# PERFORMANCE

| Term | Remarks | Staff Member's Signature |
|---|---|---|
| I | | |
| II | | |

Exam Seat No. _____

# THAKUR COLLEGE OF SCIENCE & COMMERCE

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

THAKUR TRUSTS

ISO 9001 : 2015 Certified

## Degree College

# Computer Journal

# CERTIFICATE

**SEMESTER** II **UID No.** _____

**Class** F.Y.B.Sc **Roll No.** 1755 **Year** 2019-20

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

who has worked for the year 2019-20 in the Computer Laboratory.

Teacher In-Charge

Head of Department

Date : _____

Examiner

# ☆☆ INDEX ☆☆

| No. | Title | Page No. | Date | Staff Member's Signature |
|---|---|---|---|---|
| 1 | To search a number from the list using linear unsorted. | 33 | 21/12/19 | |
| 2 | To search a number from the list using linear sorted method | 35 | 21/12/19 | |
| 3 | To search a number from the given sorted list using binary search | 37 | 21/12/19 | |
| 4 | To demonstrate the use of stack | 38 | | |
| 5 | To demonstrate Queue add and delete | 40 | | |
| 6 | To demonstrate the use of circular queue in data structure. Topics: 9 | 42 | 23/01/20 | |
| 7 | To demonstrate the use of Linked list in data structure | 44 | | |
| 8. | To evaluate postfix expression using stacks | | | |

# ☆☆ INDEX ☆☆

# PRACTICAL - 1

Aim: To search a number from the list using linear unsorted.

Theory:

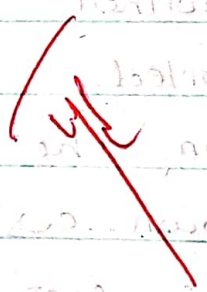
Searching: The process of identifying or finding a particular record is called searching.
There are two types of search
i) Linear Seach and ii) Binary Search
The Linear search is further classified as
a) sorted and b) unsorted
Here we will look on the UNSORTED linear search.
Linear search also known 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. i.e. what it calls unsorted linear search.

UNSORTED LINEAR SEARCH: a) The data is entered in random manner.

b) User needs to specify the element to be searched in the entered list.

c) Check the condition that whether the entered number matches if it matches then display the location plus increment 1 as data is sorted from location zero.

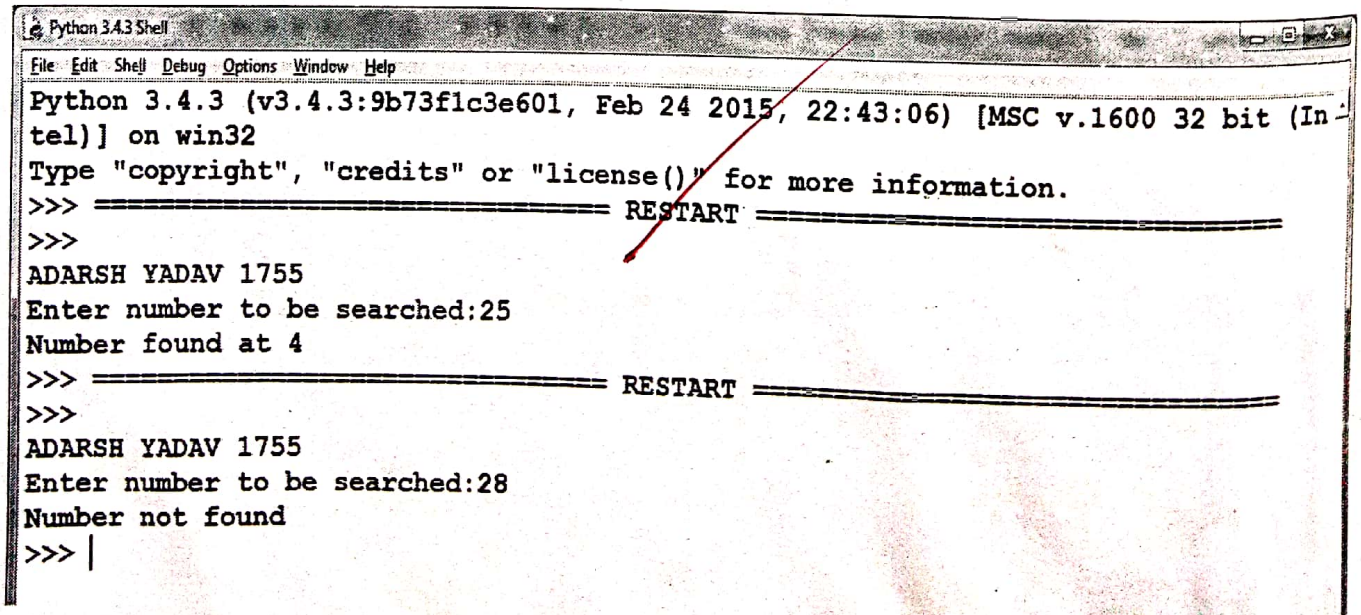
d) If all elements are checked one by one and element not found then prompt message number not found.



PROGRAM :

```
print("ADARSH YADAV 1755")
found=False
A=[10,5,16,25,35,18,14,13]
search=int(input("Enter number to be searched:"))
for i in range(len(A)-1):
   if (search==A[i]):
      print("Number found at",i)
      found=True
      break;
if(found==False):
      print("Number not found")
```

OUTPUT:



# PRACTICAL – 03

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 is meant by searching through 'sorted' that is will arranged data.

SORTED LINEAR SEARCH:

a) The user is supposed to enter data in sorted manner.

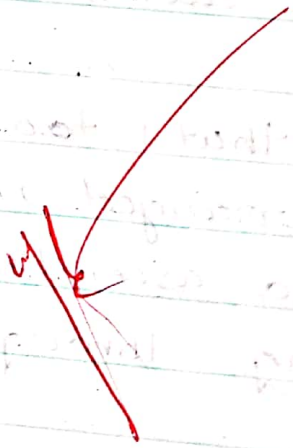
b) User has given an element for searching through sorted list.

c) If element is found display with an updation as value is sorted from location '0'.

ANS

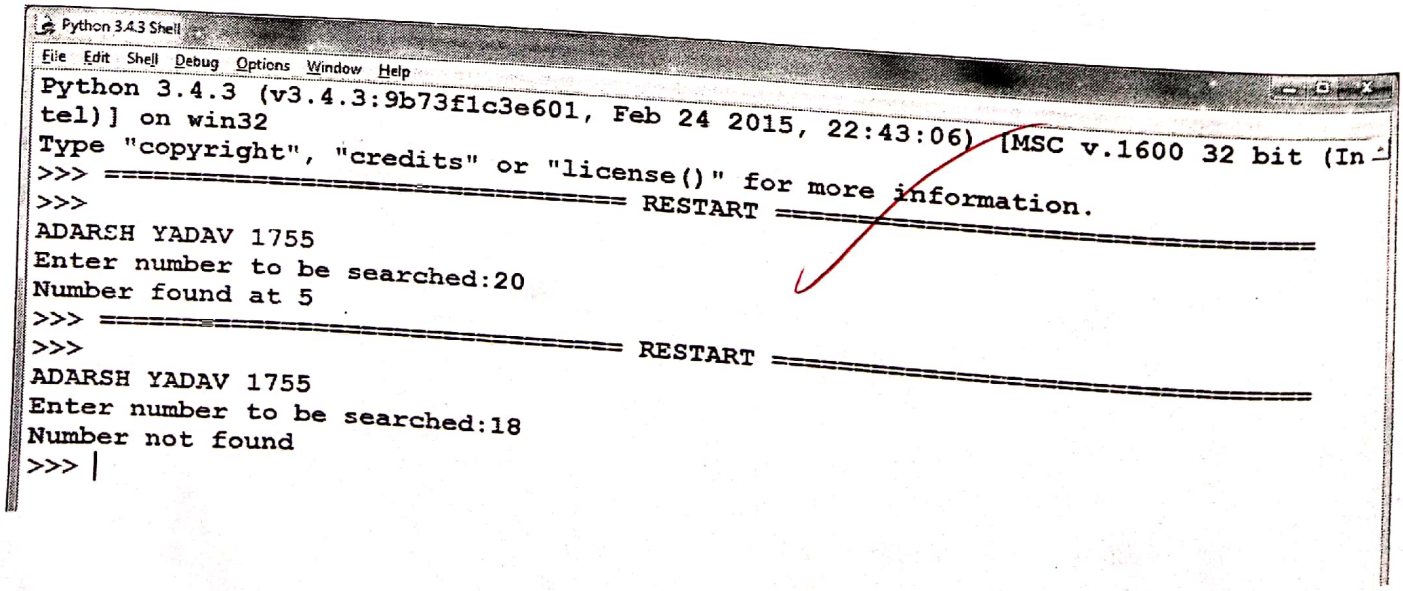
d) If data or element not found print the same.

e) In sorted ~~list~~ 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.



```
print("ADARSH YADAV 1755")
found=False
A=[7,8,10,15,20,100,110]
search=int(input("Enter number to be searched:"))
if ((search<A[0]) or (search>A[len(A)-1])):
  print("Number does not exist")
else:
  for i in range(len(A)-1):
    if (search==A[i]):
      print("Number found at",i)
      found=True
      break;
  if(found==False):
      print("Number not found")
```

OUTPUT:



# PRACTICAL-3

Aim : To search a number from the given sorted list using binary search.

Theory :

Binary Search : a) 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.

b) At each step of the algorithm a comparison is made and the procedure branches into one of two directions.

c) Specifically, the key value is compared to the middle element of the array.

d) If the key value is less than or greater than this middle element, the algorithm known which half of the array to continue searching in because the array is sorted.

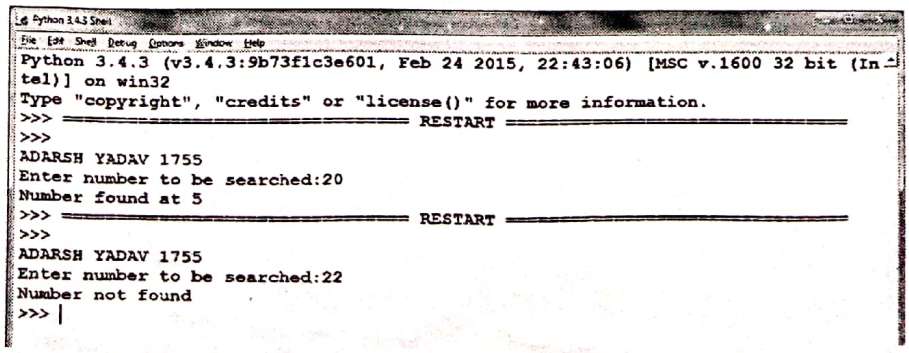
e) The process is repeated on progressively smaller segments of the array until the value is located.

f) Because each step in the algorithm divides the array size in half a binary search will complete successfully in logarithmic time.

PROGRAM:

```
print("ADARSH YADAV 1755")
A=[7,8,10,15,20,100,110]
search=int(input("Enter number to be searched:"))
l=0
r=len(A)-1
while True:
  m=int((l+r)/2)
  if(l>r):
    print("Number not found")
    break;
  if (search==A[m]):
    print("Number found at",m+1)
    found=True
    break;
  else:
    if (search<A[m]):
      r=m-1
    else:
      if (search>A[m]):
        l=m+1
```

OUTPUT:



# PRACTICAL-04

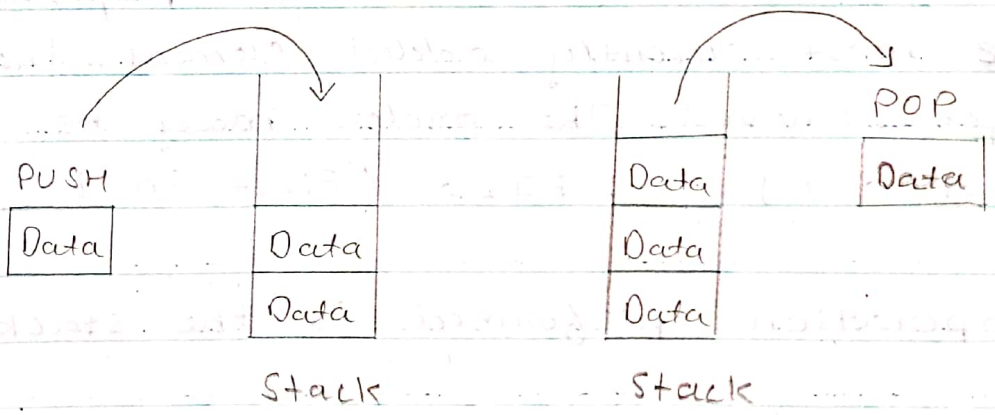
Aim: To demonstrate the use of stack.

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).

The basic operation performed in the stack are:

- PUSH: Adds an item in the stack if the stack is full then it is said to be over flow condition.
- POP: Removes an item from the stack. The items are popped in the reverse 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.
- is Empty : Returns true if stack is empty else false.



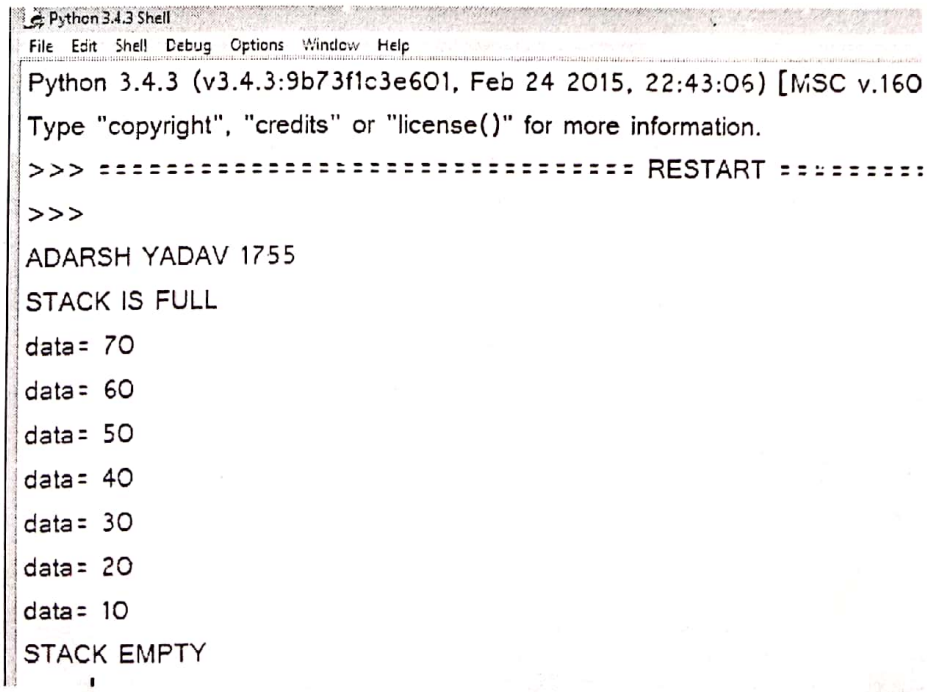

Last - in - First - Out

PROGRAM:

```
##stack##
print("ADARSH YADAV 1755")
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:



# 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 begining of the queue and Rear points to the end of a queue.

Queue follows the FIFO (First_in_First_out) structure.

According to its FIFO structure element inserted first will also be removed first.

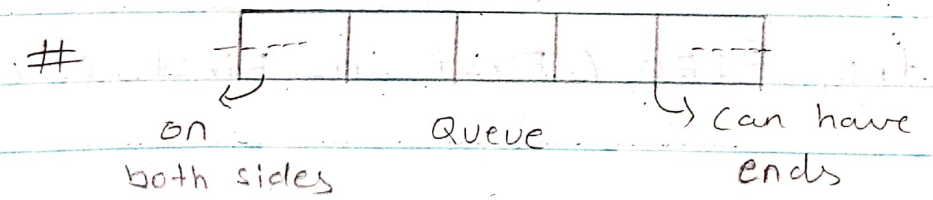
In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue) because queue is open at both of its ends.

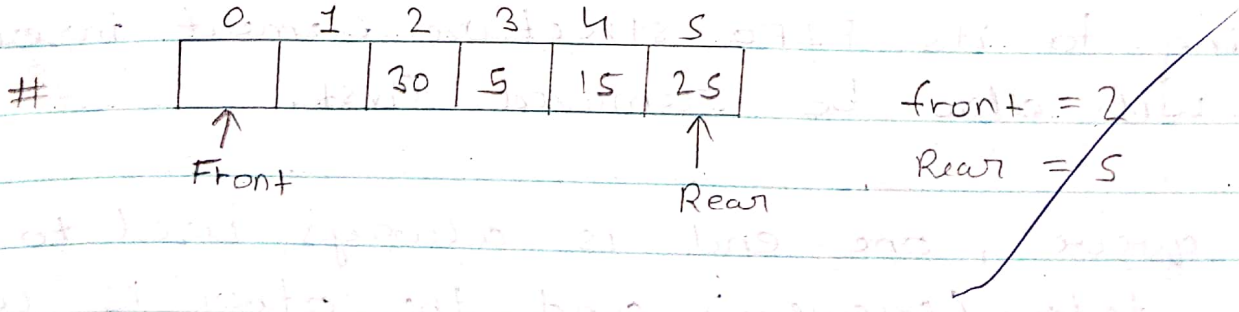
enqueue () can be termed as add() in queue i.e. adding a element in queue.

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.



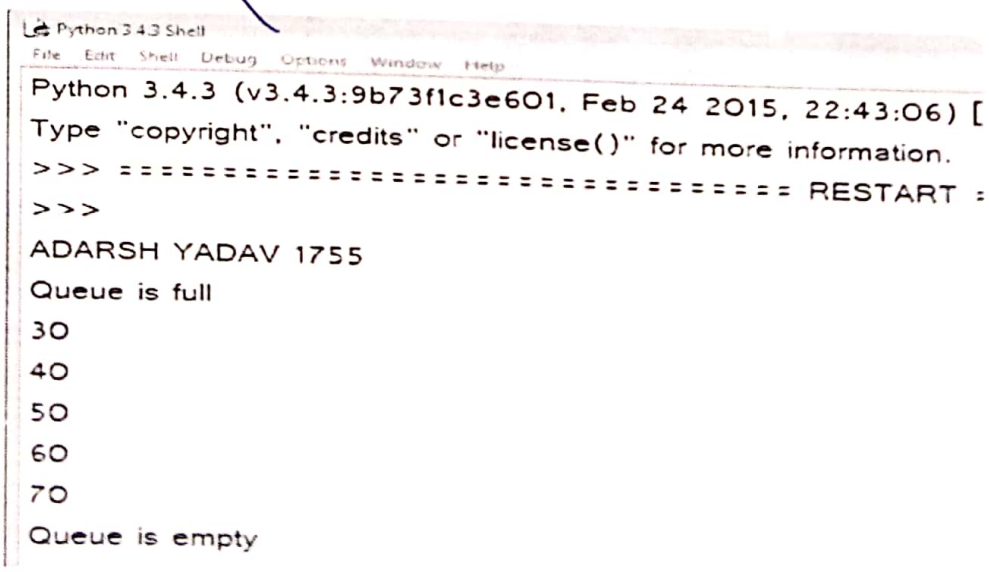



PROGRAM:

```
## Queue add and Delete ##
class Queue:
   print("ADARSH YADAV 1755")
   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()
```

OUTPUT:



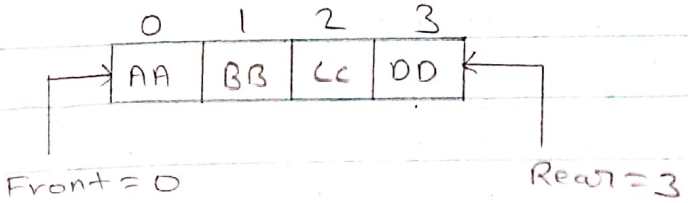
# PRACTICAL - 6

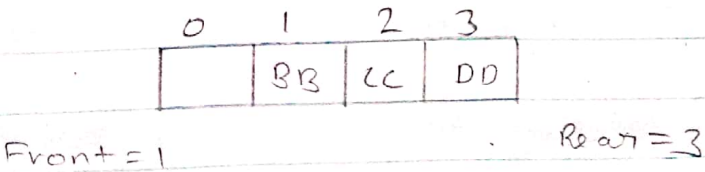
Aim: To demonstrate the use of circular queue in data structure.

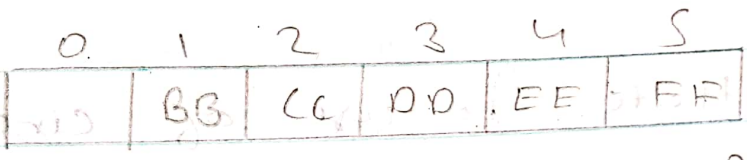
Theory: The queue that we implement using an array from one limitation. In that implementation there is a possibility that the queue is reported as full, even though in actuality there might be empty slots at the beginnings of the queue.

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:

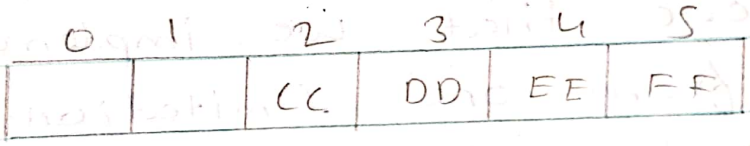




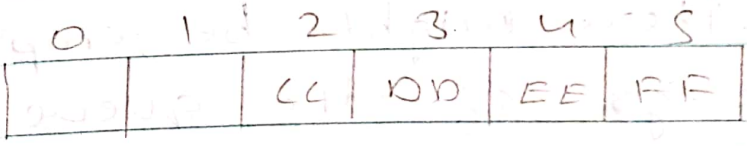

Front = 1

Rear = 5




Front = 2

Rear = 5




Front = 2

Rear = 0

PROGRAM:

```
#(circular queue)
print("ADARSH YADAV 1755")
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 se!f.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)
```

OUTPUT:

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART :
>>>
ADARSH YADAV 1755
data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
data removed: 44
```

# PRACTICAL – 07

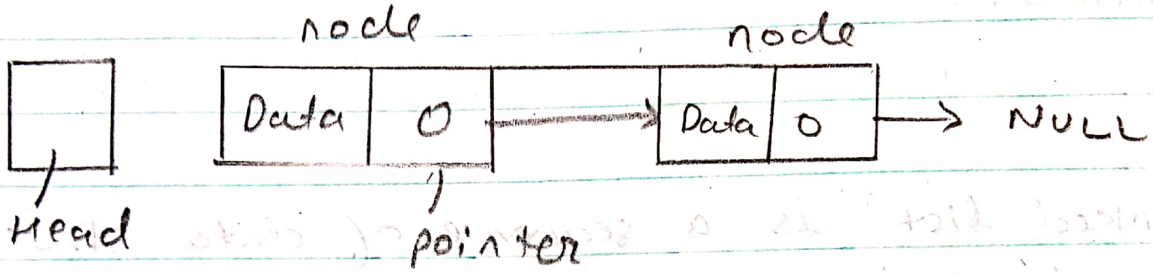
Aim: To demonstrate the use of Linked list in data structure.

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 NE
- 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 :

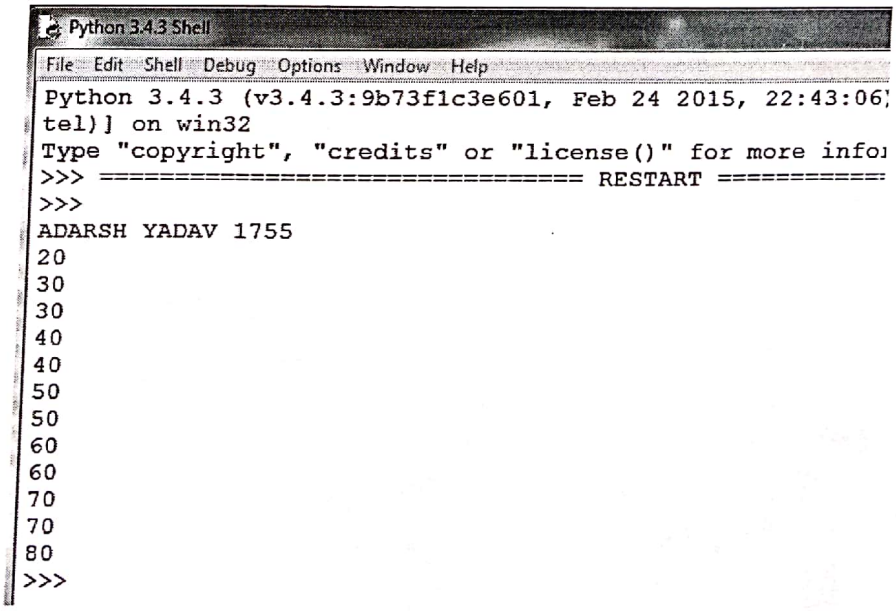
1) Insertion
2) Deletion
3) Display
4) search
5) Delete

PROGRAM:

```
### After Before linkedlist(simple)###
print("ADARSH YADAV 1755")
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
whilehead.next!=None:
print(head.data)
head=head.next
print(head.data)
start=linkedlist()
start.addL(50)
start.addL(60)
start.addL(70)
start.addL(80)
start.addB(40)
start.addB(30)
start.addB(20)
start.display()
```

OUTPUT:



# PRACTICAL -08

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.

Algorithm:

Step 1: Read all the symbols one by one from left to right in the given postfix expression.

Step 2: If the reading symbol is operand then push it on to the stack.

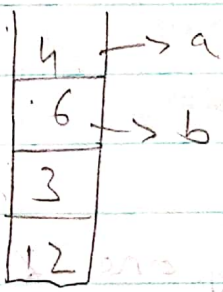
Step 3: If the reading symbol is operator ( +, -, *, /, etc. ) the perform two pop operation.

Step 4: Finally, perform a pop operation and display the popped value as final result.

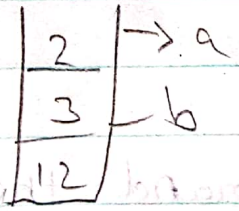
Value of postfix expression;

S = 1 2 3 6 4 - + *

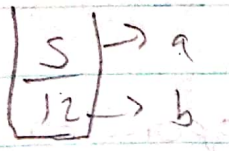
Stack




b - a = 6 - 4 = 2 //store again in stack




b + a = 3 + 2 = 5



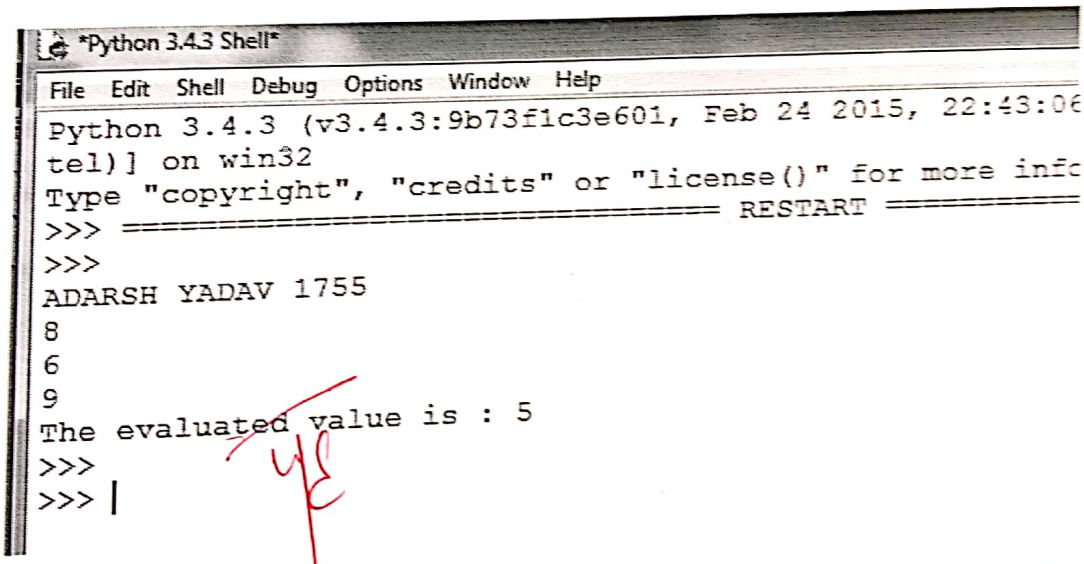

b * a = 12 * 5 = 60

PROGRAM:

```
##Postfix Evaluation##
print("ADARSH YADAV 1755")
def evaluate(s):
    k=s.split()
    n=len(k)
stack=[]
for i in range(n):
if k[i].isdigit():
print(int(k[i]))
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.apend(int(b)/int(a))
returnstack.pop()
s="8 6 9 - +"
r=evaluate(s)
print("The evaluated value is :",r)
```

OUTPUT:



# PRACTICAL - 09

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 to 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 order.

The pass through the list is repeated until the list is sorted.

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

Although the algorithm is simple it is too slow 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 5 > 2

(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 since 4 > 2

(1 2 4 5 8) → (1 2 4 5 8)

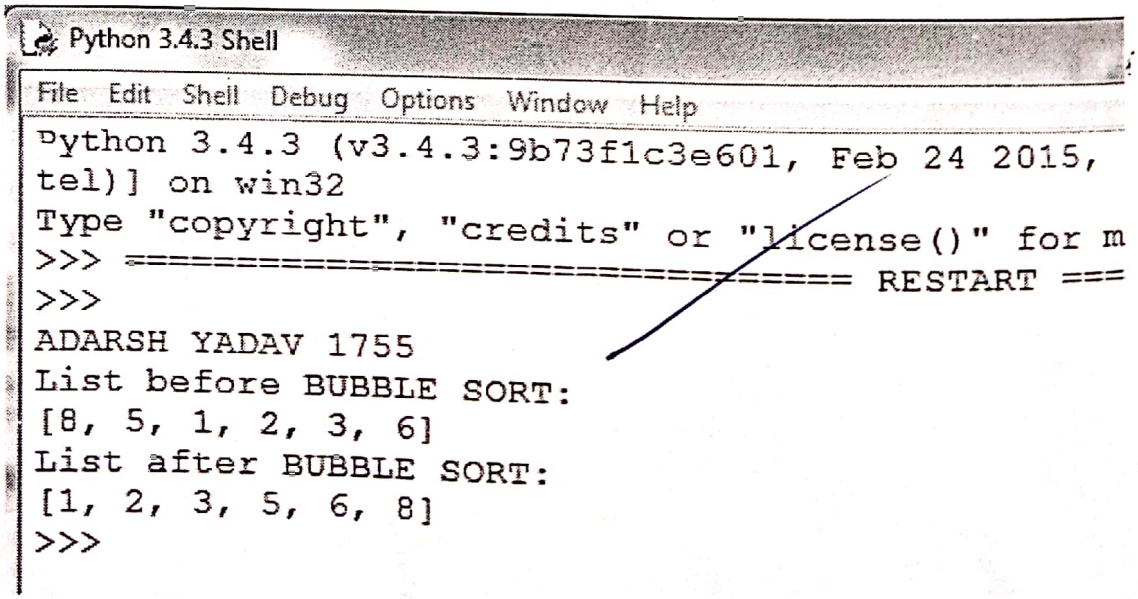
Third pass

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

PROGRAM:

```
## BUBBLE SORT ##
print("ADARSH YADAV 1755")
print("List before BUBBLE SORT:")
A=[8,5,1,2,3,6]
print(A)
for i in range (len(A)-1):
    for j in range (len(A)-1-i):
        if(A[j]>A[j+1]):
            t=A[j]
            A[j]=A[j+1]
            A[j+1]=t
print("List after BUBBLE SORT:")
print(A)
```

OUTPUT:



# PRACTICAL – 10

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

Theory:

QUICK SORT: Quick Sort is an efficient sorting algorithm type of a divide and 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.

080

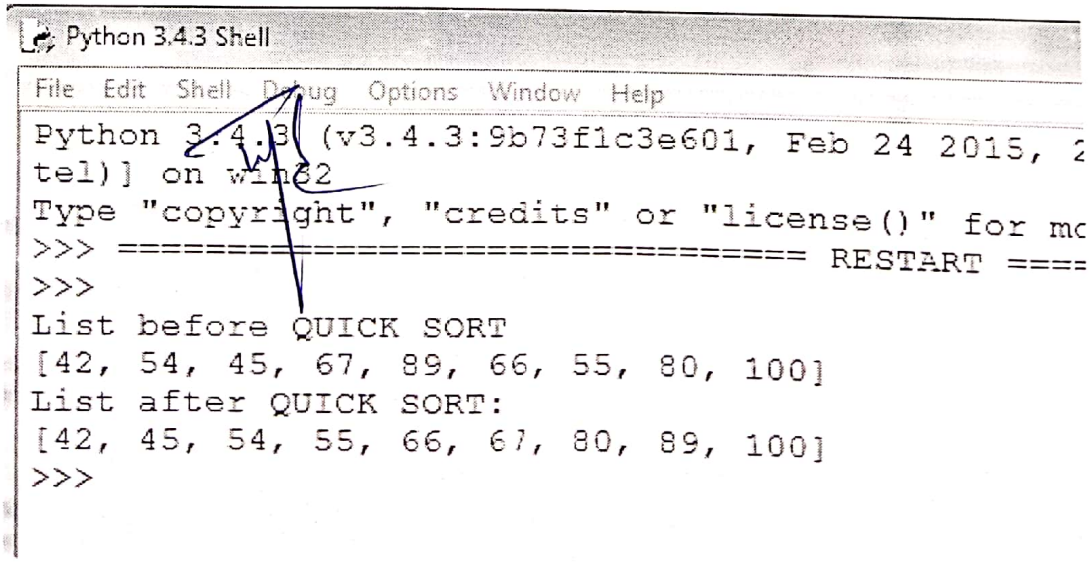
The key process in quicksort is partition(). Target of partition is given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in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

PROGRAM:

```
## QUICK SORT ##
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)
def quickSortHelper(alist,first,last):
   if first<last:
      splitpoint=partition(alist,first,last)
      quickSortHelper(alist,first,splitpoint-1)
      quickSortHelper(alist,splitpoint+1,last)
def partition(alist,first,last):
   pivotvalue=alist[first]
   leftmark=first+1
   rightmark=last
   done=False
   while not done:
      while leftmark<=rightmark and alist[leftmark]<=pivotvalue:
         leftmark=leftmark+1
      while alist[rightmark]>=pivotvalue and rightmark>=leftmark:
         rightmark=rightmark-1
      if rightmark<leftmark:
         done=True
      else:
         temp=alist[leftmark]
         alist[leftmark]=alist[rightmark]
         alist[rightmark]=temp
```

```
    temp=alist[first]
    alist[first]=alist[rightmark]
    alist[rightmark]=temp
    return rightmark
alist=[42,54,45,67,89,66,55,80,100]
print("List before QUICK SORT")
print(alist)
quickSort(alist)
print("List after QUICK SORT:")
print(alist)
```

OUTPUT:



# PRACTICAL – II

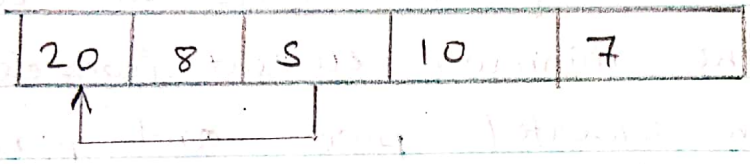
Aim : To sort given random data by using Selection sort.

Theory : The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the begining. The algorithm maintains two sub arrays in a given array. The subarray which is already sorted.

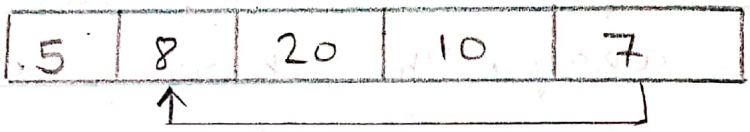
The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the smallest value as it makes a pass and after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the smallest item is in the correct place.

After the second pass, the next smallest is in place. This process continues and requires n-1 passes to sort n items, since the final item must be place after the (n-1) st pass.

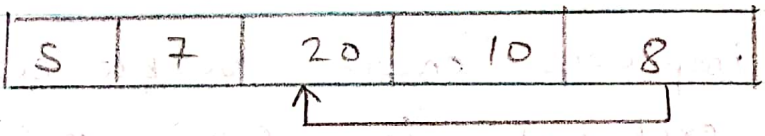
e.g. On each pass the smallest remaining item is selected and then placed in its proper location.




5 is smallest

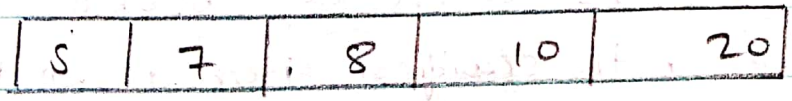



7 is smallest




8 is smallest

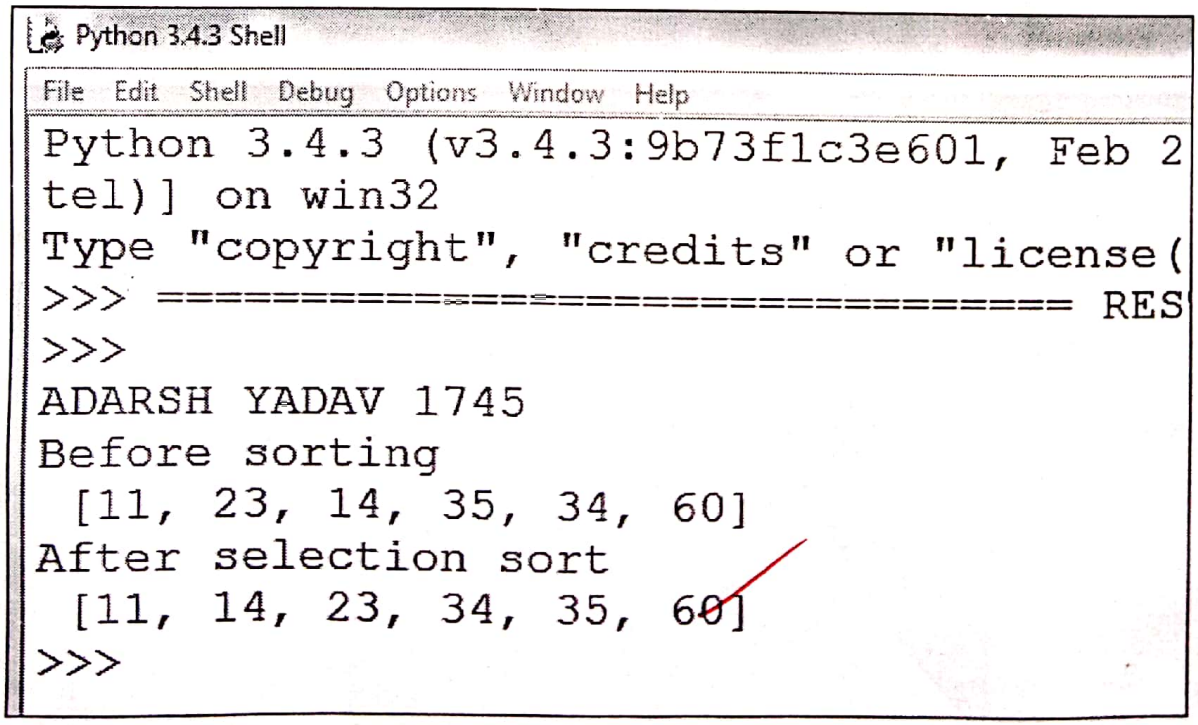

10 ok list is sorted




20 ok list is sorted

PROGRAM:

```
## SELECTION SORT ##

print("ADARSH YADAV 1745")

a=[23,22,18,96,56,60]

print("Before sorting \n",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("After selection sort \n",a)OUTPUT:
```

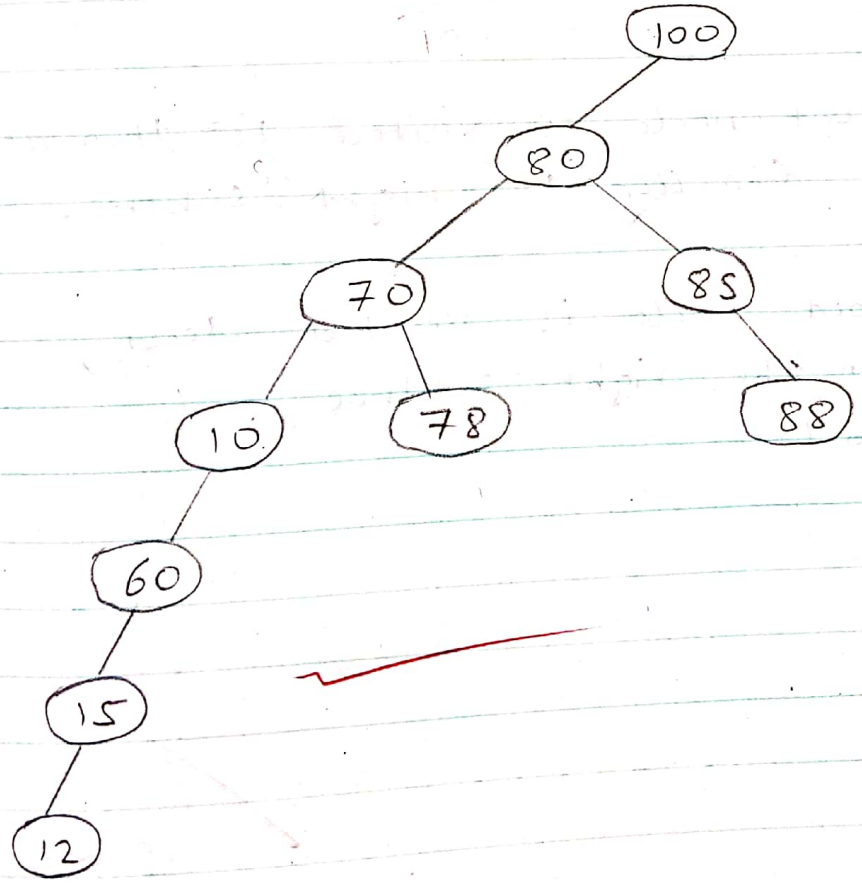


# PRACTICAL – 12

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 data structure in which a node can have at most two children.




# Diagramatic representation of BINARY SEARCH TREE

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

1) IN-ORDER : The left-subtree is visited 1st then the root and later the right sub tree, we should always remember that every node may represent a subtree itself.
Output produced is sorted key values in ASCENDING ORDER.

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

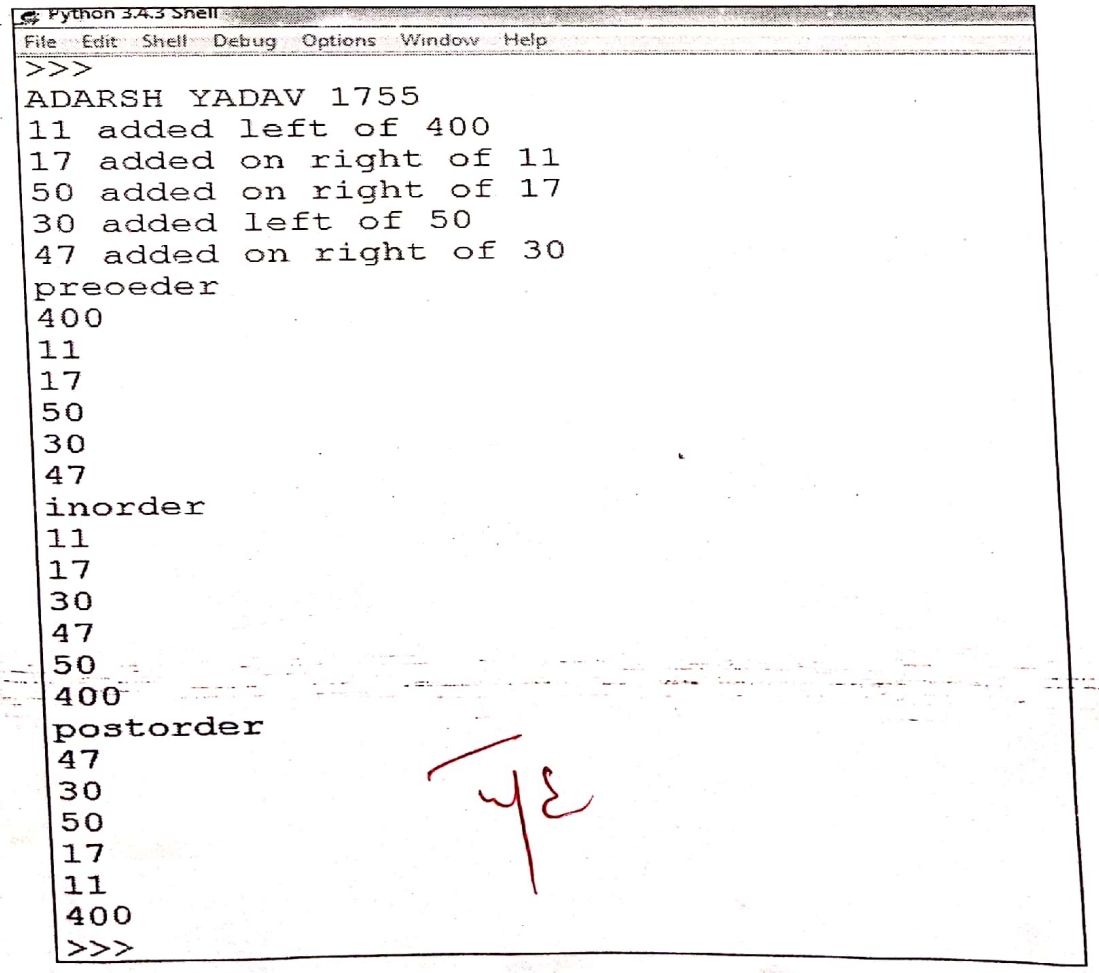
3) POST-ORDER : The root node is visited last, left subtree, then the right subtree and finally root node.

PROGRAM:

```
## BINARY SEARCH TREE ##
class Node:
print("ADARSH YADAV 1755")
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):
ifself.root==None:
self.root =Node(val)
else:
newnode=Node(val)
        h=self.root
while True:
ifnewnode.data<h.data:
ifh.l!=None:
                h=h.l
else:
h.l=newnode
print(newnode.data,"added left of",h.data)
break
else:
ifh.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)
definorder(self,start):
if start!=None:
self.inorder(start.l)
print(start.data)
self.inorder(start.r)
defpostorder(self,start):
if start!=None:
self.postorder(start.l)
self.postorder(start.r)
print(start.data)
T=Tree()
T.add(400)
T.add(11)
```

```
T.add(17)
T.add(50)
T.add(30)
T.add(47)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
```

OUTPUT:



# PRACTICAL - 13

AIM: MERGE SORT

Theory: Merge sort is a sorting technique based on divide and conquer technique with worst-case time complexity being O(nlogn), is 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 [1 ... m] and arr [m+1 .... r] are sorted and merges the two sorted sub-array into one.

PROGRAM :

```
print("ADARSH YADAV 1755")
def sort(arr,l,m,r):
    n1=m-l+1
    n2=r-m
    L=[0]*(n1)
    R=[0]*(n2)
    for i in range(0,n1):
        L[i]=arr[l+i]
    for j in range(0,n2):
        R[j]=arr[m+1+j]
    i=0
    j=0
    k=l
    while i<n1 and j<n2:
        if L[i]<=R[j]:
            arr[k]=L[i]
            i+=1
        else:
            arr[k]=R[i]
            j+=1
            k+=1
        while i<n1:
            arr[k]=L[i]
            i+=1
            k+=1
        while j<n2:
            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("array before sorting \n",arr)
n=len(arr)
mergesort(arr,0,n-1)
print("array after merge sorting \n",arr)
```

OUTPUT: