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 checks every element in the list Sequentially until the desired element is found. when the elements to be searches 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 Plus increment 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.

```
n=0
a=[4,9,15,38,50,67]
print(a)
search=input("enter no to be searched:")
for i in range(len(a)):
   if (search==a[i]):
      print "no found at",i
      n=1
      break
if(n!=1):
   print("entered no not found")
print("Nehal Tawade\n1737")
```

OUTPUT:

```
>>>[4, 9, 15, 38, 50,67 ]
enter no to be searched:4
no found at 0
Nehal Tawade
1737

>>>[4, 9, 15, 38, 50, 67]
enter no to be searched:35
entered no not found
Nehal Tawade
1737
```

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 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 updations 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.

PROGRAM CODE:

```
n=0
a=[1,32,48,74,80,10,87]
print(a)
search=input("enter no to be searched:")
if ((search<a[0]) or (search>a[len(a)-1])):
   print("enterd no doesn't exist")
else:
   for i in range(len(a)):
      if (search==a[i]):
         print "no found at",i
         n=1
         break
   if(n!=1):
      print("entered no not found")
print("Nehal Tawade\n1737")
```

OUTPUT:

```
>>>[1, 32, 48, 74, 80, 10, 87]
enter no to be searched:48
no found at 2
Nehal Tawade
1737

>>>[1, 32, 48, 74, 80, 10, 87]
enter no to be searched:100
entered no doesn't exist
Nehal Tawade
1737

>>>[1, 32, 48, 74, 80, 10, 87]
enter no to be searched:55
entered no not found
Nehal Tawade
1737
```

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 Comparision is made and 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 size in half, a binary search will complete successfully in logarithm time.

Source code:

```
print("Nehal Tawade \n1737")
a=[4,18,26,38,52,68,79]
print(a)
search=int(input("Enter number to be searched from the list:"))
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  given list!")
         break
```

Output:

Case1:
Nehal Tawade
1737
[4,18,26,38,52,68,79]

Enter number to be searched from the list:46
Number found at location: 4

Case2:
Nehal Tawade
1737
[4,18,26,38,52,68,79]

Enter number to be searched from the list:88
Number not in RANGE!

Case3:
Nehal Tawade
1737.
[4,18,26,38,52,68,79]

Enter number to be searched from the list:15
Number not in given list!

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 descending Order

BUBBLE Sort Sometimes referred to as Sinking Sort.

Is a Simple Sorting algorithm that repeatedly Steps through the lists, 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 Comparision 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 Compares one element checks if 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.

Source code:

```
print("Nehal Tawade \n1737")
a=[14,25,10,41,18,6]
print("Before BUBBLE SORT elemets 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 elemnts list: \n",a)
```

Output:

```
Nehal Tawade
1737
Before BUBBLE SORT elemets list:
 [14,25,10,41,18,6]
After BUBBLE SORT elemnts list:
 [6, 10, 14, 18, 25, 41]
```

Practical-5

AIM: To demonstrate the use of Stack

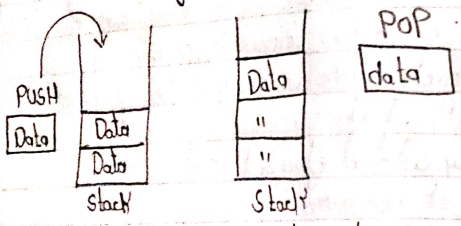
THEORY: In computer Science, a Stack is an abstract data type that Serves as a Collection of elements with two Principal operations Push, which adds an element to the Collection and PoP, which removes the most recently added element that was not yet removed.

The order may be LIFO (Last In, First out) or FILO (First In Last out

Three Basic operations are Performed in the Stack

- PUSH: Adds an item in the if the Stack is full then it 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 Spid to be an underflow Condition.

- Peek or Top: Returns top element of Stack
- isEmpty: Returns true if Stack is empty else false



Last_in_First_out


```
# Stack #
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 empty")
        else:
            k=self.l[self.tos]
            print("data=",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()
print("Nehal Tawade\n1737")
OUTPUT:
>>>stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty
Nehal Tawade
1737
```

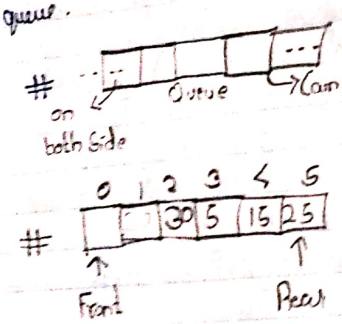
Practical-6

AIM: To demonstrate Queue add and delete

THEORY: Queue is a linear data Structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT
Front Points to the beginning of the queue and Rear Points to the end of the queue.
Queue follows the FIFO (First-in-First-out) Structure
According to its FIFO Structure element inserted first will also be removed first
In a queue, One end is always used to insert data (enqueue) and the other is used to delete data (dequeue) because queue is open 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.



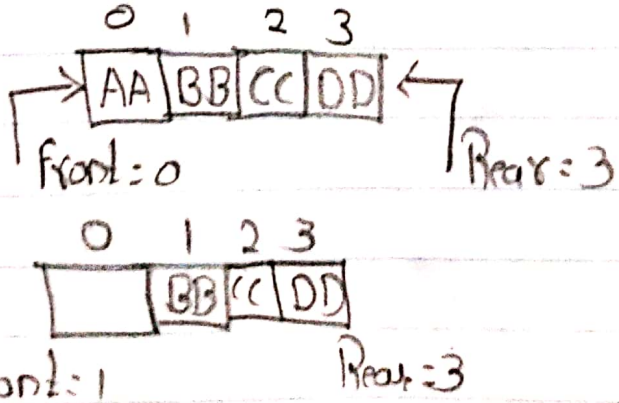

```
class Queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if self.r<n-1:
      self.l[self.r]=data
      self.r=self.r+1
    else:
      print("Queue is full")
  def remove(self):
    n=len(self.l)
    if 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()
print("Nehal Tawade \n 1737")
OUTPUT:
>>>Queue is full
30
40
50
60
70
Queue is empty
NEHAL TAWADE
 1737
```

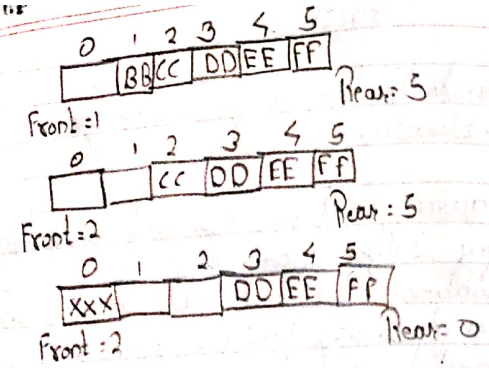
Practical-7

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 possibility 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.

Example:






```
#circular queue
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
    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)
print("Nehal Tawade\n 1737")
OUTPUT:
>>>data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
data removed: 44
NEHAL TAWADE
 1737
```

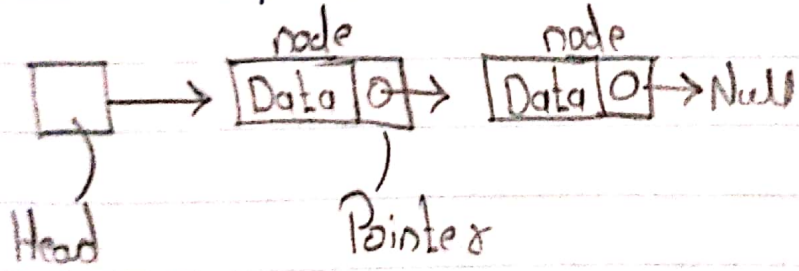
Practical-8

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. 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


```
#linked list
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()
print("Nehal Tawade\n1737")
OUTPUT:
>>>20
30
40
50
60
70
80
Nehal Tawade
1737
```

Practical-9

AIM: To evaluate Postfix expression using Stack

THEORY: Stack is an (ADT) and Works on LIFO (Last_in First-out) i.e PUSH & PoP operations

A Postfix expression is a Collection of operators and operands in which the operator is placed after the operands.

Steps to be followed:

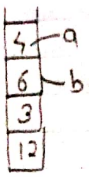
1. Read all the symbols one by one from left to right in the given Postfix expression
2. If the reading Symbol is operand then Push it on to the Stack.
3. If the reading Symbol is operator (+, −, *, /, etc.) then Perform TWO PoP operations and Store the two popped operands in two different Variables (operand 1 & operand 2. and Push result back on to Stack
4. Finally! Perform a PoP operation and display the popped Value as final result.

display the popped value as final result.

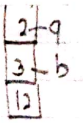
Value of Past fix expression:

5 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

```
#Postfix Evaluation

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="13 5 3 7 - + *"
r=evaluate(s)
print("The evaluated value is:",r)

print("Nehal Tawade \n 1737")
```

OUTPUT:

```
The evaluated value is: 13
Nehal Tawade
 1737
```

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 quick Sort 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.

```
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=[90,80,10,55,67,20]
quickSort(alist)
print(alist)
print("Nehal Tawade\n1737")
```

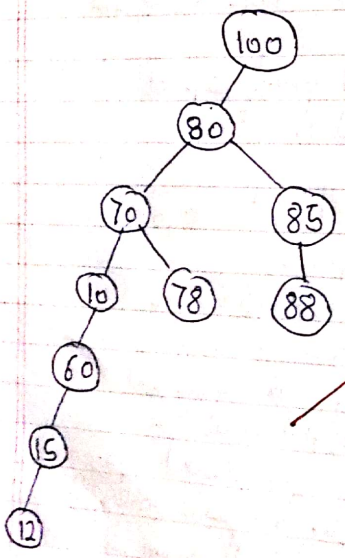
OUTPUT:

```
>>>[10, 80, 20, 55, 67, 90]
NEHAL Tawade
1737
```

Practical-11

Aim :- Binary tree and 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 or two child nodes. A binary tree is an important class of a tree is an data structure in which a node can have at most two children



Diagramatic Representation of Binary Search Tree

```
## 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.inorder(start.l)
      self.inorder(start.r)
      print(start.data)
```

```
T=Tree()
T.add(50)
T.add(40)
T.add(34)
T.add(33)
T.add(22)
T.add(60)
T.add(78)
T.add(11)
T.add(15)
T.add(12)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
print("Nehal Tawade \n 1737")
```

```
40 added on left of 50
34 added on left of 40
33 added on left of 34
22 added on left of 33
60 added on right of 50
78 added on right of 60
11 added on left of 22
15 added on right of 11
12 added on left of 15
preorder
50
40
34
33
22
11
15
12
60
78
inorder
11
12
15
22
33
34
40
50
60
78
postorder
11
12
15
22
33
34
40
60
78
50
Nehal Tawade
1737
```


Traversal is a process to visit all the nodes of a tree and may print their value too.
There are 3 ways which we use to traverse a tree.

- ↳ INORDER
- ↳ PREORDER
- ↳ POSTORDER

IN-ORDER :- The left Subtree is visited 1st then the root & later the right Sub tree. We should always remember that every node may represent a Subtree itself. output Product is Sorted Key values in ASCENDING ORDER

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

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

```
#MERGE SORT
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=[4,90,99,23,56,34,12,0]
print(arr)
n=len(arr)
mergesort(arr,0,n-1)
print(arr)
print("Nehal Tawade\n 1737")
```

OUTPUT:

```
[4, 90, 99, 23, 56, 34, 12, 0]
[0, 4, 90, 23, 99, 34, 56, 12]
NEHAL Tawade
 1737
```


Practical - 12

Aim :- Merge Sort

THEORY: Merge Sort is a Sorting technique based on divide and conquer technique with worst-case time Complexity being $O(n \log n)$, it one of the most respected algorithm.

Merge Sort first divides the array into equal halves and then Combines them in a Sorted manner.

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 2 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-array into one.