Practical 1

Aim: To search a number from the list using linear method

Theory: The process of identifying or finding a particular record is called searching

There are two types of search

Linear search

Binary search

The linear search is further classified as

SORTED

UNSORTED

Here we will look on the UNSORTED Linear search

Linear search, also known as sequential search, is a process that check every element in the list sequentially unit 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 plus increment 1 as data is stored from location zero.
If all elements are checked on by one and element not found then prompt message number not found.

Code :

```
print("Dhaval 1765")
f=False
a=[1,2,3,4,5,6,7,8,9]
s=int(input("Enter the number to find"))
for i in range (len(a)):
  if(s==a[i]):
    print("Number found")
    f=True
    break;
if (f==False):
  print("Number not found")
```

Output :

```
Dhaval 1765
Enter the number to find6
Number found
>>>
 RESTART: C:/Users/Dhaval/AppData/Local/Programs/Python/Python37-32/1765/dspractical 1.py
Dhaval 1765
Enter the number to find10
Number not found
>>>
```

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 well arranged data.

SORTED Linear search.

The user is supposed to enter data in sorted manner.

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

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

If data or element not found print the same.

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

Code:

```
print("Dhaval 1765")
f=False
a=[2,6,19,22,26,28,33,36]
s=int(input("Enter the number to find"))
if(s<a[0] or s>a[len(a)-1]):
   print("Number does not exist")
else:
   for i in range(len(a)):
      if(s==a[i]):
         f=True
         print("Number found at index",i)
         break
   if(f==False):
         print("Number is not found")
```

Output:-

```
Dhaval 1765
Enter the number to find2
Number found at index 0
>>>
 RESTART: C:/Users/Dhaval/AppData/Local/Programs/Python/Python37-32/1765/dspractical 2.py
Dhaval 1765
Enter the number to find26
Number found at index 4
>>>
 RESTART: C:/Users/Dhaval/AppData/Local/Programs/Python/Python37-32/1765/dspractical 2.py
Dhaval 1765
Enter the number to find40
Number does not exist
```

Code :

```
print("Dhaval 1765")
a=[2,7,9,13,16,17,23]
s=int(input("Enter number to be find from list"))
l=0
h=len(a)-1
m=int((l+h)/2)
if((s<a[l]) or (s>a[h])):
   print("Number not in range")
elif(s==a[h]):
   print("Number Found at location",h+1)
elif(s==a[l]):
   print("Number found at location",l+1)
else:
   while(l!=h):
      if(s==a[m]):
         print("Number found at location",m+1)
         break
      else:
         if(s<a[m]):
            h=m
            m=int((l+h)/2)
         else:
```

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. 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 segment of the array until the value is corated.

Because each step in the algorithm divides the array size in half a

binary search will complete successfully in logarithmic time.

Scanned with CamScanner

Output:

```
Dhaval 1765
Enter number to be find from list13
Number found at location 4
>>>
 RESTART: C:/Users/Dhaval/AppData/Local/Programs/Python/Python37-32/1765/dspractical 3.py
Dhaval 1765
Enter number to be find from list24
Number not in range
>>>
 RESTART: C:/Users/Dhaval/AppData/Local/Programs/Python/Python37-32/1765/dspractical 3.py
Dhaval 1765
Enter number to be find from list10
Number not in given list
```

Code

```
print("Dhaval Prakashkar\nRoll no:176")
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 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
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()
```

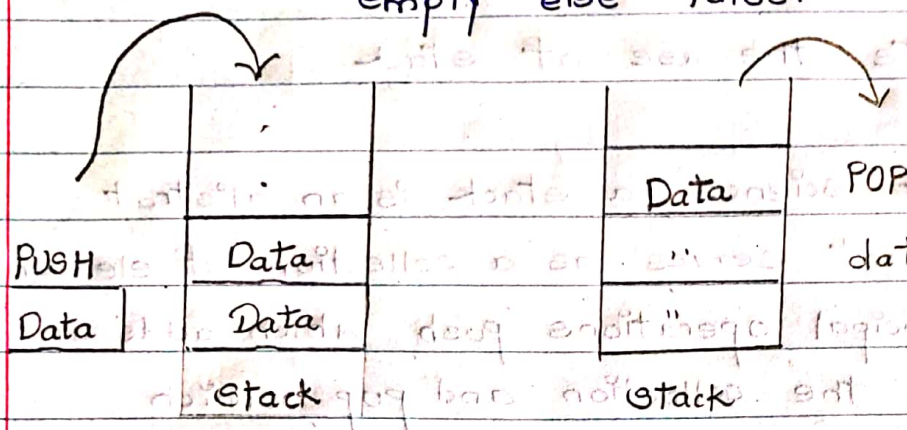
Practical 4

Aim : To demonstrate the use of stack

Theory : In computer science, a stack is an alistrat 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 operation are performed in the stack.

- PUSH : Adds an item in the if stack if full then it is said to be over flow condition.
- POP : Removes on 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 stack.

Empty : Returns true if stack is empty else false.



Last_in - First - Out

Output

```
Dhaval Prakashkar
Roll no:176
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack is empty
>>>
```

Code

```
print("Dhaval Prakashkar\nRoll No:1765")
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()
```

Practical 5

Aim: To demonstrate Queue add and delete

Theory: Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the other end called as FRONT.
Front points to the beginning of the queue and Rear points to the end of the queue.
Queue follows the FIFO (First_in_First Out) structure.
According to its FIFO structure element inserted first will also be removed first
In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue) because queue is open at both of its ends.
enqueue() can be traced 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.

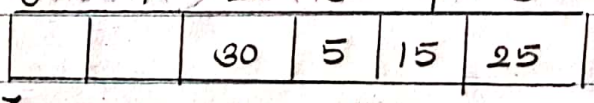
Front is used to get the front data item from a queue

Rear is used to get the last item from a queue.

One end for both sides

Queue

Can have end



Front = 2

Rear = 5

Output :-

```
Dhaval Prakashkar
Roll No:1765
Queue is Full
30
40
50
60
70
Queue is Empty
>>>
```

Code

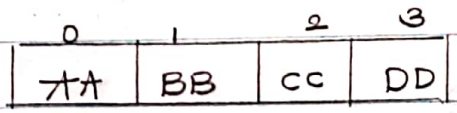
```
print("Dhaval Prakashkar\nRoll No:1765")
class Queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if self.r<=n-1:
      self.l[self.r]=data
      print("data added",data)
      self.r=self.r+1
    else:
      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)
```

Practical 6.

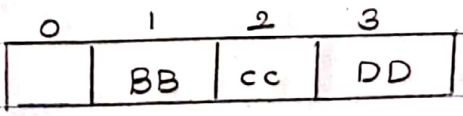
Aim: To demonstrate the use of circular queue in data-structure

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 is 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 end of the array. The next element is ~~stored~~ stored in the first slot of the array.

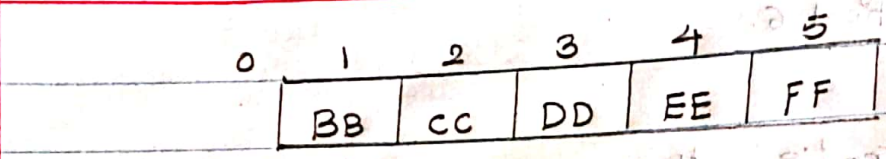
Example:



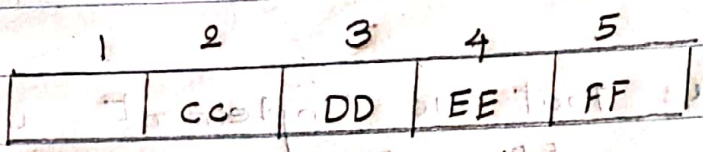
Front = 0 Rear = 3



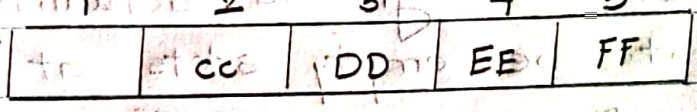
Front = 1 Rear = 3



Front = 1 Rear = 5



Front = 2 Rear = 5



Front = 2 Rear = 5

Output:-

```
Dhaval Prakashkar
Roll No:1765
data added 44
data added 55
data added 66
data added 77
data added 88
data added 99
data removed 44
>>>
```

```
print("Name : Dhaval Prakashkar \nDiv : A \nRoll no : 1765 \nBatch : B")
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)
```

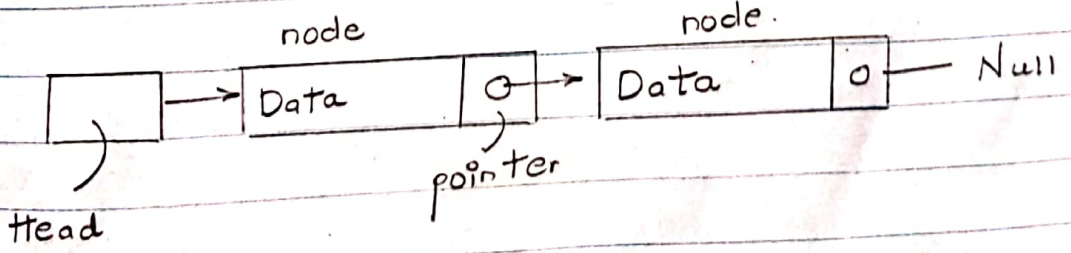
Practical- 8

Aim: To demonstrate the use of Linked List in data structure.

Theory: A linked list is a sequence of data structure 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 operation

- Insertion
- Deletion
- Display
- Search
- Delete

```
start.addL(60)
start.addL(70)
start.addL(80)
start.addB(40)
start.addB(30)
start.addB(20)
start.display()
```

```
>>> ================================ RESTART ================================
>>>
Name : Dhaval Prakashkar
Div : A
Roll no : 1765
Batch : B
20
30
40
50
60
70
80
>>>
```

```
print("Name : Dhaval Prakashkar \nDiv : A \nRoll no : 1765 \nBatch : B")
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=input("Enter a postfix with spaces between each element : ")
r=evaluate(s)
print("The evaluated value is ",r)
```

Practical-9

Aim: To evalute postfix expression using stack.

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

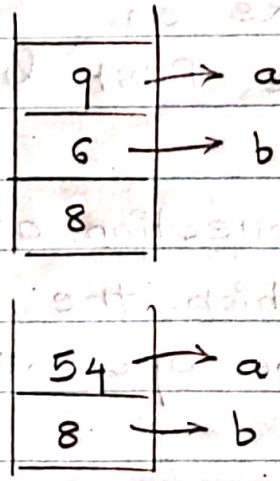
Steps to be followed

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 operation and store the two popped operands in two different variable (operand 1 & operand 2). Then perform reading symbol operation using operand 1 & operand 2 and push result back on to the stack.
4. Finally! Perform a pop operation and display the popped value as final result.

Value of postfix expression

S = 8 6 9 * +

Stack:



b * a = 6 × 9 = 54

b + a = 54 + 8 = 62

```
>>> ================================ RESTART ================================
>>>
Name : Dhaval Prakashkar
Div : A
Roll no : 1765
Batch : B
The evaluated value is  62
>>>
```

```
print("Dhaval Prakashkar\nRoll No:1765")
a=[20,19,18,17,16]
print(a)

for i in range(len(a)-1):
    for j in range(len(a)-1):
        if(a[j]>a[j+1]):
            t=a[j]
            a[j]=a[j+1]
            a[j+1]=t
print(a)
```

Practical - 10

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

Theory: SORTING is type in which any random data is sorted i.e arranged in ascending or descending order.

BUBBLE - sort sometimes referred it as sinking sort.

It is a simple sorting algorithm that repeatedly steps through the lists, compares adjacent elements and swaps them if they are in wrong orde.

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 to slow as it compares one element checks if condition fails then only swaps otherwise goes on.

Example:

First pass

(5 1 4 2 8) → (1 5 4 2 8) Here algorithm compares the first two elements and swaps 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

(1 4 2 5 8) → (1 4 2 5 8) Now since these elements are already in order (8>5) algorithm does not swap them

Second pass:

(1 4 2 5 8) → (1 4 2 5 8)

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

(1 2 4 5 8) → (1 2 4 5 8)

Third pass

(1 2 4 5 8) It check and gives the data in sorted order.

```
Dhaval Prakashkar
Roll No:1765
[20, 19, 18, 17, 16]
[16, 17, 18, 19, 20]
>>>
```

```
print("Dhaval Prakashkar\n Roll No:1765")
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]
quickSort(alist)
print(alist)
```

Practical- 11

Aim: To evaluate i.e to sort the given data in Quick sort.

Theory: Quicksort is an efficient sorting algorithm Type of a Divide & conquer algorithm It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick sort that pick pivot in different ways.

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

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

```
Dhaval Prakashkar
 Roll No:1765
[42, 45, 54, 55, 66, 89, 67, 80, 100]
>>>
```

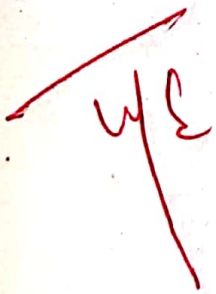
```
print("Dhaval Prakashkar\nRoll No:1765")
a=[21,20,19,18,17,16]
print(a)

for i in range(len(a)-1):
    for j in range(len(a)-1):
        if(a[j]>a[i+1]):
            t=a[j]
            a[j]=a[i+1]
            a[i+1]=t
print(a)
```

Practical - 12

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

Theory: Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison based algorithm in which the list is divided into two part the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes a part of sorted array. This process continues moving unsorted array boundary by one element to the right.
This algorithm is not suitable for large data set as its average and worst case complexities are of $O(n^2)$ where n is the number of items.

```
Dhaval Prakashkar
Roll No:1765
[21, 20, 19, 18, 17, 16]
[16, 17, 18, 19, 20, 21]
>>>
```



```
print("Dhaval Prakashkar\nRoll No:1765")
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
```

Practical - 13.

Topic: Binary Tree

Theory: Binary tree is a tree which supports maximum of 2 children for any node within the tree thus any particular node can have either 0 or 1 or 2 children.

Leaf Node: Nodes which do not have any children.

Internal Node: Nodes which are non-leaf nodes.

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

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

Inorder: i) Traverse the left subtree. The left subtree inturn might have left and right subtree.
ii) Visit the root node.
iii) Traverse the right subtree. The

right subtree inturn might have left and right subtrees.

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

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

iii) Visit the root node.

```
            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.l)
    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(90)
T.add(80)
T.add(75)
T.add(20)
T.add(68)
T.add(50)
T.add(77)
T.add(25)
T.add(12)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
```