Sem-II

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:
(i) Linear Search (ii) 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 searched are not specifically arranged in ascending or descending order.

→ They are arranged in random manner. That is what is called unsorted Linear Search.

Unsorted Linear Search:-

→ The Data is entered in random search.

→ 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

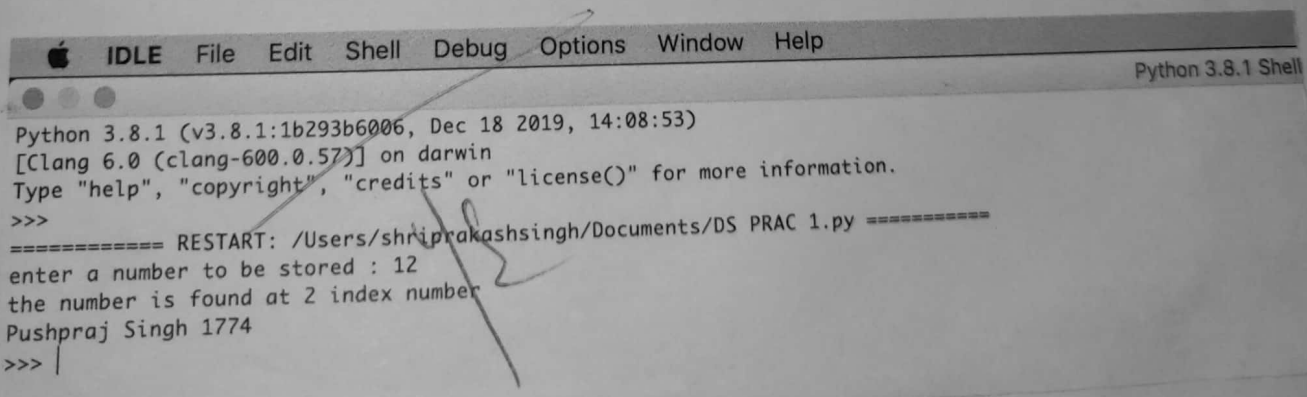
plus increment 1 as data is stored from location zero.

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

Source Code:-

```
found = False
a=[24, 6, 12, 4, 15]
search=int(input("Enter a number to be stored: "))
for x in range(len(a)-1):
    if(search==a[x]):
        print("The number is found at",x,"index number)
        found = True
        break;
if(found==False):
    print("number is not present in the list")
print("Pushpraj Singh","1774")
```

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 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 means by searching through 'sorted' that is well arranged data.

Sorted Linear Search:

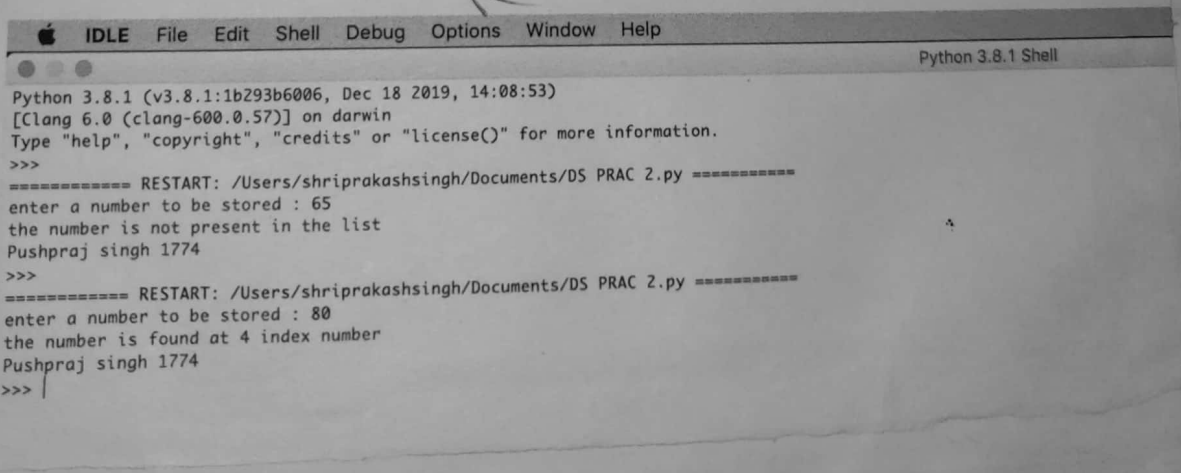
→ The user is supposed to enter data in sorted manner.
→ User has to give an element for searching through sorted list.
→ If element is found, display with an updation as value is sorted from location '0'.
→ If data or element not found print the same.

→ 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.

SOURCE CODE:-

```
found = False.
a=[2,34,56,67,80]
search = int(input("enter a number to be stored: "))
if(search < a[0] or search > a[len(a)-1]):
    print("number does not exist")
else:
    for x in range(len(a)):
        if(search = a[x]):
            print("The number is found at",x,"index number)
            found = True
            break;
if(found == False):
    print("number not present in the list")
print("Pushpraj Singh","1774")
```

OUTPUT:



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 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 logarithmic time.

Source Code:-

IDLE File Edit Format Run Options Window Help

*DS PRAC3.py - /Users/shriprakashsingh/Docume

```
a=[10,25,50,65,90]
print("PUSHPRAJ SINGH","1774")
print("a")
search=int(input("enter a number to be searched : "))
lb=0
ub=len(a)-1
while(True):
    m=(lb+ub)//2
    if(lb>ub):
        print("CASE.2-UNSUCCESSFULL SEARCH")
        print("number not found")
        break;
    if(search==a[m]):
        print("CASE.1-SUCCESSFULL SEARCH")
        print("number is found at",m,"index number")
        break;
    else:
        if(search<a[m]):
            ub=m-1
        else:
            lb=m+1
```

u=m-1

else:

l=m+1.

IDLE File Edit Shell Debug Options Window Help

Python 3.8.1 Shell

```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /Users/shriprakashsingh/Documents/DS PRAC3.py ==========
PUSHPRAJ SINGH 1774
a
enter a number to be searched : 34
CASE.2-UNSUCCESSFULL SEARCH
number not found
>>>
========== RESTART: /Users/shriprakashsingh/Documents/DS PRAC3.py ==========
PUSHPRAJ SINGH 1774
a
enter a number to be searched : 65
CASE.1-SUCCESSFULL SEARCH
number is found at 3 index number
>>>
```

Practical: 4

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 lie LIFO (last in first out) or FILO (First in Last out).

→ Three Basic Operations Performed in the Stack are:

1.) **Push (PUSH):** Adds an item in the stack. If the Stack is full, then it is said to be overflow condition.

2.) **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.

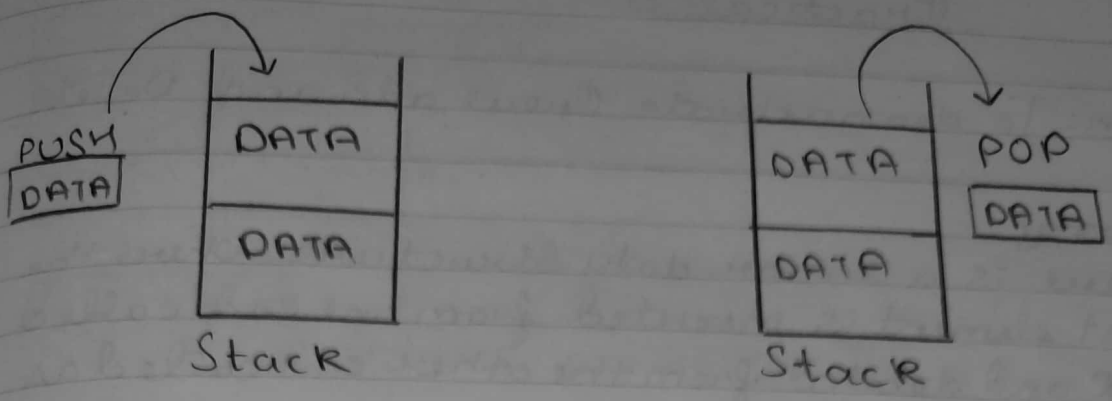
3) **TOP:** Returns top elements of Stack.

4) **isEmpty:** Returns true is Stack is empty else false.

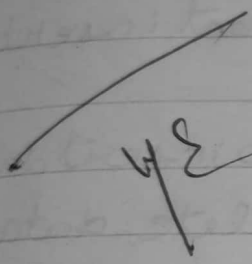
```
##Stack##
print("Pushpraj Singh","1774")
class stack:
  global tos
  def __init__(self):
     self.l=[0,0,0,0,0,0,0]
     self.tos=-1
  def push(self,data):
     n=len(self.l)
     if self.tos==n-1:
        print("stack is full")
     else:
        self.tos=self.tos+1
        self.l[self.tos]=data
def pop(self):
   if self.tos<0:
      print("stack 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()
```

OUTPUT:

```
Pushpraj Singh 1774
stack is full
data = 70
data = 60
data = 50
data = 40
data = 30
data = 20
data = 10
stack empty
```



Last in first Out



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 last 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 ie 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.

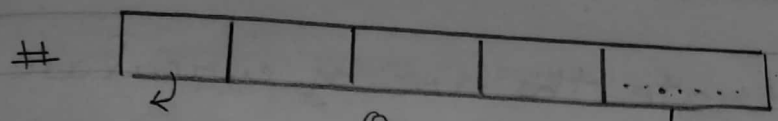
Rear is used to get the last item from a queue.

```
##Queue and add Delete##
print("PUSHPRAJ SINGH","1774")
class Queue:
   global r
   global f
   def __init__(self):
       self.r=0
       self.f=0
       self.I=[0,0,0,0,0,0]
   def add(self,data):
           n=len(self.I)
           if self.r<n-1:
               self.I[self.r]=data
               self.r=self.r+1
           else:
               print("Queue is full")

   def remove(self):
       n=len(self.I)
       if self.f<n-1:
           print(self.I[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()
```

OUTPUT:

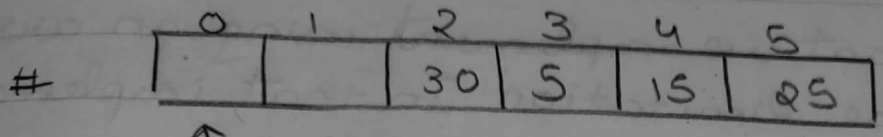
```
PUSHPRAJ SINGH 1774
data added :  44
data added :  55
data added :  66
data added :  77
data added :  88
data added :  99
data removed :  44
>>>
```



on both sides

Queue

can have ends

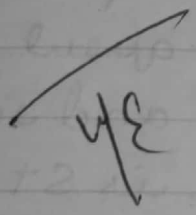


front = 2

Rear = 5

FRONT

REAR



Practical: 5

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

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.

```
#(circular queue)
print("PUSHPRAJ SINGH","1774")

class Queue:
    global r
    global f
    def __init__(self):
        self.r=0
        self.f=0
        self.I=[0,0,0,0,0,0]
    def add(self,data):
        n=len(self.I)
        if self.r<=n-1:
            self.I[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.I[self.r]=data
                self.r=self.r+1
            else:
                self.r=s
                print("Queue is full")
  def remove(self):
      n=len(self.I)
      if self.f<n-1:
          print("data removed : ",self.I[self.f])
          self.f=self.f+1
      else:
          s=self.f
          self.f=o
          if self.f<self.r:
              print(self.I[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: 7

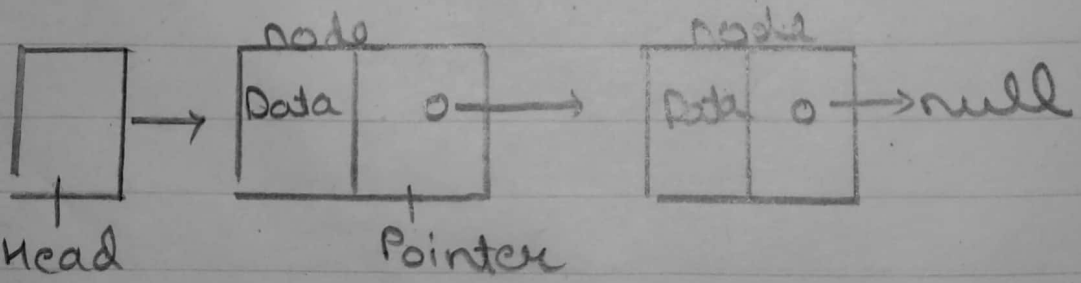
Aim: To demonstrate the use of linked list in data structure.

Theory:

→ A linked list is a sequence of data structure linked list is a sequence of links which contain 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:**



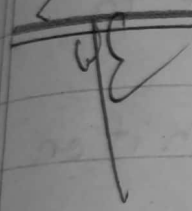
```
print("PUSHPRAJ SINGH,1774")
class node:
  global data
  global next
  def __init__(self,item):
        self.data=item
        self.next=None
class linkedlist:
  global s
  def __init__(self):
    self.s=None
  def addL(self,item):
    newnode=node(item)
    if self.s==None:
      self.s=newnode
    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()
```

Types of Linked List:-

- simple
- Doubly
- circular

Basic Operations:

- Insertion
- Deletion
- Display
- Search
- Delete



OUTPUT:

PUSHPRAJ SINGH,1774
20
30
40
50
60
70
80

Practical: 8

Aim: To evaluate postfix expression using stack.

Theory:

→ Stack is an (ADT) and works on LIFO i.e PUSH and POP operations.

→ A postfix expression is a collection of operator and operands in which the operator is placed after the operands.

Steps:

→ Read all the symbols one by one from left to right in the postfix expression.

→ If the reading symbol is operand then push it o the stack.

→ If the reading symbol is operator (+, -, *, / , e then perform Two POP operations and store the t operands in two different variables (operand 1 operand 2) then perform reading symbol opera using operand 1 & operand 2 and push re back on to the stack.

→ finally, perform a pop operation and display popped value as final result.

```
print("PUSHPRAJ SINGH 1774")
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(a)*int(b))
        else:
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)/int(a))
    return stack.pop()
s="8 6 9 * +"
r=evaluate(s)
print("the evaluated values is: ",r)
```

OUTPUT:

```
PUSHPRAJ SINGH 1774
the evaluated values is:  62
>>>
```

Practical: 9

Aim: To demonstrate working of Bubble Sort in Data Structure.

Theory:

→ Bubble Sort is simplest sorting algorithm that works by repeating, swapping the adjacent elements if they are wrong. Bubble sort sometimes referred to as sinking sort is simple sorting 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.

Worst complexity = n^2.

Average complexity = n^2

Best complexity = n.

The algorithm which is a comparision sort, is named for the way smaller or larger elements "bubble" to the top of the list.

This simple algorithm performs poorly in real world use and is used primarily as an educational tool. More efficient algorithm as timesort or merge sort are used by the sorting libraries built into popular programming language suchas Python and Java.

```
print( "Pushpraj")
a=[10,11,12,16,14,15]
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)
```

Output

Pushpraj

[10,11,12,16,14,15]

[10,11,12,14,15,16]

Working:

5	1	12	-5	16	unsorted
5	1	12	-5	16	5 > 1, swap
5	1	12	-5	16	5 < 12, ok
1	5	12	-5	16	12 > -5, swap
1	5	12	-5	16	12 < 16, OK
1	5	-5	12	16	
1	5	-5	12	16	1 < 5, OK
1	5	-5	12	16	5 > -5, swap
1	-5	5	12	16	5 < 12, OK
1	-5	5	12	16	1 > -5, swap
-5	1	5	12	16	1 < 5, OK
-5	1	5	12	16	5 < 1, OK,
-5	1	5	12	16	SORTED.

Practical:10

Aim: To demonstrate working of Selection Sort in Dat a Structures.

Theory:

Selection Sort is a simple sorting algorithm. This sorting algorithm is an in-place comparision -based algorithm in which the list is divided into two parts, the sorted part at the left end and unsorted part at the right end. It has an $O(n^2)$ time complexity, which makes it inefficient on large lists and generally performs worse than the similar insertion sort.

Worst Complexity: n^2

Average Complexity: n^2

Best Complexity: n^2

Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The time efficiency of selection sort is quadratic, so there are a number of sorting techniques which have better time complexity than selection sort.

```
print( "Pushpraj")
a=[10,11,12,16,14,15]
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)
```

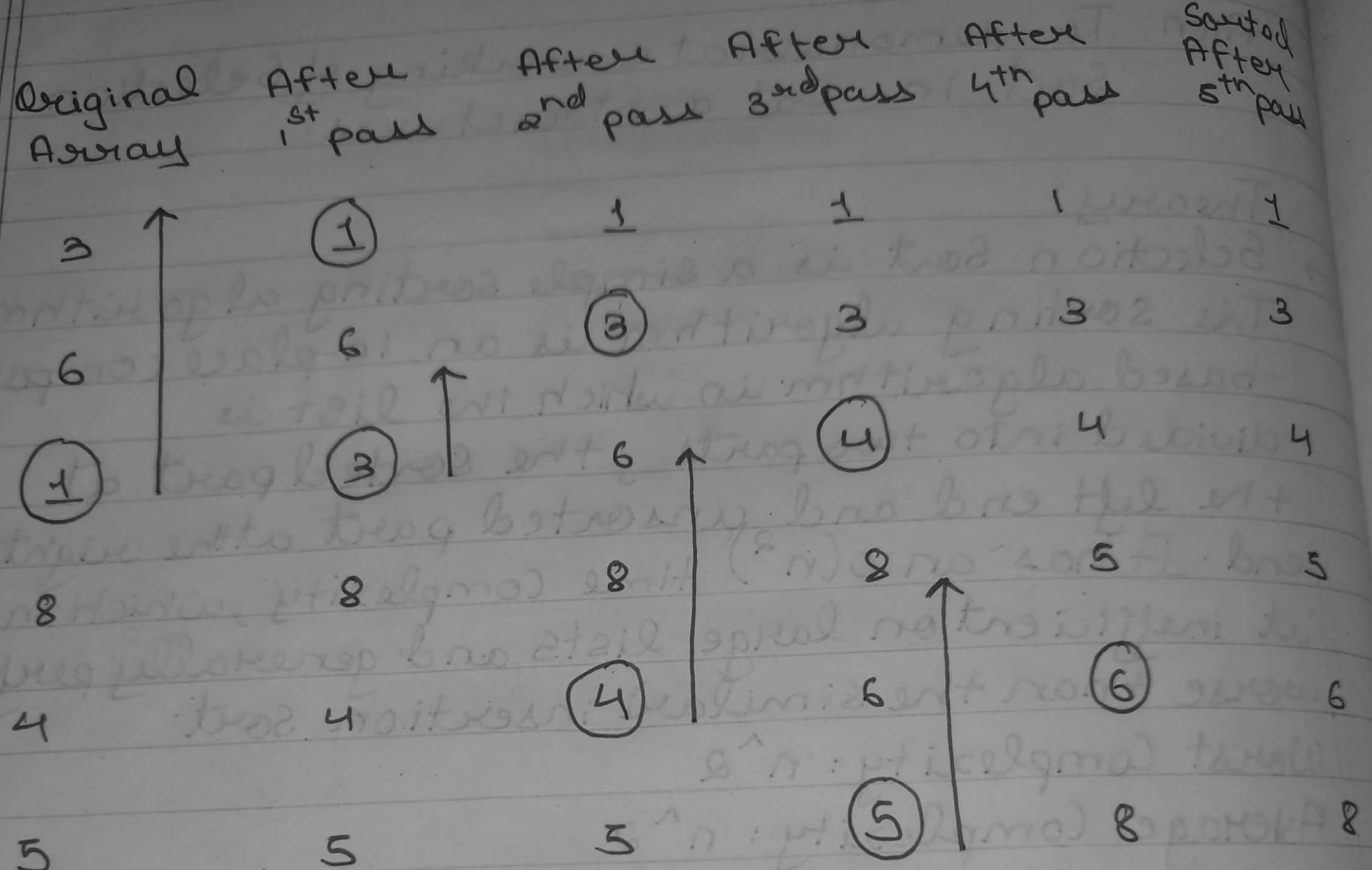
Output

Pushpraj

[10,11,12,16,14,15]

[10,11,12,14,15,16]

Working:



Practical: 11

Aim: To demonstrate the use of Quick sort in Data Structure.

Theory:-

→ Quicksort is an efficient sorting algorithm Developed by British Computer Scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors Merge Sort and Heapsort.

Worst Complexity: n^2

Average Complexity: $n * \log(n)$

Best Complexity: $n * \log(n)$

Method:- Partioning.

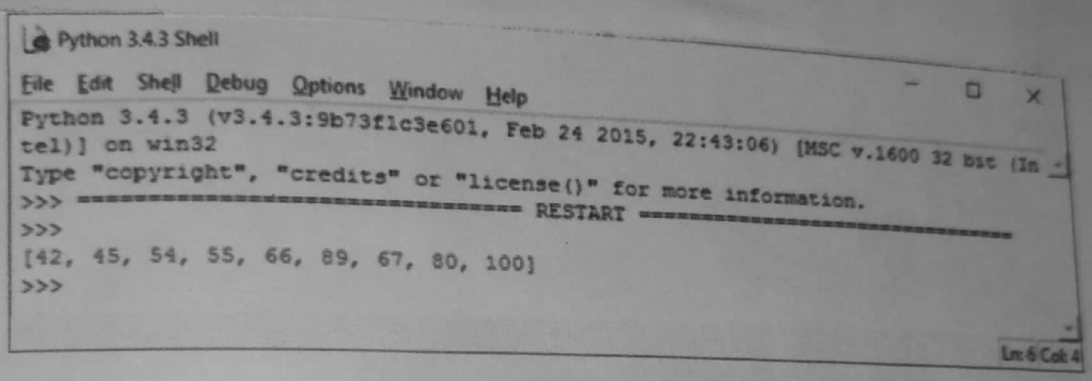
Quicksort is a divide and conquer algorithm. It works by selecting a 'pivot' element from the array and partioning the other element into two sub-array according to whether they are less than or greater than the pivot.

The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the Sorting.

```
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)
```

Working:

(a)	2	8	4	1	3	5	6	Pivot 4
(b)	2	1	Pivot 3	4	7	5	6	Pivot 8
(c)	2	Pivot 1	3	4	7	5	Pivot 6	8
(d)	1	2	3	4	Pivot 5	6	7	8
(e)	1	2	3	4	5	6	7	8



Practical:12

Aim: To sort a list using merge sort

Theory:

→ Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.
The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two-sorted sub-arrays into one.
The array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes come in to action and starts merging array back till the complete array is merged.

Applications:

→ Merge Sort is useful for sorting linked lists in O(nlogn).
→ Merge Sort accesses data sequentially and the need of random access is low.
→ Inversion count Problem.
→ Used in External sorting.
→ Merge Sort is more efficient.

```
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)
print("PUSHPRAJ SINGH 1774")
```

```
[12, 23, 34, 56, 78, 45, 86, 98, 42]
[12, 23, 56, 56, 42, 45, 78, 86, 98]
PUSHPRAJ SINGH 1774
>>>
```

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 process of visiting every node of the tree exactly once.

Border: 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 in turn might have left and right subtree.

<u>In order</u>:

→ Traverse the left subtree. The left subtree in turn might have left and right subtree.
→ Visit the root node.

```
## Binary Tree and Traversal ##

print("PUSHPRAJ SINGH,1774")

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)
```

(iii) Traverse the right subtree. The right subtrees intern might have left and right subtrees.

Postorder:

→ Traverse the left subtree.
→ The left subtrees intern might have left and right subtrees.
→ Traverse the rightsubtree. The right subtrees intern might have left and right subtrees.
→ Visit the root node.

```
    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)
T.add(78)
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)
```

```
PUSHPRAJ SINGH,1774
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
15 added on left of 60
12 added on left of 15
preorder
100
80
70
10
60
15
12
78
85
88
inorder
10
12
15
60
70
78
80
85
88
100
postorder
10
12
15
60
70
78
80
85
88
100
>>>
```

THAKUR COLLEGE OF SCIENCE & COMMERCE

NAAC Accredited with Grade "A" (3rd Cycle)

THAKUR® TRUSTS

ISO 9001 : 2015 Certified

Degree College

Computer Journal

CERTIFICATE

SEMESTER II **UID No.** _____

Class FYBSC.CS **Roll No.** 1774 **Year** 2019-20

This is to certify that the work entered in this journal is the work of Mst. / ~~Ms.~~ Pushpraj Singh

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

Teacher In-Charge

Head of Department

Date :_____

Examiner

PERFORMANCE

Term	Remarks	Staff Member's Signature
I		
II		