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 Sequentia 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 method. That is what it calls unsorted linear Search.

Unsorted linear Search.

1- The data is entered in random
2- User needs to Specify the element to be Searched in the entered list
3- Check the condition that whether the entered number matches if the then display the location plus 1 as Data is Stored from location Zero.
4- If all elements are checked one by one element not found then prompt message number not found.

PRACTICAL - 1.

Input/Source Code:

sorted.py - C:/Us

File Edit Format Run Options Window Help

```
print("shaikh Rizwana \n 1718")


                    #LINEAR SEARCH UNSORTED
print("linear search sorted")
j=0
a=[2,6,7,8,9,10,11,14]
s=int(input("enter no to be searched: "))
print(s)
if(((s<a[0]) or (s>a[len(a)-1]))):
    print("doesnt exist")
else:
    for i in range(len(a)):
        if(s==a[i]):
            j=1
            print("number found at %d",i)
            break
    if(j==0):
        print("number not found")
```

038

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 & to display the Same.

In Searching that too in LINEAR SORTED Search the data is arranged in ascending to descending or descending to docending that do all what it meant by Searching through 'Sorted' that is well arranged data.

Sorted Linear Search:

1- The user is Supposed to enter data is sorted manner.
2- User has to given an element for Searching through Sorted list.
3- If element is found displaying with an updation as value is stored from location '0'.

- If data or element not found print the same.
- In sorted order list of element we can check the condition that whether the entered number lies from starting point till then without any processing we can say number not in the list.

PRACTICAL -2.

Source code:

```
#LINEAR SEARCH UNSORTED

print("linear search unsorted")
j=0
a=[34,45,67,89,45,34,56,78,89]
s=int(input("enter no to be searched: "))
print(s)
for i in range(len(a)):
    if(s==a[i]):
        print("number found at",i)
        j=1
        break
if(j==0):
        print("number not found")
```

output:

```
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>
```

Source code:

```
print("sheikh rizwana 1718")
a=[3,6,9,12,15,28,39]
print(a)
search=int(input("Enter number to be searched :"))
l=0
h=len(a)-1
m=int((l+h)/2)
if(search<a[l] or (search>a[h]):
print("Number not in RANGE!")
elif(search==a[h]):
print("number found at location :",h+1)
elif(search==a[l]):
print("number found at location :",l+1)
else:
while(l!=h):
if(search==a[m]):
print ("Number found at location:",m+1)
break
else:
if(search<a[m]):
 h=m
m=int((l+h)/2)
else:
  l=m
m=int((l+h)/2)
if(search!=a[m]):
                    print("Number not in list!")
break
```

Output:

Case1:

sheikh rizwana 1718
[3,6,9,12,15,28,39]
Enter number to be searched :12
Number found at location: 3

Case2

[12,15,18,20,23,26]
Enter number to be searched from the list:44
Number not in RANGE!

PRACTICAL - 03.

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 & the procedure branches into 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.

PRACTICAL - 4.

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 decending order.
BUBBLE Sort sometimes refered to as Sinking Sort.
Is a Simple Sorting algorithm that repeadtedly steps through the lists, compares adjacent elements & swaps them if they are in wrong order.
The pass through the list is repeated until the dist is Sorted. The algorithm which is a comparison sort is named for the way smaller or larger elements "bubble" to the top of the dist.
Although the algorithm is Simple, it is too slow as it compar one element checks if condition fails then only swaps otherwise goes on.

Source code:

```
print(" shaikh  rizwana1718")
a=[12,15,10,76,45]
pri,n,t("elements in list: \n ",a)
for passes in range (len(a)-1):
for compare in range (len(a)-1-passes):
 if(a[compare]>a[compare+1]):
temp=a[compare]
a[compare]=a[compare+1]
a[compare+1]=temp
print("After BUBBLE SORT elements list: \n",a)
```

Output:

```
sayed  tatheer 1719
 Elements in list:
[12,15,10,76,45]
After BUBBLE SORT elements list:
[10,12,15,45,76]
```

Example:

First pass

(5 1 4 2 8) → 1 5 4 2 8 Here algorithm compa
the first two elements & swap
since 5 > 1.

(1 5 4 2 8) → (1 4 5 2 8) swap since 5 > 4

(1 4 5 2 8) → (1 4 2 5 8) swap since 5 > 2.

(1 4 2 5 8) → (1 4 2 5 8) Now since these
element does not swap H

Second pass:

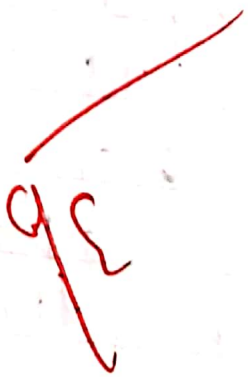
(1 4 2 5 8) → (1 4 2 5 8)

(1 4 2 5 8) → (1 2 4 5 8) swap since 4 > 2

(1 2 4 5 8) → (1 2 4 5 8)

Third pass:

(1 2 4 5 8) It checks & gives the data in sorted order.



AIM: To demonstrate the use of Stack.

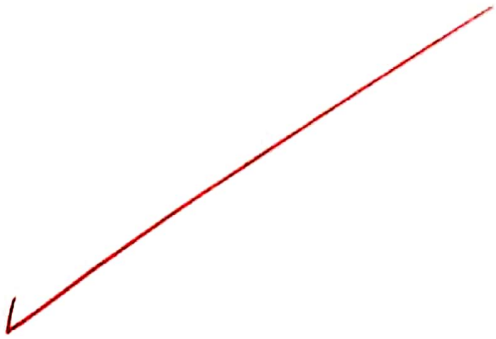
THEORY: In computer Science, a Stack is an abstract data types that Serves as a collection of elements with two principal operations push, which adds an element to the collection & pop, which removes the most recently added element that was not yet removed.
The order may be LIFO (last in first out) or FILO (first in last out).
Three Basic operations are performed in the Stack.

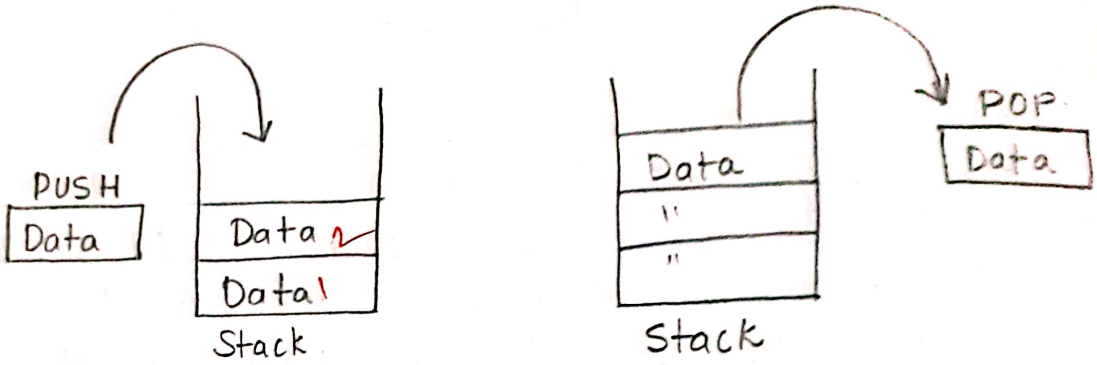
- PUSH: Adds an item in the if Stack. is full it is said to be over flow condition.
- POP: Removes an item from the Stack the items are popped in the reversed order in which they are pushed if the Stack is empty, then it is said to be an underflow condition.

Peek or TOP: Returns top element of stalk.
s Empty: Returns true if Stalk is empty else false.

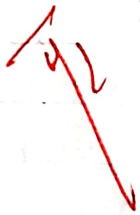
```
class stack:
    global tos
    def __init__(self):
        self.l=[0,0,0,0,0,0,0,0]
        self.tos=-1
    def push(self,data):
        n=len(self.l)
        if self.tos==n-1:
            print("stack is full")
        else:
            self.tos=self.tos+1
            self.l[self.tos]=data
    def pop(self):
        if self.tos<0:
            print("stack is empty")
        else:
            k=self.l[self.tos]
            print("data= ",k)
            self.tos=self.tos-1

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

a.pop()
a.pop()
a.pop()
a.pop()
a.pop()
a.pop()
a.pop()
a.pop()
```

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.19
Type "help", "copyright", "credits" or "license()" for more informatio
>>>
========== RESTART: C:/Users/RDX123/Desktop/ds/stack.py ==========
shaikh Rizwana
 1718
stack is full
data=  70
data=  60
data=  50
data=  40
data=  30
data=  20
data=  10
stack is empty
>>>
```





Last - in - first - out

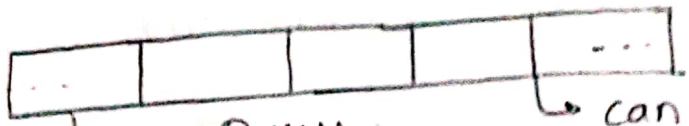


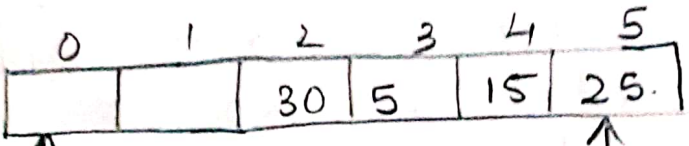
AIM: To demonstrate Queue add & delete.

THEORY: Queue is a linear data structure where the first element is inserted from one end called REAR & deleted from the other end called as FRONT.
Front points to the beginning of the Queue & Rear points to the end of the queue.
Queue follows the FIFO (First in first out) element inserted first will also be removed first.
In a queue, one end is always used to insert data & the other is used to delete data 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.

```
print("queue add and delete")
print("shaikh Rizwana \n 1718")
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 is full")
    def remove(self):
        n=len(self.l)
        if self.f<n-1:
            print(self.l[self.f])
            self.f=self.f+1
        else:
            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()
q.remove()
```

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: C:/Users/RDX123/Desktop/ds/queue add and delete.py ===
queue add and delete
shaikh Rizwana
 1718
queue is full
30
40
50
60
70
queue is empty
queue is empty
>>>
```





Front = 2
Rear = 5

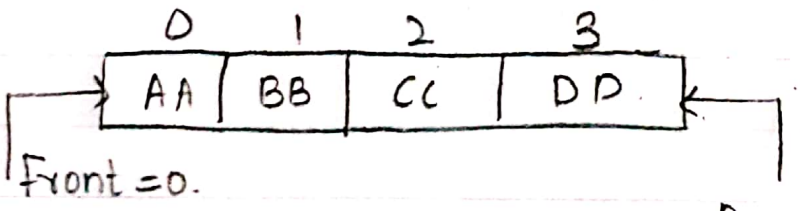
PRACTICAL - 7.

049

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

Theory: The queue that use implement using an array suffer from one limitation. In that implementation. there is a possibility that the queue is reported as full, even though in actuality there might be empty ends 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 & reach the end of the array. the next element is stored in the first stot of the array.

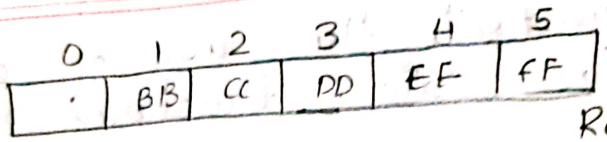
Example:



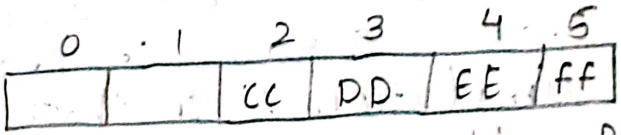
0 1 2 3

SOURCE CODE:

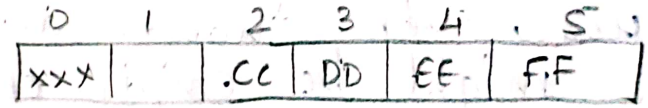
```
print("Rizwana \n 1718")
class Queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if (self.r<n-1):
      self.l[self.r]=data
      print("data added:",data)
      self.r=self.r+1
    else:
      s=self.r
      self.r=0
      if (self.r<self.f):
        self.l[self.r]=data
        self.r=self.r+1
      else:
        self.r=s
```



Front = 1. Rear = 5



Front = 2 Rear = 5.



Front = 2 Rear = 0

```
        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
  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 is 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)
```

OUTPUT:

Rizwana

1718

data added: 44

data added: 55

data added: 66

data added: 77

data added: 88

Queue is full

Data removed: 44

Source code:

```
Print("Shaikh Rizwana")
class node:
global data
global next
def __init__(self,item):
self.data=item
self.next=None
classlinkedlist:
global s
def __init__(self):
self.s=None
defaddL(self,item):
newnode=node(item)
ifself.s==None:
self.s=newnode
else:
head=self.s
whilehead.next!=None:
head=head.next
head.next=newnode
defaddB(self,item):
newnode=node(item)
ifself.s==None:
self.s=newnode
else:
newnode.next=self.s
self.s=newnode
def display(self):
head=self.s
```

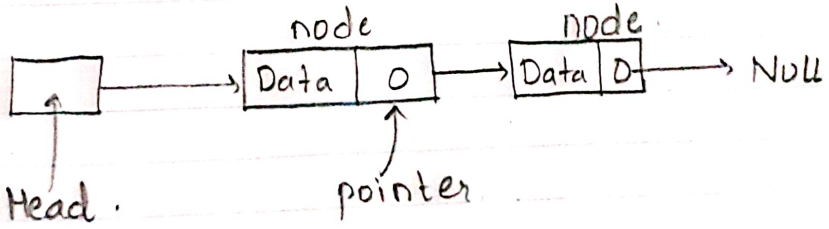
PRACTICAL - 8

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 which 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:

1 - Simple
2 - Doubly
3 - Circular.

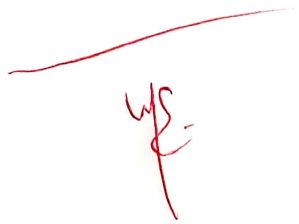
BASIC operations.

1 - Insertion.
2 - Deletion.
3 - Display.
4 - Search
5 - Delete.

```
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:

```
Shaikh Rizwana
20
30
40
50
60
70
80
```



AIM: To evaluati postfix expression using stack.

Theory: Stack is an (ADT) & works on LIFO i.e PUSH & POP operations.
A postfix expresion is a collection of operators & 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 oprations & Store the two popped operands in two different variables (operand 1 & operand 2) Then perform reading Symbol operation using operand Symbol operand using operand 1 & operand 2 & push result back on the stalk.
4- Finally perform a pop operation & display the popped value as final result.

```
Print("Shaikh Rizwana")
def evaluate(s):
    k=s.split()
    n=len(k)
  stack=[]
  fori 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 * +"
  r=evaluate(s)
  print("The evaluated value is:",r)
```

Output:

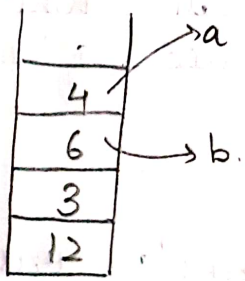
Shaikh Rizwana

The evaluated value is: 62

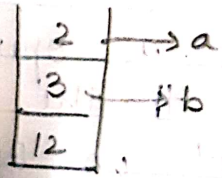
Value of postfix expression:

S = 12 3 6 4 - + *

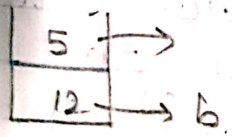
Stack:



b - a = 6 - 4 = 2 // store again in Stack.



b + a = 3 + 2 = 5 // Store result in Stack



b * a = 12 * 5 = 60

PRACTICAL – 10.

AIM : To 'evaluati' 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 & 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.

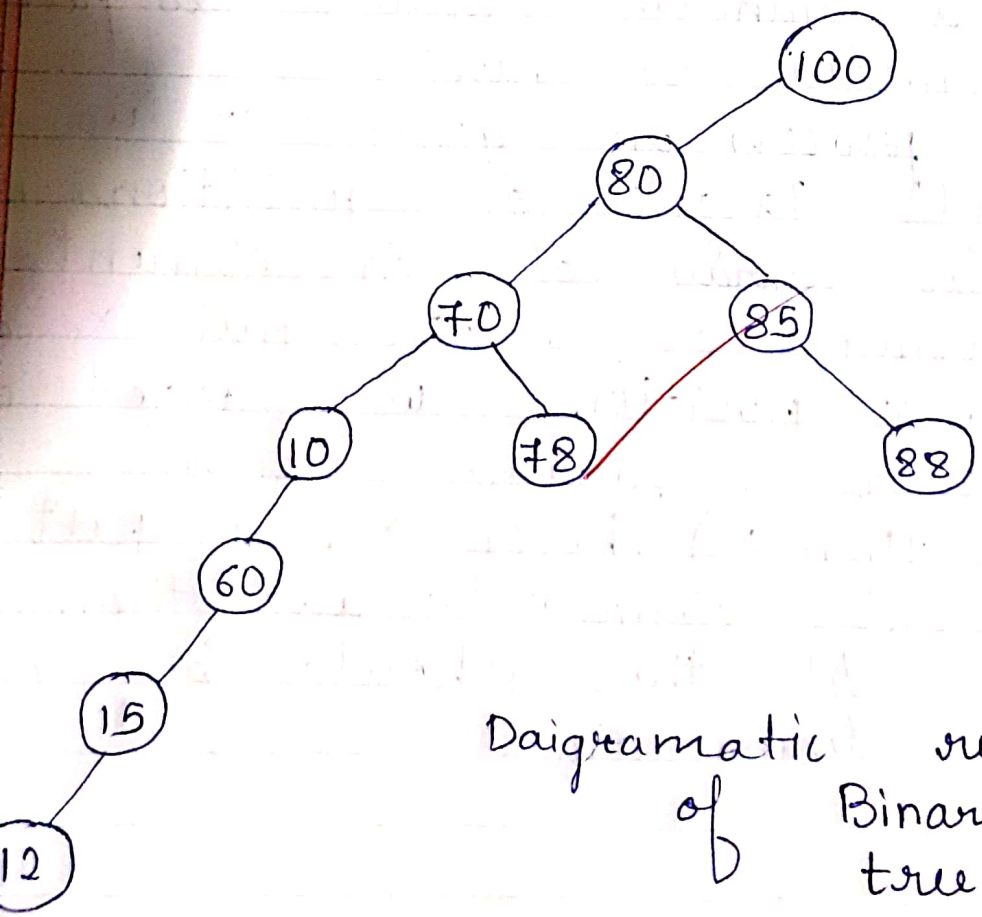
The key process in quicksort is partition () Target of partitions is given an array. & an element x of array as pivot, put x at its correct position in sorted array & put all smaller elements (smaller than x) before x, & put all greater element (greater than x) after x. All this should be done in linear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

PRACTICAL - 11

AIM: Binary tree & Traversal.

THEORY:

A binary tree is a special type of tree in which every node or vertex has either no child or one child node. A binary tree is an important class of a tree data structure in which a node can have at most two children.



Daigramatic representation of Binary Search tree.

```
Print(" shaikh rizwana /n 1718")
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)
    T=Tree()
    T.add(100)
    T.add(80)
    T.add(70)
    T.add(85)
    T.add(10)
```

Traversal is a process to visit all the node. of a tree & may print their values.

There are 3 ways which we use to traverse a tree.

1 - INORDER
2 - PREORDER
3 - POSTORDER.

INORDER :

The left-Subtree is visited 1st then the root & later the right Sub-tree we should also remember that every node may represent a Sub tree itself output produced is Sorted key values in ASCENDING ORDER

PRE-ORDER : The root node is visited 1st then left Subtree & finally the right Subt

POST-ORDER : The root node is visited last, left subtree then the right subtree & finally root node.

```
T.add(60)
T.add(88)
T.add(15)
T.add(12)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
```

Output:

```
Shaikh rizwana
1718
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 80
15 added on left of 60
12 added on left of 15
Preorder
100
80
70
10
60
```

```
Print ("shaikh rizwana /n 1718")
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+=l
        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)
```

Output:

Shaikh rizwana

1718

PRACTICAL - 12.

AIM: MERGE SORT.

THEORY:

Merge Sort is a sorting technique based on divide & conquer techniques with worst-case time complexity being $\theta(n \log n)$, is one of the most respected algorithm,

Merge Sort first divides the array into equal halves & then combines them in a sorted manner.

It divides input array in two halves, calls itself for the two halves & then merge the two sorted halves. The merge function is used for merging 2 halves. The merge (arr, l, m, r) is key process that assumes that arr [l ... m] & arr [m+1, ... r] are sorted & merges the two sorted sub-arrays into one