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 _____ **UID No.** _____

Class FYCS(A) **Roll No.** 1705 **Year** _____

This is to certify that the work entered in this journal is the work of Mst. / Ms. Ayushmaan Kamokar.

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

Teacher In-Charge

Head of Department

Date : _____

Examiner

No.	
1.	Dra
	E
	Act
2.	Wri
	on
3.	Creat
	Writ
4.	Alter
5.	Using
	mathe
6.	Using
	fu
7.	Use
8.	Using
9.	Create
	Word

Semester 2 : DS

☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member's Signature
1.	Unsorted Search.	31	27.11.19	
2.	Sorted Search.	32	27.11.19	
3.	Binary Search.	33	27.11.19	
4.	Bubble Sort.	34.	4.12.19	
5.	Stack Queue	35.	8.1.20	
6.	Queue.	36.	8.1.20	
7.	Circular Queue.	37	8.1.20	
8.	Linked List	38.	15.1.20	
9.	Postfix Evaluation	39.	15.1.20	
*LS.	Impetus 19' — Library Issue.	—	20.1.20	
10.	Quick Sort.	40.	22.1.20	
11.	Merge Sort	41.	22.1.20	
12.	Binary Tree & TRAVERSAL	48.	5.2.20	

Code:-

```
File  Edit  Format  Run  Options  Window  Help
#Practical1 - UnsortedSearch
A=[2,3,8,6,4,1,9,5,7,0]
search=int(input("Enter No. To Search: "))
count=0
print("Ayushmaan Karmokar\n FYCS 1705")
for i in range(len(A)):
        if(search==A[i]):
            print("Found at Location: ",i)
            count=count+1
            break
if(count==0):
        print("Number Not Found.")
```

```
========== RESTART
Enter No. To Search: 5
Ayushmaan Karmokar
 FYCS 1705
Found at Location:  7
>>>
========== RESTART
Enter No. To Search: 10
Ayushmaan Karmokar
 FYCS 1705
Number Not Found.
>>>
```

Output:- ↑

No.

Semeste

27/11/19

Aim: To

Theory:

27/11/19

Experiment 1: Unsorted Search

Aim: To find the number in a list, using unsorted search.

Theory:

- The process of identifying or finding a particular record is called searching.
- There are 2 types of search:
  1. Linear Search
  2. Binary Search
- This linear search is further classified as:
  1. Sorted Search & 2. Unsorted 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, in unsorted search the elements are not arranged in a specific order and are in random mode.
- The process of Unsorted Search is therefore
- the data is arranged in list in a random variable
- Initializing 'count' as variable to set a counter for program.
- Asking the user for a number to search.
- Check the input with the data arranged sequentially and increment counter if found.
- Conditioning counter give appropriate output.

:8 27/11/19

Experiment 2: Sorted Search

Aim: To find number from a sorted list.

Theory: Sorting is basically a way in which the data in a random sequence is arranged in an ascending order.

Searching is to find one single element from a range of elements of data.

In Linear Sorted Search the data is arranged in ascending order and then is searched through it.

For a Linear Sorted Search:-

→ We have the data in an order.
→ User is asked for an element req. to search.
→ Using conditional statement it is checked, if the data is in the range.
→ If True the data is then passed onto a loop to check the location of the element in question.
→ Using appropriate condition, proper output is displayed.

Code:-

```
Practical2 - SortedSe
=[1,2,3,4,5,6,7,8,9,1
earch=int(input("Ent
rint("Ayushmaan Ka
f(search>=A[0] and
 for i in range(len(A
   if(search==A[i]):
        print("Foun
        break
lse:
 print("Number doe
```

```
Enter No. To Sear
Ayushmaan Karm
FYCS 1705
Number does not
>>>
Enter No. To Sea
Ayushmaan Karm
FYCS 1705
Found at Locatio
>>>
```

Output:

Code :-

```
#Practical2 - SortedSearch
A=[1,2,3,4,5,6,7,8,9,10]
search=int(input("Enter No. To Search: "))
print("Ayushmaan Karmokar\n FYCS 1705")
if(search>=A[0] and search<=A[len(A)-1]):
  for i in range(len(A)):
    if(search==A[i]):
        print("Found at Location: ",i)
        break
else:
  print("Number does not exist in range")
```

```
========== RESTART: E:
Enter No. To Search: 99
Ayushmaan Karmokar
 FYCS 1705
Number does not exist in range
>>>
========== RESTART: E:
Enter No. To Search: 10
Ayushmaan Karmokar
 FYCS 1705
Found at Location:  9
>>>
```

Output :- ↑

W/E

```
#!\Python3\bin\usr3
'''Binary Search Made Dated:01.12.2019. usingt Python3'''
#VariableIntialization
A=[5,10,20,25,35,40,45,50,55]
count=0
L=0
U=(len(A)-1)
search=int(input("Number to Search: ")) #Search(USER_INPUT)
#Loop_Starts_HERE
while(L<=U):
    M=(L+U)//2
    if(search>A[M]): #RIGHT_SECTION
        L=(M+1)
    elif(search<A[M]): #LEFT_SECTION
        U=(M-1)
    else:          #MID_SECTION
        count=1
        break
#Number_Checking
if(count==1):
    print("Number Found at: ",M)
else:
    print("number not Found")
```

```
==========RESTART: I
Number to Search: 20
Number Found at:  2
>>>
========== RESTART: I
Number to Search: 60
number not Found
>>>
```

27/11/19.

Experiment 3: Binary Search.

Aim:: To find an element from a sequence of data using binary search.

Theory: Binary search is a search a sorted array of elements by repeatedly dividing the elements of search interval in half. Begin with an interval covering the whole array.

If the value of the search key is less than the value of the the item in the middle of the interval, narrow the interval to the lower half.

Otherwise narrow the interval to the upper half.

Repeatedly check until the value is found or the interval is empty.

4/12

Experiment 4: Bubble Sort

Aim: To sort a random list of elements using bubble sort.

Theory: Bubble Sort, sometimes referred to as sinking sort, is a simple 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 and impractical for most problems even compared to insertion sort. Bubble Sort can be practical if the input is in mostly sorted order with some out of order elements nearly in position.

<code>

```
#Practical4-Bu
print("Ayushma
a=[5,3,8,19,12
print("Element
for dataset in
    for bubble
        if(a[b
            a[
print("Element
```

</code>

<output>

```
Python 3.8.0 (
D64)] on win32
Type "help", "
>>>
Ayushmaan Karn
FYCS 1705
Elements befo
Elements afte
>>>
```

</output>

<code>

```
#Practical4-BubbleSort
print("Ayushmaan Karmokar\nFYCS 1705")
a=[5,3,8,19,12,7,2,4,1]
print("Elements before sort: ",a)
for dataset in range(len(a)):
    for bubble in range(len(a)-(dataset+1)):
        if(a[bubble]>a[bubble+1]):
            a[bubble],a[bubble+1]=a[bubble+1],a[bubble]
print("Elements after bubble sort: ",a)
```

</code>

<output>

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: E:/Python/bubble-sort.py ==========
Ayushmaan Karmokar
FYCS 1705
Elements before sort:  [5, 3, 8, 19, 12, 7, 2, 4, 1]
Elements after bubble sort:  [1, 2, 3, 4, 5, 7, 8, 12, 19]
>>>
```

</output>

Code

```
# prac5_stack
print(
    stack:
        tos
        __init__(self):
        self.l=[0,0,0,0,0,0,0,0]
        self.tos=-1
        push(self,data):

        n=len(self.l)
        self.tos==n-1:
        print(
        :
        self.tos=self.tos+1
        self.l[self.tos]=data
        pop(self):
        self.tos<0:
        print(
        :
        k=self.l[self.tos]
        print( ,k)
        self.tos=self.tos-1

s=stack()
s.push(10)
s.push(20)
s.push(30)
s.push(40)
s.push(50)
s.push(60)
s.push(70)
s.push(80)

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

Output

```
====== RESTART: C:/Users/MOTHER RUSSIA/Desktop/PythonCodes/prac5_stack.py ======
Ayushmaan Karmokar.
FYCS1705
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty
>>>
```

Aim:

Theory

a

: Experiment 5:

: Stack Queue:

Aim: To perform data structure operation using stack.

Theory:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Consider an example of plates stacked over one another in canteen. The plate which is at the top is the first one to be removed, i.e the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO / FILO order.

Stack is an abstract data type that serves as a collection of elements, with two principal elements:

push, which adds element to the collection, and

pop, which removes the most recently added collection.

: Practical 6:

Queue Add & Delete.

Aim: To perform data structuring using Queue.

Theory:

A queue is a collection of entities that are maintained in a sequence and can be modified by addition of entities at one end of the sequence.

By convention, the end of the sequence at which elements are added is called back or rear of the queue and the end at which elements are removed is called head or front of the queue; analogously of the words used when people line up for goods or services.

The operation of 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.

<code>

```
#practical 6 - Queue
print(
    Queue:
        r
        f
    __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
    add(self,data):
    n=len(self.l)
      self.r<n-1:
        self.l[self.r]=data
        self.r=self.r+1
        :
        print(
    remove(self):
    n=len(self.l)
      self.f<n-1:
      print(self.l[self.f])
      self.f=self.f+1
        :
        print(

Q=Queue()
Q.add(30)
Q.add(40)
Q.add(50)
Q.add(60)
Q.add(70)
Q.add(80)

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

Output.

```
Python 3.8.0 (tags/v3.8
Type "help", "copyrigh
>>>

====== RESTART: C:/
>>>
Ayushmaan Karmokar
FYCS1705
Queue is full
30
40
50
60
70
Queue is empty
>>>
```

<Code>

```
#practical 6 - Queue
print( [illegible] )
    Queue:
        r
        f
      __init__(self):
     self.r=0
     self.f=0
     self.l=[0,0,0,0,0,0]
     add(self,data):
     n=len(self.l)
       self.r<n-1:
         self.l[self.r]=data
         self.r=self.r+1
         :
         print( Queue is full )
     remove(self):
      n=len(self.l)
        self.f<n-1:
         print(self.l[self.f])
         self.f=self.f+1
           :
          print( Queue is empty )

Q=Queue()
Q.add(30)
Q.add(40)
Q.add(50)
Q.add(60)
Q.add(70)
Q.add(80)

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

Output.

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>

====== RESTART: C:/Users/MOTHER RUSSIA/Desktop/PythonCodes/prac6_queue.py ======
>>>
Ayushmaan Karmokar.
FYCS1705
Queue is full
30
40
50
60
70
Queue is empty
>>>
```

<Code>

```
* Practical: Circular queue
print(
    Queue:
        r
        f
      _init_(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
      add(self,data):
    n=len(self.l)
      self.r<=n-1:
      self.l[self.r]=data
      print( ,data)
      self.r=self.r+1
        :
      s=self.r
      self.r=0
        self.r<self.f:
        self.l[self.r]=data
        self.r=self.r+1
          :
        self.r=s
        print( )
    remove(self):
  n=len(self.l)
    self.f<=n-1:
    print( ,self.l[self.f])
    self.f=self.f+1
      :
    s=self.f
    self.f=0
      self.f<self.r:
      print(self.l[self.f])
      self.f=self.f+1
        :
      print( )
      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)
```

Output:

```
==== RESTART: C:/Users/MOTHER RUSSIA/Desktop/PythonCodes/prac7_circqueue.py ====
Ayushmaan Karmokar.
FYCS 1705
data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
data removed: 44
>>>
```

:Practical 7:

Circular Queue

Aim: To perform data structuring using circular queue.

Theory:

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is called 'Ring Buffer'.

In a normal queue, we can insert elements until queue becomes full, But once queue becomes full, we cannot insert the next element even if queue ~~has~~ there is space in front of queue.

Operations are:

- Front - Get front item from queue.
- Rear - Get last item from queue.
- EnQueue(value) - function used to insert an element. New element is always inserted in rear.
- deQueue() - function used to delete an element. Element is always deleted from front position.

: Practical 8 :

: Linked List :

Aim: To perform data structure op using linked list.

Theory:

A linked list is a linear data structure, in which the elements are not stored at contiguous memory location.

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

① The size of array is fixed: So we must know the upper limit on the number of elements in advance. Also generally, the allocated memory is equal to the upper limit irrespective of the usage.

② Inserting a new element in an array of elements is expensive because the room has to be created for the new elements & to create room existing elements have to be shifted.

Advantages over arrays is
Dynamic size
Ease of insertion / Deletion.

Drawbacks:
Random Access is not allowed.
Not cache friendly.
Extra memory space for a pointer.

<code>

```
print("LINKED LIST\
class node:
    global data
    global next
    def __init__(se
        self.data=i
        self.next=N
class linkedlist:
    global s
    def __init__(se
        self.s=None
    def addL(self,i
        newnode=nod
        if self.s==
            self.s=
        else:
            head=self.
            while head
                head=he
            head.next=
    def addB(self,i
        newnode=nod
        if self.s==
            self.s=
        else:
            newnode
            self.s=
    def display(se
        head=self.
        while hea
            print
            head=h
        print (he
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()
```

</code>

<output>

```
Python 3.6.0 (tags
Type "help", "cop
>>>
==========
LINKED LIST

Ayushmaan Karmo
20
30
40
50
60
70
80
>>>
```

</output>

<Code>

```
print("LINKED LIST\n\nAyushmaan Karmokar. /nFYCS 1705")
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 (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()
```

<Output>

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/MOTHER RUSSIA/Downloads/linked-list.py ==========
LINKED LIST

Ayushmaan Karmokar. /nFYCS 1705
20
30
40
50
60
70
80
>>>
```

<Code>

```
print("Posttix Evaluation.\n\nAyushmaan Karmokar.\n1705")
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="3 8 4 / + 5 *"
r=evaluate(s)
print("The evaluated value is:",r)
```

<Output>

```
========== RESTART: C:/Users/MOTHER RUSSIA/Downloads/postfix.py ==========
=
Postfix Evaluation.

Ayushmaan Karmokar.
1705
The evaluated value is: 25
>>>
```

Aim:

Theory

: Practical 9 :

: Postfix Evaluation :

Aim: To perform operations using postfix Evaluation.

Theory:

The postfix notation is used to represent algebric expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Following algorithm for postfix evaluation:

① Create a stack to store operands.

② Scan the expression and do following for every scanned element:

→ If the element is a number, push in to stack.

→ If the element is a operator, pop operands for the operator from stack Evaluate the operator & push results back to stack.

③ When the expression is ended, the number in the stack is the final answer.

: Practical 10 :

: Quick Sort :

Theory:

QuickSort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partioning the elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

It is a comparison sort, meaning that it can sort items of any type for which a 'less-than' relation is defined. Efficient implementation of QuickSort is not a stable sort, meaning that relative order of equal sort items is not preserved.

On average the algorithm takes O(nlogn) comparison to sort n items.

<code>
print("Ayushmaa
def quickSort(alis
quickSortHelp
def quickSortHe
if first<last:
splitpoint=p
quickSortHe
quickSortHe
def partition(alis
pivotvalue=ali
leftmark=first
rightmark=las
done=False
while not don
while leftma
leftmark=
while alist[
rightmar
if rightmark
done=Tr
else:
temp=ali
alist[left
alist[righ
temp=alist
alist[first]=
alist[right
return righ
alist=[42,54,45
print("unsorte
quickSort(alis
print("Quicks
</code>

<output>
Python 3.8.
Type "help"
>>>
Ayushmaa
FYCS 1705
unsorted:
Quicksort:
>>>
</output>

Output

Code

```
# Quick Sort Practical
print("Ayushmaan Karmokar\nFYCS 1705")
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=[42,54,45,67,89,66,55,80,100]
print("unsorted: ",alist)
quickSort(alist)
print("Quicksort: ",alist)
```

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: G:/PG/quick_sort.py ==========
Ayushmaan Karmokar
FYCS 1705
unsorted:  [42, 54, 45, 67, 89, 66, 55, 80, 100]
Quicksort:  [42, 45, 54, 55, 66, 89, 67, 80, 100]
>>>
```

Output

Code

```
Merge Sort
print("Ayushmaan Karmokar \n FYCS 1705")
def sort(arr,l,m,r):
    n1=m-l+1
    n2=r-m
    L=[0]*(n1)
    R=[0]*(n2)
    for i in range(0,n1):
        L[i]=arr[l+i]
    for j in range(0,n2):
            R[j]=arr[m+1+j]
    i=0
    j=0
    k=l
    while i<n1 and j<n2:
        if L[i]<=R[j]:
            arr[k]=L[i]
            i+=1
        else:
            arr[k]=R[j]
            j+=1
            k+=1
        while i<n1:
            arr[k]=L[i]
            i+=1
            k+=1
        while j<n2:
            arr[k]=R[j]
            j+=1
            k+=1
def mergesort(arr,l,r):
    if l<r:
        m=int((l+(r-1))/2)
        mergesort(arr,l,m)
        mergesort(arr,m+1,r)
        sort(arr,l,m,r)
arr=[12,23,34,56,78,45,86,98,42]
print(arr)
n=len(arr)
mergesort(arr,0,n-1)
print(arr)
```

k+=1

```
========== RESTART: G:/PG/merger_sort.py ==========
Ayushmaan Karmokar
 FYCS 1705
[12, 23, 34, 56, 78, 45, 86, 98, 42]
[12, 23, 56, 56, 42, 45, 78, 86, 98]
>>>
```

Code

```
print("Ayushmaan Karmokar\n 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)
            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.postorder(start.l)
            self.postorder(start.r)
            print(start.data)
T=Tree()
T.add(100)
T.add(80)
T.add(70)
T.add(85)
T.add(10)
T.add(78)
T.add(60)
T.add(88)
T.add(13)
T.add(12)
print("1. preorder:")
T.preorder(T.root)
print("2. inorder:")
T.inorder(T.root)
print("3. postorder:")
T.postorder(T.root)
```

```
Ayushmaan Karmokar
 Binary Tree and Traversal
80 added on Left of 100
70 added on Left of 80
85 added on right of 80
10 added on Left of 70
78 added on right of 70
60 added on right of 10
88 added on right of 85
13 added on Left of 88
12 added on Left of 13
1. preorder:
```

:Practical 12:

Binary Tree — Traversal.

Theory:

A binary tree is a tree data structure in which each node has at most two children, which are referred to as Left child & Right child. A recursive definition using just set theory notion is that a binary tree is a tuple (LSR), where L & R are binary tree or the empty set & S is a singleton set.

Main application of trees include:

→ Manipulate hierarchical data.
→ Make information easy to search
→ Manipulate sorted list of data.

Unlike linear data structures, which have only one logical way to traverse them, trees can be traversed in different ways.

Depth First Traversal:

(a) Inorder (L, V, R) : 4 2 5 1 3
(b) Preorder (V, L, R) : 1 2 4 5 3
(c) Postorder (L, R, V): 4 5 2 3 1