **Roll No.** 1730 **Year** 2019-20

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

who has worked for the year 2019-20 in the Computer Laboratory.

Teacher In-Charge

Head of Department

Date :_____

Examiner

☆☆ INDEX ☆☆

SEM II
DS.

☆☆ INDEX ☆☆

INDEX

No.	Title	Page No.	Date	Staff Member's Signature
12.	PRACTICAL - 12			
	BINARY TREE and			
	Pre-Order, In-Order and			
	Post-Order traversing.	65-67		

CODING (INPUT) :-

P1 LSUM.py - F:\PRACTICALS\DS\27.11.19 (P1 2 3)\P1 LSUM.py (2.7.17)

File Edit Format Run Options Window Help

```
print "DINESH K LOHAR\nFYBSC CS 1730\n27-11-2019\nPRACTICAL 1\n"

#LINEAR SEARCH UNSORTED METHOD
print "* LINEAR SEARCH UNSORTED METHOD\n"
A=[70,58,27,36,49,69,10]
d=0
search=int(input("Enter the number to be searched : "))
for i in range (len(A)):
    if (search==A[i]):
        print "Number is Found at ",i
        d=1
        break
if d==0:
    print "Number is not Found"
```

Ln: 18 Col: 0

27-11-19

PRACTICAL - 1

Aim :- To search a number or an item from a list using Linear search Unsorted Method.

Theory :- LINEAR SEARCH UNSORTED METHOD :-

- The process of identifying or finding a particular record is called searching.
- There are two types of linear search method:

* Unsorted
* Sorted

- Linear search Unsorted method - linear search is also known as sequential search because it 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. They are arranged in random manner. That is what it calls unsorted linear search.

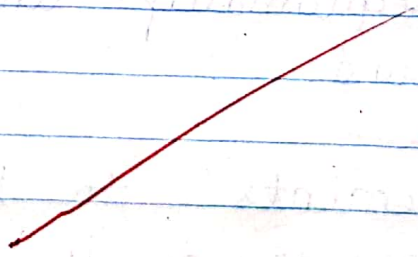
- Unsorted linear search :-

↳ The data is entered in random manner.

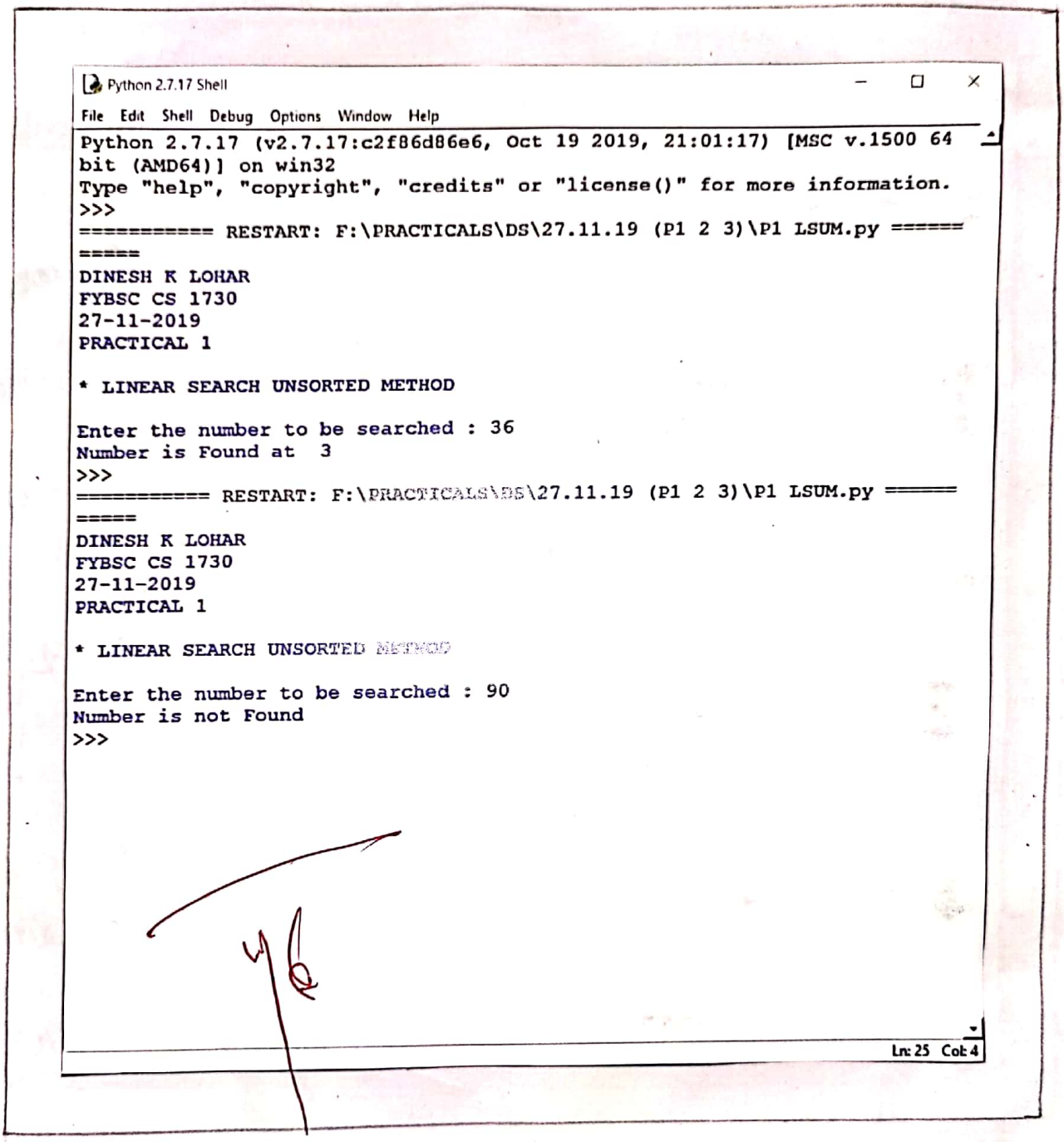
→ User needs to specify the element to be searched in the entered list.

→ Check the condition that whether the entered number matches, if it matches then display the location of that number.

→ If all elements are checked one by one and elements not found then prompt message number is not found.



OUTPUT :-



CODING (INPUT) :-

P2 LSSM.py - F:\PRACTICALS\DS\27.11.19 (P1 2 3)\P2 LSSM.py (2.7.17)

File Edit Format Run Options Window Help

```
print "DINESH K LOHAR\nFYBSC CS 1730\n27-11-2019\nPRACTICAL 2\n"

#LINEAR SEARCH SORTED METHOD
print "* LINEAR SEARCH SORTED METHOD\n"
A=[10,27,36,49,58,69,70]
search=int(input("Enter the number to be searched : "))
if search<A[0] or search>A[len(A)-1] :
    print "Number does not EXISTS"
else:
    for i in range (len(A)):
        if (search==A[i]):
            print "Number is Found at ",i
            break
```

Ln: 16 Col: 0

PRACTICAL-2

Aim :- To search a number or a item from a list using linear search Sorted Method.

Theory :- LINEAR SEARCH SORTED METHOD :-

- Searching and sorting are different modes or types of data structure.

Sorting - To arrange the data in the ascending or discending order.

Searching - To search elements and to display the location of the element.

- In linear search sorted method the elements are arranged in ascending or descending manner. That is all what is meant by searching through a sorted list.

- Sorted linear search :-

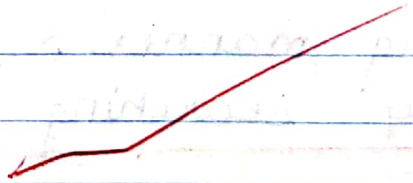
→ The user is supposed to enter data in sorted manner (ascending or descending order).

→ User has to given an element for searching through the sorted list.

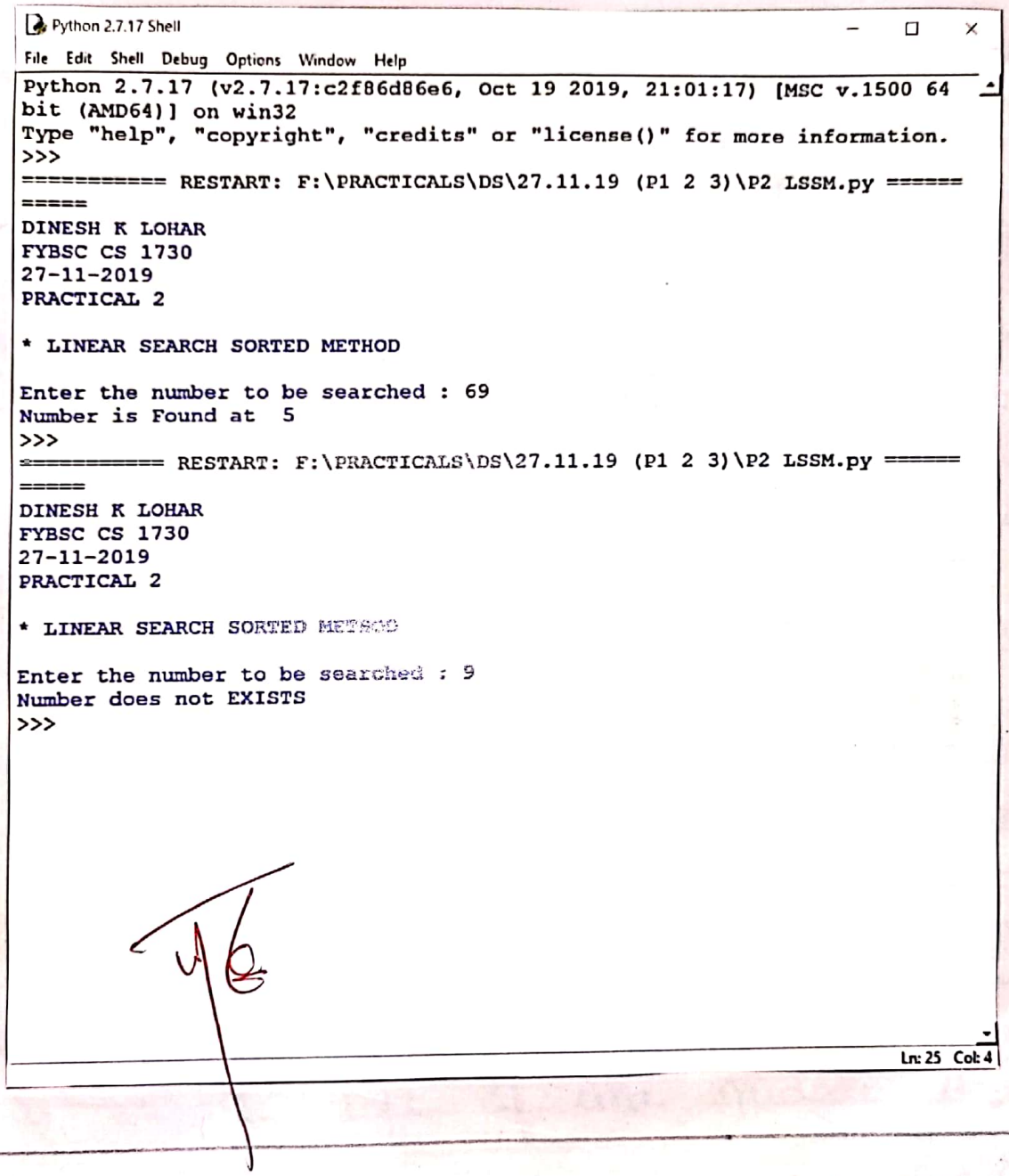
→ If element is found display with an updation as value is stored from location '0';

→ If data or element not found print Number does not EXISTS.

→ 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 is not in the list.



OUTPUT :-



Python 2.7.17 Shell

File Edit Shell Debug Options Window Help

```
Python 2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019, 21:01:17) [MSC v.1500 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: F:\PRACTICALS\DS\27.11.19 (P1 2 3)\P2 LSSM.py ======
=====
DINESH K LOHAR
FYBSC CS 1730
27-11-2019
PRACTICAL 2

* LINEAR SEARCH SORTED METHOD

Enter the number to be searched : 69
Number is Found at  5
>>>
========== RESTART: F:\PRACTICALS\DS\27.11.19 (P1 2 3)\P2 LSSM.py ======
=====
DINESH K LOHAR
FYBSC CS 1730
27-11-2019
PRACTICAL 2

* LINEAR SEARCH SORTED METHOD

Enter the number to be searched : 9
Number does not EXISTS
>>>
```

Ln: 25 Col: 4

CODING (INPUT) :-

P3 BSM.py - F:\PRACTICALS\DS\27.11.19 (P1 2 3)\P3 BSM.py (2.7.17)

File Edit Format Run Options Window Help

```
print "DINESH K LOHAR\nFYBSC CS 1730\n27-11-2019\nPRACTICAL 3\n"

#BINARY SEARCH METHOD
print "* BINARY SEARCH METHOD\n"
def bsm(A,l,h,k):
    if h>=l:
        m=l+(h-l)/2
        if A[m]==k:
            return m
        elif A[m]>k:
            return bsm(A,l,m-1,k)
        else:
            return bsm(A,m+1,h,k)
    else:
        return -1
A=[9,14,27,34,51,78,83]
k=int(input("Enter the number to be searched : "))
result=bsm(A,0,len(A)-1,k)
if result != -1:
    print "Number is found at ",result
else:
    print "Number is not present"
```

Ln: 25 Col: 0

PRACTICAL - 3

Aim :- To search a number or a item from a list using Binary Search Method.

Theory :- **BINARY SEARCH METHOD :-**

- Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.
- Binary search looks for a particular item by comparing the middle-most item of the list.
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item.
- This process continues on the sub-arrays as well until the size of the sub-array reduces to zero.

- Binary search works :-

→ Let the following array is sorted and assume that we need to search the location of 51.

low			mid			high
9	14	27	34	51	78	83
0	1	2	3	4	5	6

→ First, determine half of the array by formula:-
mid = low + (high - low) / 2.

→ Here, it is 0 + (6-0)/2 = 3

∴ The mid = 3

→ Now, we compare the stored value at location

→ 3 with value being searched i.e., 51.

∴ 34 < 51.

→ we change our low to mid + 1 and find the new mid value again

→ low = mid + 1

mid = low + (high - low)/2 ⇒ mid = 5

→ Again we compare mid value with 51.

→ ∴ The stored value at location 5 ~~does~~ not match, it is more than what

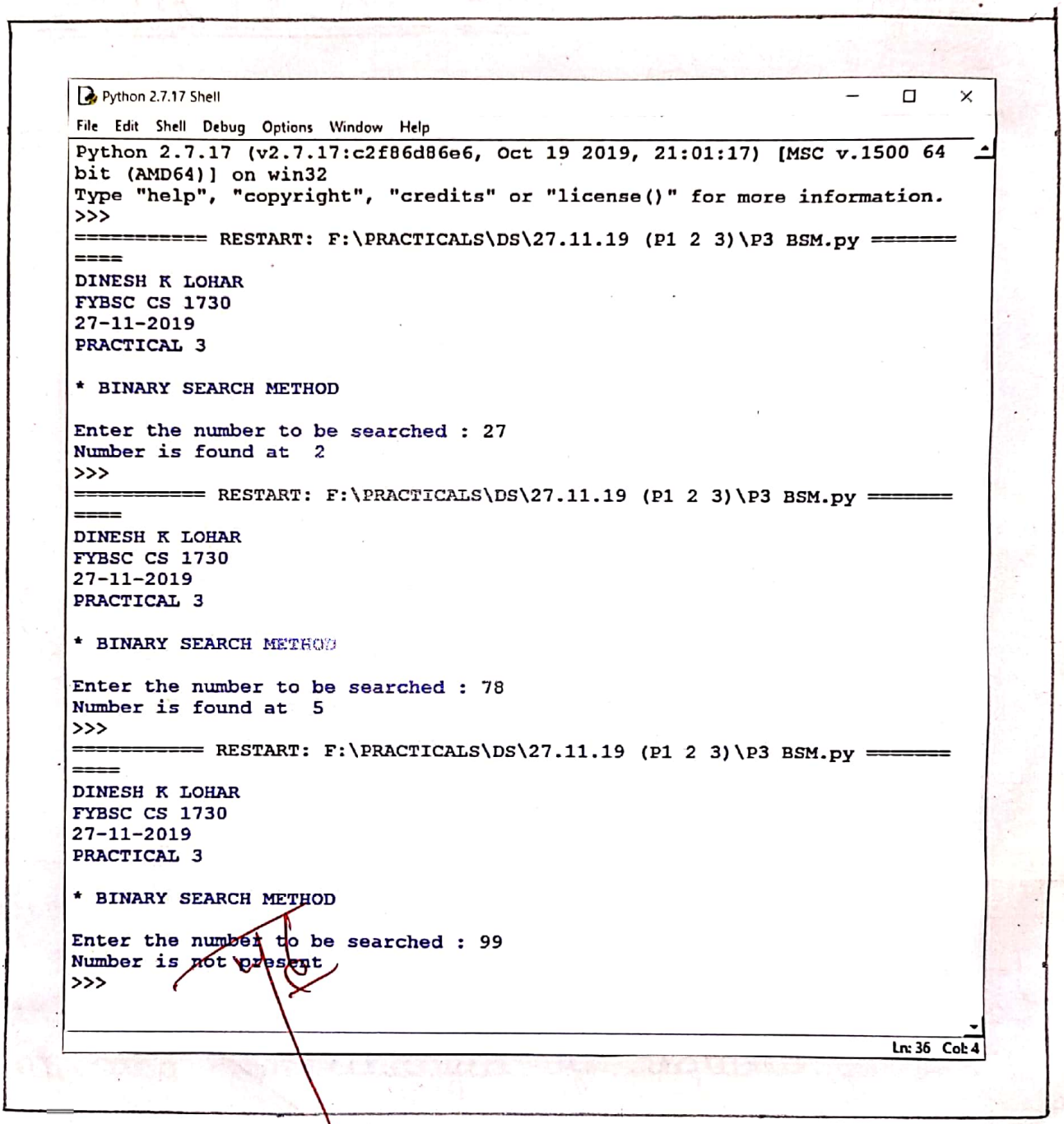
→ we are looking for. So, the value must be in the lower part from the location.

→ Hence, we calculate the mid again, this time it is 4.

→ we compare the stored value at location 4 and the searched value. We find that it is a match.

→ So, we conclude that the value 51 is at 4.

OUTPUT :-



Python 2.7.17 Shell

File Edit Shell Debug Options Window Help

```
Python 2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019, 21:01:17) [MSC v.1500 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: F:\PRACTICALS\DS\27.11.19 (P1 2 3)\P3 BSM.py =======
====
DINESH K LOHAR
FYBSC CS 1730
27-11-2019
PRACTICAL 3

* BINARY SEARCH METHOD

Enter the number to be searched : 27
Number is found at  2
>>>
========== RESTART: F:\PRACTICALS\DS\27.11.19 (P1 2 3)\P3 BSM.py =======
====
DINESH K LOHAR
FYBSC CS 1730
27-11-2019
PRACTICAL 3

* BINARY SEARCH METHOD

Enter the number to be searched : 78
Number is found at  5
>>>
========== RESTART: F:\PRACTICALS\DS\27.11.19 (P1 2 3)\P3 BSM.py =======
====
DINESH K LOHAR
FYBSC CS 1730
27-11-2019
PRACTICAL 3

* BINARY SEARCH METHOD

Enter the number to be searched : 99
Number is not present
>>>
```

Ln: 36 Col: 4

Formula

$m = (l+h)/2$

$m = (l+(h-l))/2$

$m = (l+(h+l))/2$

SOURCE CODE -

```
print "Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 4"

#BUBBLE SORT METHOD
print "\n* BUBBLE SORT METHOD\n"

A=[78,56,12,37,52,0,1,9,25]
print "The RANDOM Sorted list : \n",A

for passes in range(len(A)-1):
     for compare in range(len(A)-1-passes):
          if (A[compare]>A[compare+1]):
               t=A[compare]
               A[compare]=A[compare+1]
               A[compare+1]=t

print "\nThe BUBBLE Sorted list : \n",A
```

PRACTICAL-4

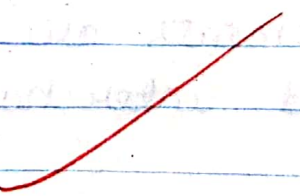
Aim :- To sort given random data by using bubble sort method.

Theory :- BUBBLE SORT METHOD :-

- Sorting means arranging a set of data in some order. There are different that are used to sort the data in ascending or descending order.
- In this method 0^{th} element is compared with the 1^{st} element. If it is found to be greater than 1^{st} element then they are interchanged.
- Then the first element is compared with the 2^{nd} element, if it is found to be greater, then they are interchanged. In this way all the elements are compared with the next element and interchanged if required.
- This is the first iteration and on completing this iteration the last element gets placed at the last position.
- Similarly, in the second iteration the comparisons are made till the last but one element and this time the second largest element gets

placed at the second last position in the list.

- As a result after all the iteration the list becomes a sorted list.



OUTPUT -

```
Python 2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019, 21:01:17) [MSC
v.1500 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
========== RESTART: C:/Users/danis/Desktop/bsmP4.py ==========
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 4

* BUBBLE SORT METHOD

The RANDOM Sorted list :
[78, 56, 12, 37, 52, 0, 1, 9, 25]

The BUBBLE Sorted list :
[0, 1, 9, 12, 25, 37, 52, 56, 78]
>>>
```

SOURCE CODE -

```
print ("Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 5")

#STACKS METHOD
print ("\n* STACKS METHOD\n")

class stack:
     global tos

     def __init__(self):
          self.l=[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 : FULL")
          else:
               self.tos=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(80)
s.push(70)
s.push(60)
s.push(50)
s.push(40)
s.push(30)
s.push(20)
s.push(10)

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

PRACTICAL - 5

Aim :- use of stacks in data structure.

Theory :- STACKS :

- In computer science, a stack is an abstract data type that serves as a collection of elements, with two principal operations :

1) PUSH : It adds an element to the collection.

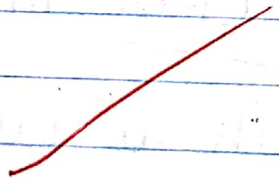
2) POP : It removes the most recently added element that was not yet removed.

- The order in which elements comes off a stack gives rise to its alternative name, LIFO (Last In, First Out).

- If the stack is full and does not contain enough space to accept an entity to be pushed, the stack then considered to be in an overflow state. The pop operation removes an item from the top of the stack.

- A stack can be easily implemented either through an array or a linked list. What identifies the data structure as a

stack in either case is not the implementation but the interface is the user is only allowed to pop or push items onto the array, with few other helper operations.



OUTPUT -

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20)
[MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
========== RESTART: C:/Users/Desktop/stacks.py ==========
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 5

* STACKS METHOD

STACK : FULL
Data : 20
Data : 30
Data : 40
Data : 50
Data : 60
Data : 70
Data : 80
STACK : EMPTY
>>>
```

SOURCE CODE -

```
print ("Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 6")

#QUEUE ADD & REMOVE METHOD
print ("\n* QUEUE : ADD & REMOVE\n")

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
            self.r=self.r+1
        else:
            print("QUEUE : FULL")

    def remove(self):
        n=len(self.l)
        if self.f<n-1:
            print("Data : ",self.l[self.f])
            self.f=self.f+1
        else:
            print("QUEUE : EMPTY")

Q=Queue()
Q.add(33)
Q.add(44)
Q.add(55)
Q.add(66)
Q.add(77)
Q.add(88)

Q.remove()
Q.remove()
Q.remove()
Q.remove()
Q.remove()
Q.remove()
```

PRACTICAL - 6

Aim :- Adding and Deleting data in a Queue

Theory :- QUEUE :-

- A queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and removal from the other end of the sequence. By convention, the end of the sequence at which elements are added is called the BACK or REAR of the queue and the end at which elements are removed is called the HEAD or FRONT of the queue.

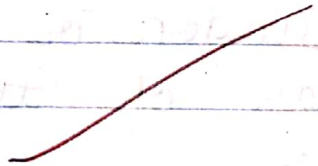
- The operation of adding an element to the rear of the queue is known as ENQUEUE.

- The operation of removing an element from the front is known as DEQUEUE.

- The operations of a queue make it a first-in-first-out (FIFO) data structure. In FIFO data structure, the first element

added to the queue will be the first one to be removed.

- A queue is an example of a linear data structure, or more abstractly a sequential collection.



OUTPUT -

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20)
[MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
============ RESTART: C:/Users/Desktop/queue.py ============
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 6

* QUEUE : ADD & REMOVE

QUEUE : FULL
Data : 33
Data : 44
Data : 55
Data : 66
Data : 77
QUEUE : EMPTY
>>>
```

SOURCE CODE -

```
print ("Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 7")

#CIRCULAR QUEUE METHOD
print ("\n* CIRCULAR QUEUE\n")

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 : 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
        else:
            s=self.f
            self.f=0
            if self.f<self.r:
                print(self.l[self.f])
                self.f=self.f+1
            else:
                print("QUEUE : EMPTY")
                self.f=s

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)
```

PRACTICAL - 7

Aim :- Using circular Queue add & remove the data.

Theory :- CIRCULAR QUEUE :-

- Circular Queue is also a linear data structure, which follows the principal of FIFO (First - In, First out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence, making the queue behave like a circular data structure.

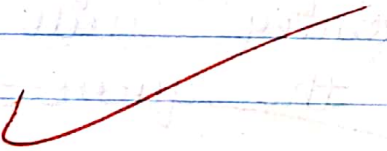
- In circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.

- New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.

- In circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when dequeue is executed. As the queue

data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.

- The head and tail pointer will get reinitialised to 0 every time they reach the end of the queue.



OUTPUT -

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20)
[MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
========== RESTART: C:/Users/Desktop/circularq.py ==========
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 7

* CIRCULAR QUEUE

Data Added : 44
Data Added : 55
Data Added : 66
Data Added : 77
Data Added : 88
Data Added : 99
Data Removed : 44
>>>
```

SOURCE CODE -

```
print ("Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 8")

#SIMPLE LINKED LIST METHOD
print ("\n* SIMPLE LINKED LIST METHOD\n")

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

    def addL(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

    def addB(self,item):
        newnode=node(item)
        if self.s==None:
            self.s=newnode
        else:
            newnode.next=self.s
            self.s=newnode

    def display(self):
         head=self.s
         while head.next!=None:
            print ("Data : ",head.data)
            head=head.next
         print ("Data : ",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()
```

PRACTICAL - 8

Aim :- Using linked list adding last and Before.

Theory :- **LINKED LIST :-**

- A linked list is a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each elements points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence. In its each most basic form, each node contains : data, and a reference to the next node in the sequence.

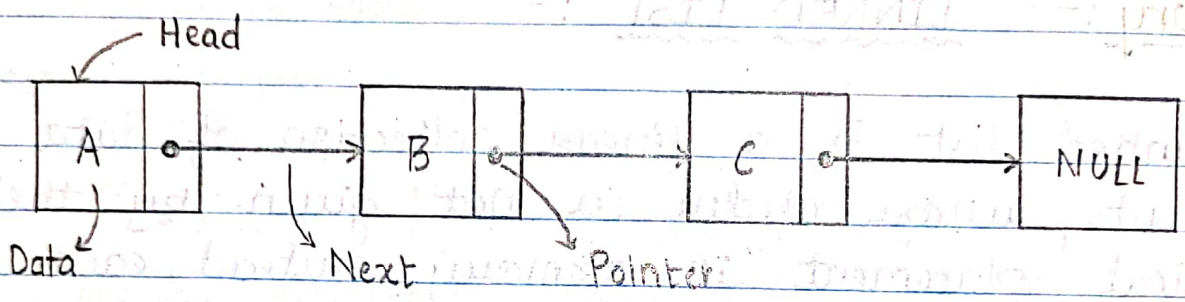
- This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.

- linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including lists, stacks, queues, associative arrays.

- Operations that can be performed on single linked list includes :-

1) Insertion - Adds an element
2) Deletion - Deletes an element
3) Traversal - Travel across the list

- **Representation of a linked list :-**



OUTPUT -

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20)
[MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
========== RESTART: C:/Users/Desktop/slinklist.py ==========
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 8

* SIMPLE LINKED LIST METHOD

Data : 20
Data : 30
Data : 40
Data : 50
Data : 60
Data : 70
Data : 80
>>>
```

SOURCE CODE -

```
print ("Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 9")

#POSTFIX EVALUATION METHOD
print ("\n* POSTFIX EVALUATION METHOD\n")

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))
     return stack.pop()

s="9 3 5 * + 9 - "
r=evaluate(s)
print("THE EVALUATED VALUE : ",r)
```

PRACTICAL - 9

Aim :- Evalute an expression using Postfix Evalution.

Theory :- POSTFIX EVALUATION :-

- The Postfix notation is used to represent algebric expressions. The expressions written in postfix from are evaluted faster compared to infix notation as parenthesis are not required in postfix. We have to convert infix to postfix.

- Algorithm for evaluation postfix expression :-

1) Create a stack to store operands (or values).
2) Scan the given expression and do following for every scanned element:
   a) If the element is a number, push it into the stack.
   b) If the element is a operator, pop operands for the operator from stack. Evalute the operator and push the result back to the stack.
3) When the expression is ended, the number in the stack is the final answer.

- Example : Expression " 9 3 5 * + 9 - "

1) Scan '9', it's a number, so push it to stack.
2) Scan '3', it's a number, so push it to stack.

3) Scan '5', it's a number, push it to stack.
4) Scan '*', it's a operation, pop two operands from stack, apply the * operator, we get 3*5 which results 15. We push the result '15' to the stack.
5) Scan '+', it's an operator, pop two operands from stack, apply the + operator, we get 9+15 which results 24. We push the result '24' to stack.
6) Scan '9', it's an number, we push it to the stack.
7) Scan '-', it's an operation, pop two operands from stack, apply the - operator, we get 24-9 which results 15. We push the result '15' to stack.
8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack.

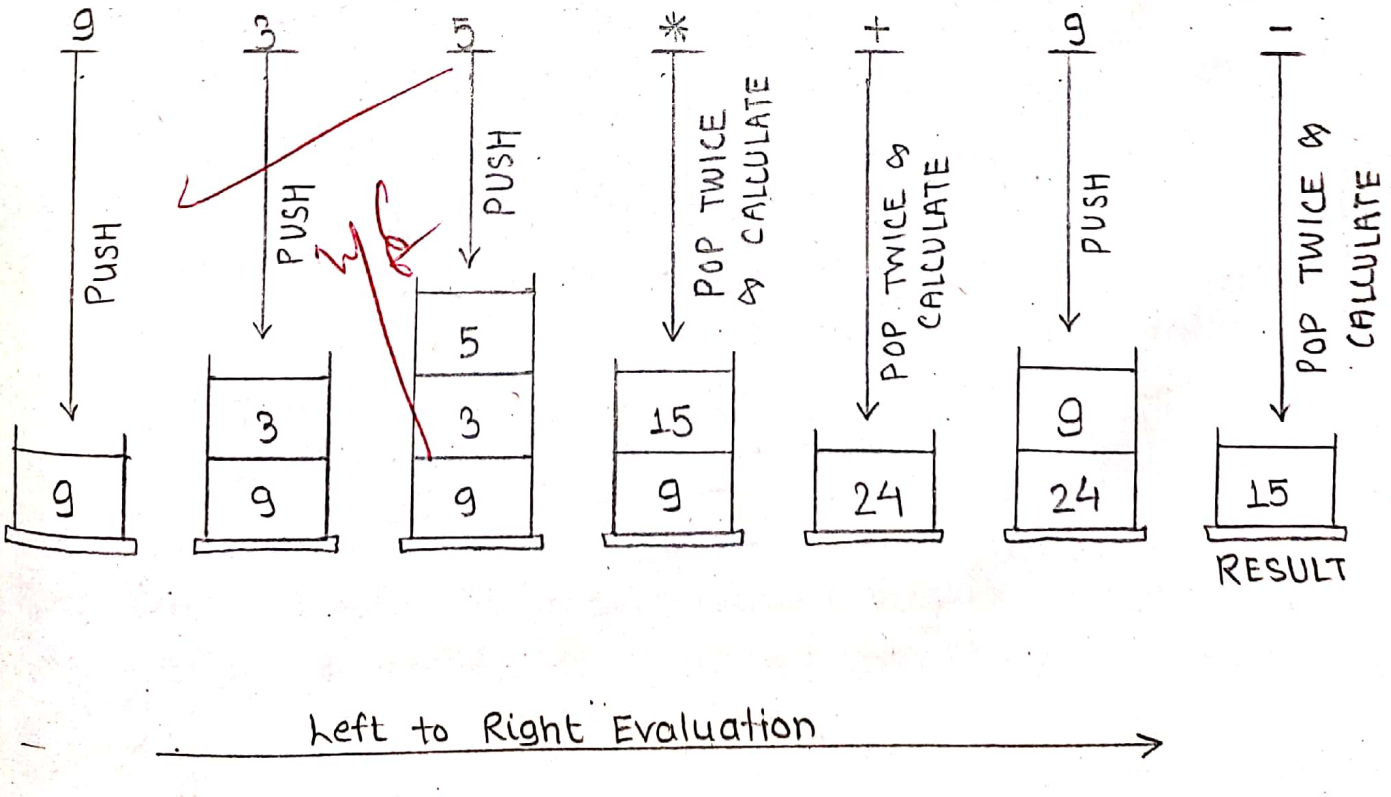
Step	symbol	Operation	Stack	Calculation
1.	9	Push	9	
2.	3	Push	9,3	
3.	5	Push	9,3,5	
4.	*	Pop (3,5)	9	3*5=15
5.		Push (15)	9,15	
6.	+	Pop (9,15)	Empty	9+15=24
7.		Push (24)	24	
8.	9	Push	24,9	
9.	−	Pop (24,9)	Empty	24-9=15
10.		Push (15)	15	Result

OUTPUT -

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20)
[MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
========== RESTART: C:\Users\Desktop\poseval.py ==========
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 9

* POSTFIX EVALUATION METHOD

THE EVALUATED VALUE : 15
>>>
```



SOURCE CODE -

```
print ("Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 10")

#QUICK SORT METHOD
print ("\n* QUICK SORT METHOD\n")

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 = [8,6,99,85,12,23,55,64,2,41,76]
print("THE UNSORTED LIST : ", alist)
quickSort(alist)
print("\nTHE QUICK SORTED LIST : ", alist)
```

PRACTICAL - 10

Aim :- Demonstrating the use of Quick Sort

Theory :- QUICK SORT :-

- QuickSort is Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

- There are many different versions of quicksort that pick pivot in different ways :

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

- They 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 and put all greater elements (greater than x) after x. All this should be done in linear time.

- **Analysis of QuickSort :-**

Time taken by QuickSort in general can be written as following.

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by Quicksort depends upon the input array and partition strategy.

Following are three cases :

Worst Case : $T(n) = T(0) + T(n-1) + \Theta(n)$
which is equivalent to
$T(n) = T(n-1) + \Theta(n)$
The above recurrence is $\Theta(n^2)$

Best Case : $T(n) = 2T(n/2) + \Theta(n)$
The recurrence is $\Theta(n \log n)$.

Average Case: $T(n) = T(n/9) + T(9n/10) + \Theta(n)$

OUTPUT -

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916
64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================= RESTART: C:\Users\Desktop\DS P10.py =================
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 10

* QUICK SORT METHOD

THE UNSORTED LIST : [8, 6, 99, 85, 12, 23, 55, 64, 2, 41, 76]

THE QUICK SORTED LIST : [2, 6, 8, 12, 23, 41, 55, 64, 76, 85, 99]
>>>
```

SOURCE CODE -

```
print ("Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 11")

#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=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 = [46,3,27,99,1,14,57,41,65,21,70]
print("THE UNSORTED LIST : ", arr)
mergeSort(arr)
print("\nTHE MERGE SORTED LIST : ",arr)
```

PRACTICAL - 11

Aim :- Demonstrating the use of MergeSort

Theory :- MERGE SORT :

- It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
- The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l...m] and arr[m+1...r] are sorted and merges the two sorted sub-arrays into one.
- Time Complexity : Sorting arrays on different machines. Merge sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \Theta(n)$$

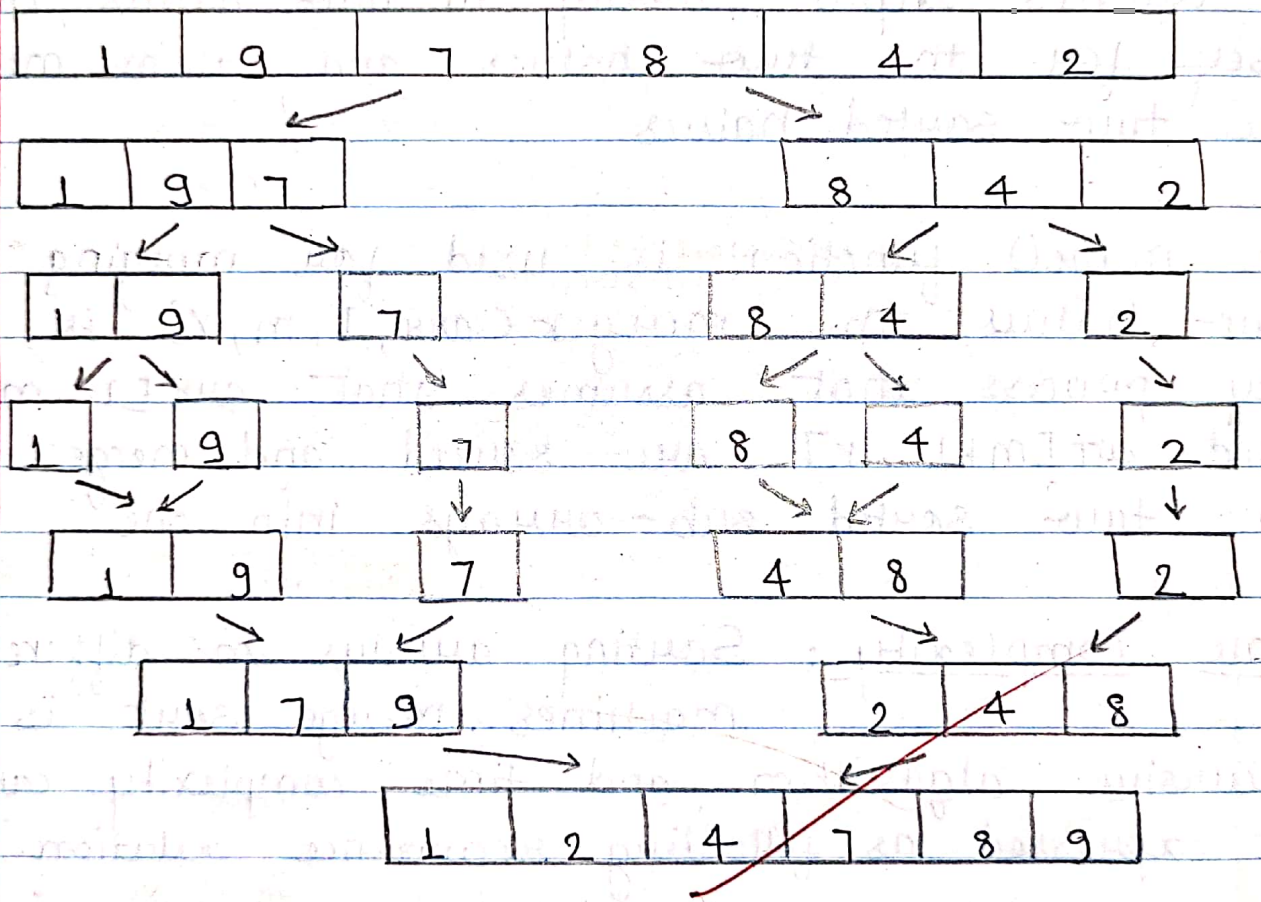
Recurrence $\Theta(n \log n)$.

- Applications of Merges Sort :

Merge sort is a sorting technique based on divide and conquer technique. With worse-case time complexity being $O(n \log n)$, it is one

of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Merge Sort example :-



OUTPUT -

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916
64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================= RESTART: C:\Users\Desktop\DS P11.py =================
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 11

* MERGE SORT METHOD

THE UNSORTED LIST : [46, 3, 27, 99, 1, 14, 57, 41, 65, 21, 70]

THE MERGE SORTED LIST : [1, 3, 14, 21, 27, 41, 46, 57, 65, 70, 99]
>>>
```

SOURCE CODE -

```
print ("Dinesh K Lohar\nFYBSC CS 1730\nPRACTICAL 12")

#BINARY TREE METHOD
print ("\n* BINARY TREE\n")

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)
            self.preorder(start.r)

    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)
```

PRACTICAL - 12

Aim:- Demonstrating the use of Binary tree.

Theory:- **BINARY TREE :-**

* A binary tree is a tree datastructure in which each node has at most two childrens, which are referred to as the left child and the right child.

* A recursive definition using just set theory notations is that a (non-empty) binary tree is a tuple (L,S,R), where L and R are binary trees or the empty set and S is a singleton set.

* **Types of Binary trees -**

1. Full Binary Tree - Every node other than leaf nodes has two child nodes.

2. Complete Binary Tree - All levels are filled except possibly the last one, and all nodes are filled in as far left as possible.

3. Perfect Binary Tree - All nodes have two childrens and all leaves are at the same level.

★ Binary Tree Traversal –

1. Pre-Order –
Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.

2. In-Order –
Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.

3. Post-Order –
Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively.

```
T=Tree()
T.add(50)
T.add(80)
T.add(40)
T.add(42)
T.add(20)
T.add(70)
T.add(100)
T.add(90)
T.add(110)
T.add(60)
T.add(10)
T.add(30)

print("* Preorder Numbers : ")
T.preorder(T.root)

print("* Inorder Numbers : ")
T.inorder(T.root)

print("* Postorder Numbers : ")
T.postorder(T.root)
```

OUTPUT -

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916
64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================== RESTART: C:\Users\Desktop\DS P12.py ==================
Dinesh K Lohar
FYBSC CS 1730
PRACTICAL 12

* BINARY TREE

80 Added on Right of 50
40 Added on Left of 50
42 Added on Right of 40
20 Added on Left of 40
70 Added on Left of 80
100 Added on Right of 80
90 Added on Left of 100
110 Added on Right of 100
60 Added on Left of 70
10 Added on Left of 20
30 Added on Right of 20

* Preorder Numbers :
50
40
20
10
30
42
80
70
60
100
90
110

* Inorder Numbers :
10
20
30
40
42
50
60
70
80
90
100
110

* Postorder Numbers :
10
20
30
40
42
60
70
80
90
100
110
50
>>>
```

* Example- A Binary Tree of a given Array

50, 80, 40, 42, 20, 70, 100, 90, 110, 60, 10, 30