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

Theory:- The process of identify or finding a particular record is called searching.
There are two type 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 elements in the list sequential until the desired element is found.
When the element to be searched are not specifically arranged is ascending or descending order. They are arranged in random manner. that is what it calls unsorted linear search
Unsorted linear search.
→ the data linear search entered in random manner.
→ user needs to specify the elements 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 massage number not found.

```
print("PRATICAL NO 1")
print("FAIZAN SALMANI")
print("1779")
a=[18,15,36,79,41,12,64]
j=0
print(a)
search=int(input("enter the number which has to be search---->"))
for i in range(len(a)):
  if (search==a[i]):
    print("number found at----->",i+1)
    j=1
    break
if(j==0):
  print("number has been not found ")
```

output:

```
Python 3.6.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/KACHHARA/AppData/Local/Programs/Python/Python36-32/faiz.py
PRATICAL NO 1
FAIZAN SALMANI
1779
[18, 15, 36, 79, 41, 12, 64]
enter the number which has to be search---->12
number found at-----> 6
>>>
 RESTART: C:/Users/KACHHARA/AppData/Local/Programs/Python/Python36-32/faiz.py
PRATICAL NO 1
FAIZAN SALMANI
1779
[18, 15, 36, 79, 41, 12, 64]
enter the number which has to be search---->56
number has been not found
>>>
```

Practical no-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 elments and to display the same.

In searching that too in LINER SORTED Search. the data is arranged to ascending to descending order. 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 as element for searching through sorted list.

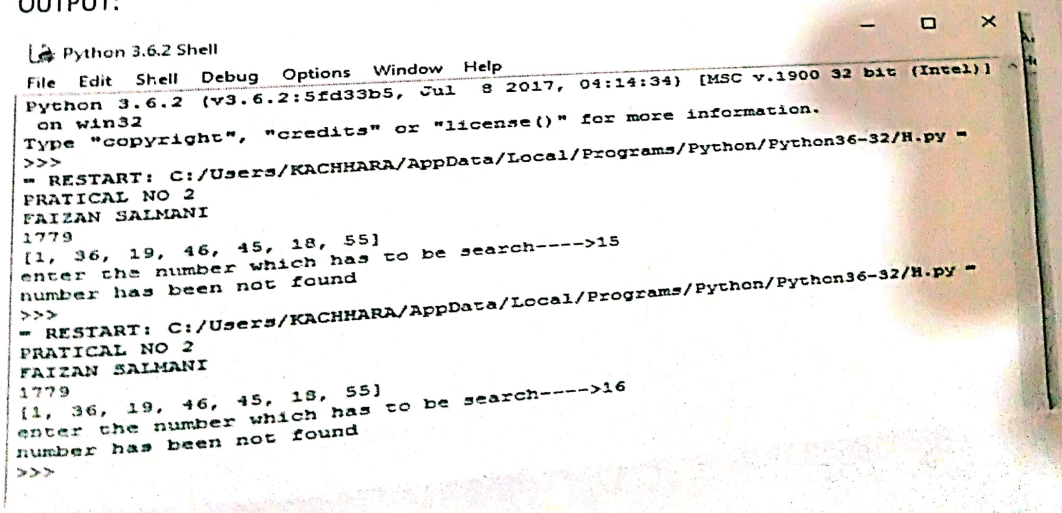
⇒ If element is founed display with an updation as value is stored from location '0'.

⇒ 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 that then without any processing will can say number not in the list.

```
print("PRATICAL NO 2")
print("FAIZAN SALMANI")
print("1779")
a=[56,45,49,18,49,79]
j=0
print(a)
search=int(input("enter the number which has to be search---->"))
if ((search<a[0]) or (search>a[6])):
  print("given number doesnt exist!")
else:
  for i in range(len(a)):
    if (search==a[i]):
      print("number found at----->",i+1)
      j=1
      break
  if(j==0):
    print("number has been not found ")
```

OUTPUT:



06

```
print("pratical no 3")
print("FAIZAN SALMANI")
print("1779")
a=[9,15,35,38,51,64]
print(a)
search=int(input("Enter number to be searched from the list:"))
l=0
h=len(a)-1
m=int((l+h)/2)  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(ll=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)
```

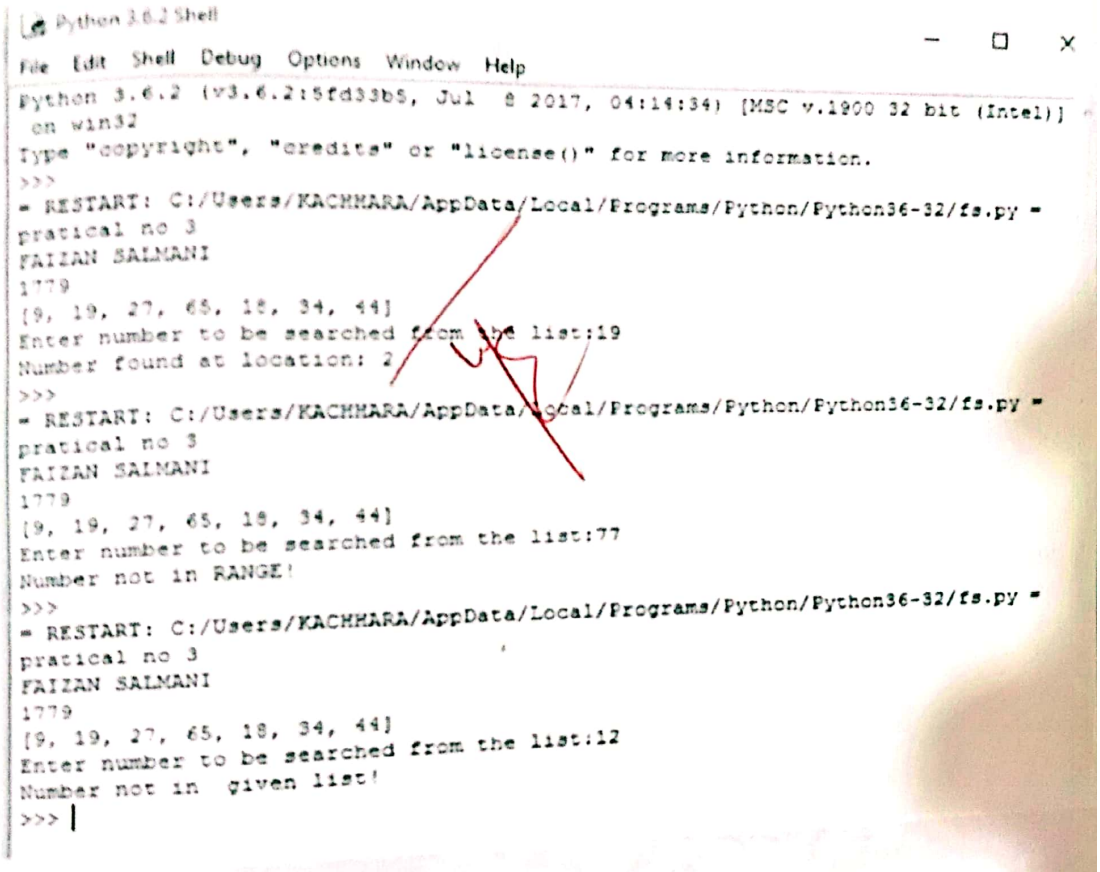
Aim:- To search a number from the given sorted list using binary search.

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 direction.
specifically the key value is compared to the middle element of array.
if the key value is less than or greater than this middle elements 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 untill the value is located

18

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

```
    if(search!=a[m]):
        print("Number not in  given list!")
break
```



practical no-4

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

Theory:- SORTING is type is in which any random data is sorted i.e arranged in ascending or descending order BUBBLE sort sometimes refered to as sinking sort

Is a simple sorting algorithm that repeatedly steps through the list compares adjacent element 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 very smaller or larger elements "bubble" to the top of the list.

Although the algorithm is simple it is also too slow as compare one elements checks if condition feels then only swaps otherwise goes on.

Example:-
first pass
(5 1 4 2 0) → (1 5 4 2 0) Here algorithm compare the first two elements and swap since 5>1
(1 5 4 2 0) → (1 4 5 2 0) swap 5>4
(1 4 5 2 0) → (1 4 2 5 0) swap 5>2
(1 4 2 5 0) → (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 0)
(1 4 2 5 0) → (1 2 4 5 0) swap sin 4>2
(1 2 4 5 0) → (1 2 4 5 8)

Third pass
(1 2 4 5 0) it checks and gives the data in sorted order.

SOURCE CODE:

```
print("faizansalmani \n1779 ")
class stack:
  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()
```

OUTPUT:

```
faizansalmani
1779
Stack is full!
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
Stack is cmpty!
```

Practical no - 5

Aim :- to demonstrate the use of stack

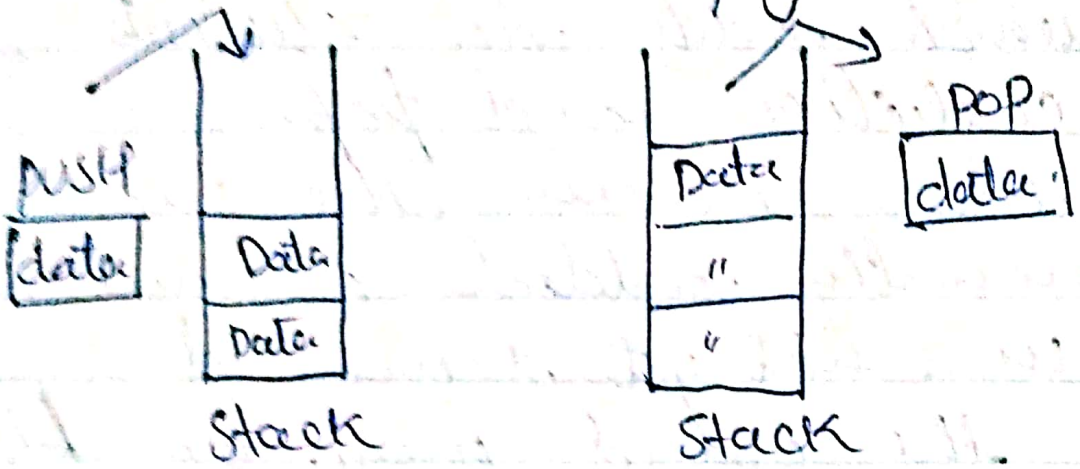
Theory :- In computer science a stack serves as collection of elements which two principle operation push which adds an elements to the collection and pop. which removes the most recently added element that was not yet removed the order may be LIFO (Last order may be LIFO (last in first out) or FILO (First in last out)

Three basic operation are performed in the stack

- PUSH : adds an items in the if the stacks is full then it is said to be an flow condition
- POP : Removes an items from the stack the items are poped in the revers 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.

isEmpty: Returns true if stack is empty else false.



Last_in_first_out

SOURCE CODE:

```
print("faizansalmani\n1779")
class queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[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:

```
faizansalmani
1779
Queue is full!
Queue is full!
30
40
50
60
Queue is Empty!
Queue is Empty!
```

Practical no-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 other end called as FRONT.

Front point to the beginning of the queue and Rear points to the end of the queue.

Queue follow the FIFO (First_in_First_out) structure

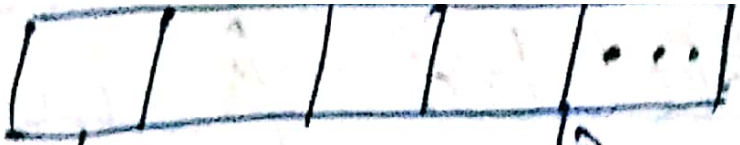
According to its FIFO structure element inserted first will also be removed first.

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 on both of its ends.

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

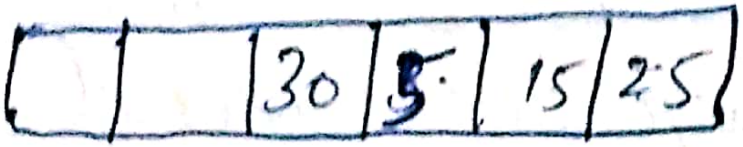
Dequeue() can be termed as delet or Remove i.e deleting or removing of elements.

#



on both sides Queue can have ~~open~~ ol

#



Front = 2

Rear = 5

SOURCE CODE:

```
print(" faizansalmani\n1779")
class q ..ue:
  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:
      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:

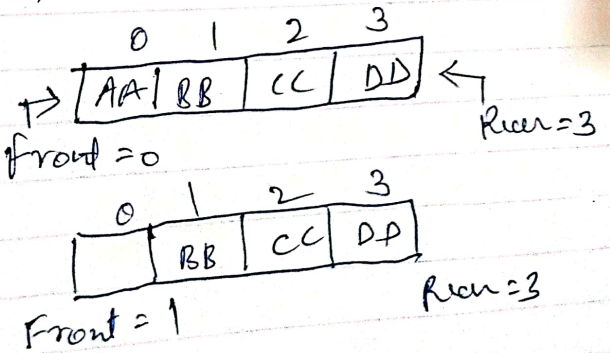
```
faizansalmani
1779
data added: 44
data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
Data removed: 44
```

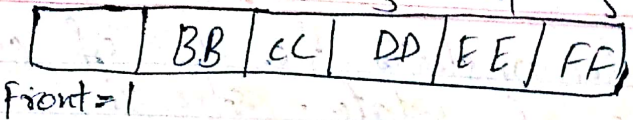
Practical no-7

Aim :- To demonstrate the use of circular queue as 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 actually 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 element to the queue and reach the end of the array. The next element is stored in the ~~first slot~~ slot of array.

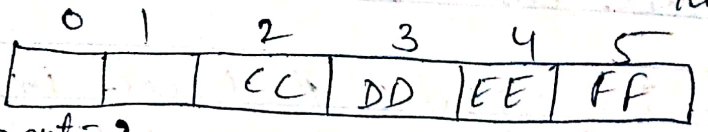
Example :-





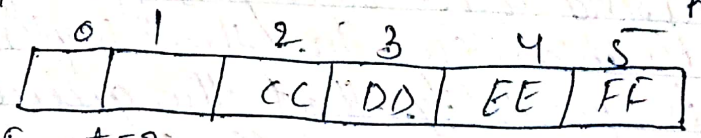
Front = 1

Rear = 5



Front = 2

Rear = 5



Front = 2

Rear = 0

```
            newnode.next=self.s
            self.s=newnode
        def display(self):
            head=self.s
            while head.nextl=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()
```

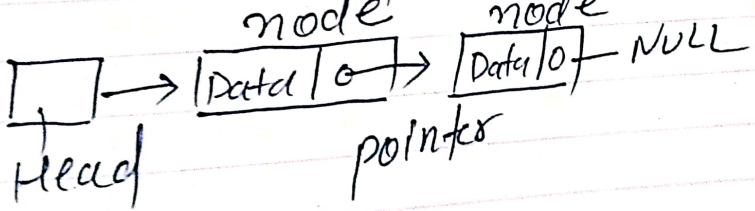
Practical no-8

Aims - To demonstrate the use of linked list

Theory :- A linked list is a sequence of data structure linked list is a sequence of links which contains items each link contain 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 contain link to the next link called next.
- LINKED LIST - A linked list contain the connection link to the first link called first.

LINKED LIST Representation

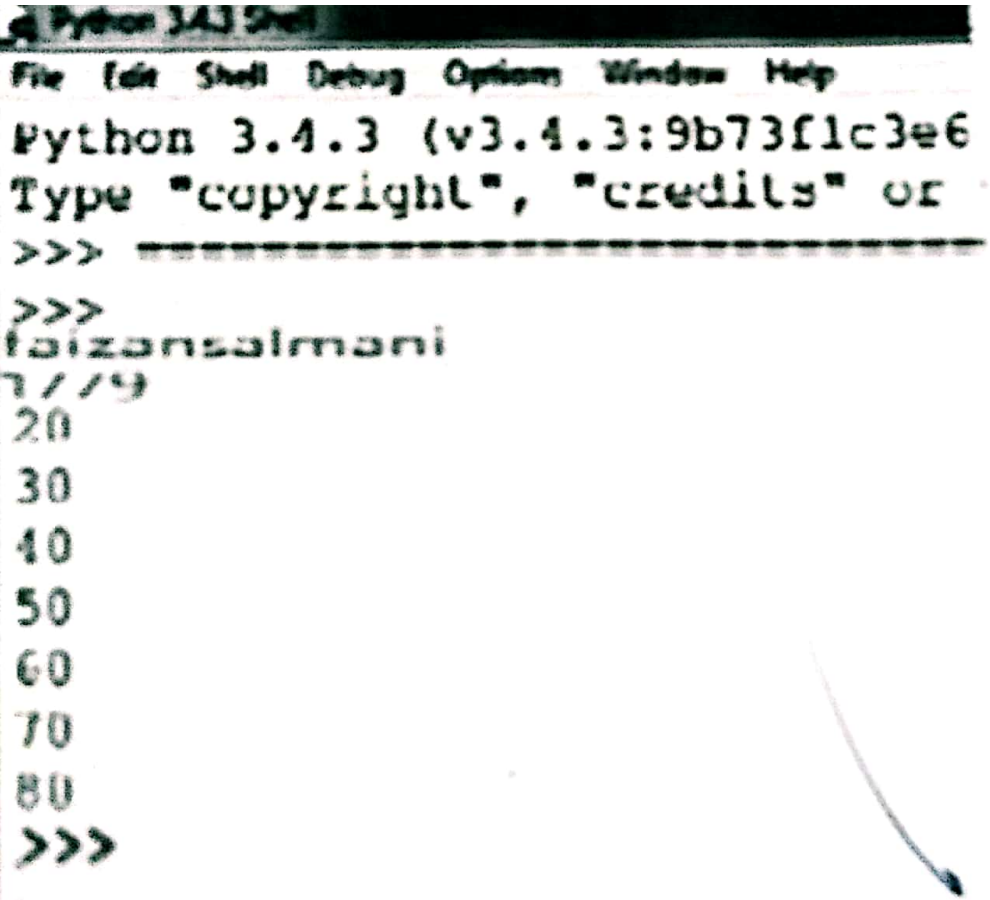


Types of linked list:

- simple
- Doubly
- circular

BASIC operations

- Insertion
- Deletion
- Display
- Search
- Delete



```
Print("faizansalmani/n1779/nPostfix Program:")
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)
```

Practical no-9

Aim:- To evaluate postfix Expression using stack.

Theory :— Stack is an (ADT) and work on LIFO (Last-in first-out) i.e PUSH & POP operation
A postfix expression is a collection of operator is placed after the operands.

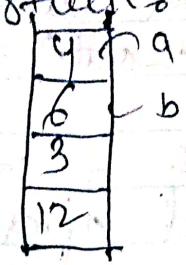
Step to the followed.

1. Read all the symbols one by one from left to right in the given postfix operation
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 operand 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.

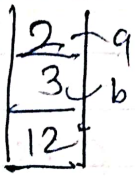
Value of postfix expression

8 = 12 3 6 4 ⊖ + *

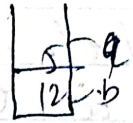
Stack:



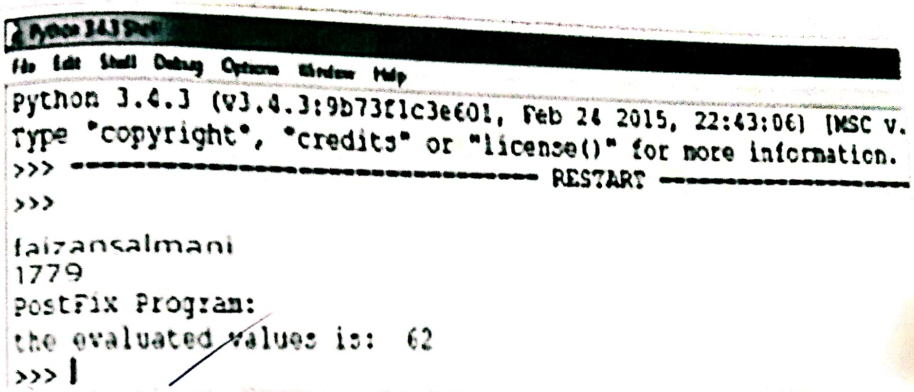
b-a = 6-4 = 2 // store again in stack.



b+a = 3+2 = 5 // store result in stack.



b * a - 12 * 5 = 60



```
print("FAIZ")
print("1779")
a=[15,23,48,19,34,76,45,94,25]
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(a)
```

OUTPUT:

```
Python 3.6.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:14:34)
 on win32
Type "copyright", "credits" or "license()" for more i
>>>
= RESTART: C:/Users/KACHHARA/AppData/Local/Programs/P
FAIZ
1779
[15, 23, 48, 19, 34, 76, 45, 94, 25]
[15, 19, 23, 25, 34, 45, 48, 76, 94]
>>>
```

Practical no 10

Bubble sort

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

Theory: Bubble Sort is the simplest Sorting algorithm that works by repeating swapping the adjacent element if they are in wrong. Bubble sort sometimes referred to as sinking sort is a simple sorting algorithm that repeatedly steps through algorithm list compares adjacent elements and swaps them if they are in the wrong order. the pass until the list is sorted

Worst complexity - n^2

Average complexity - n^2

Best complexity - n

The algorithm which is a comparison 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 primarily as an educational tool. More efficient algorithms such as timsort or merge sort are used by the

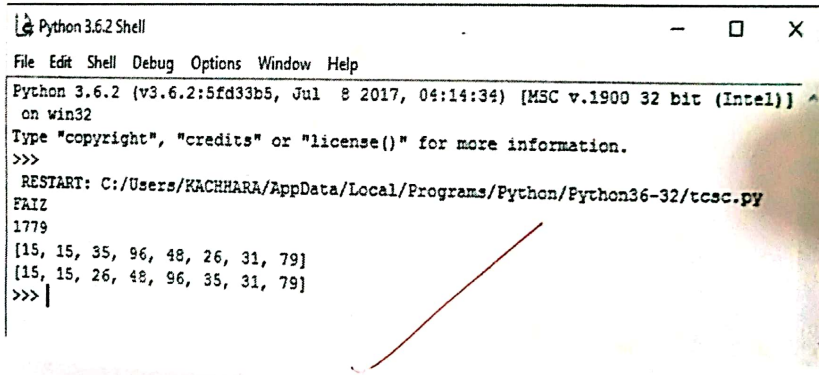
Sorting libraries built into popular programming languages — such as python and Java.

working

5	1	12	-5	16	unsorted
5	1	12	-5	16	5>1, swap
1	5	12	-5	16	5<12, ok
1	5	12	-5	16	12>-5, swap
1	5	-5	12	16	12<16, ok
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 & pass
-5	1	5	12	16	SORTED

```
print("FAIZ")
print("1779")
a=[15,15,35,96,48,26,31,79]
print(a)
for i in range(len(a)-1):
    for j in range(len(a)-1-i):
        if(a[j]>a[i+1]):
            t=a[j]
            a[j]=a[i+1]
            a[i+1]=t
print(a)
```

OUTPUT:



Practical no 11

Selection sort

Aim: To demonstrate working of selection sort in data structure

Theory:

Selection sort is a simple sorting algorithm this sorting algorithm is an in which the list is algorithm into divided two parts the sorts part at the left end and the unsorted part at the right end It has an $O(n^2)$ time complexity which makes it inefficients on large lists and generally performs worse than the similar insertion sort

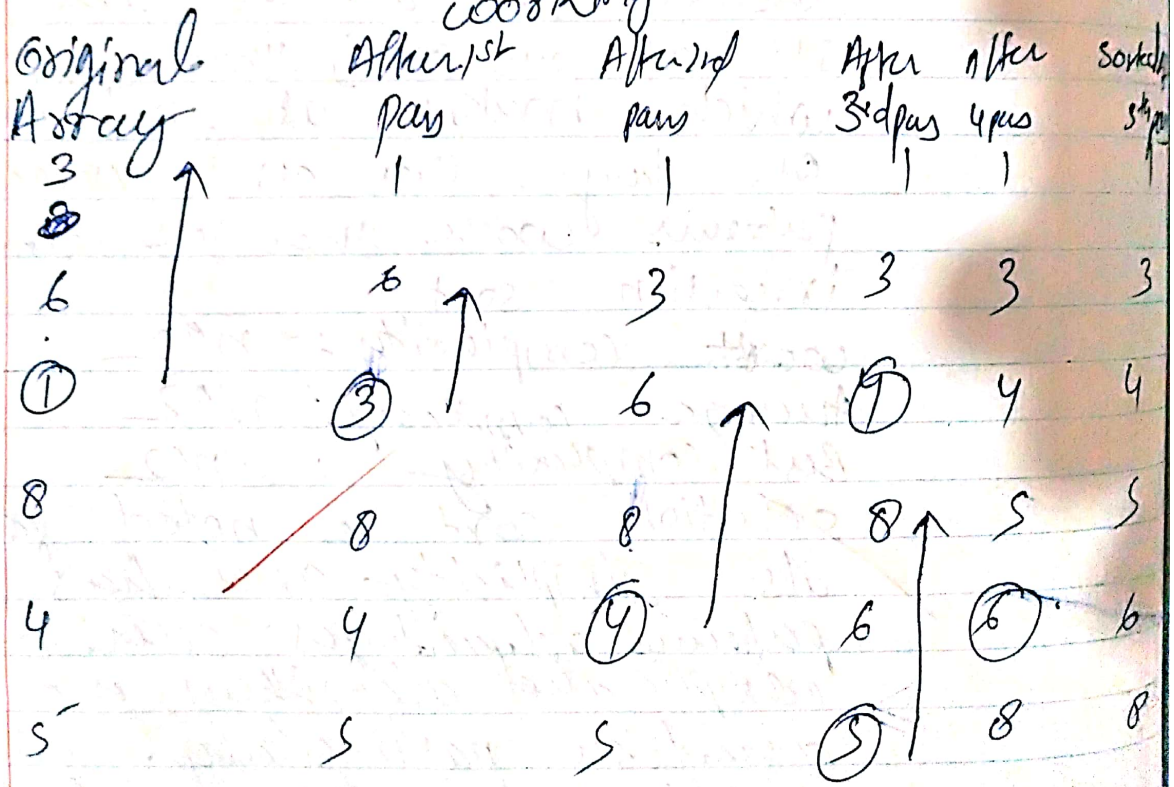
worst complexity :- n^2

Average complexity : n^2

Best complexity : n^2

selection sort is noted for its simplicity and has performs advantages over more complicated algorithem in certain situation particularly where auxiliary memory is limited

The time efficiency of selection sort is quadratic so there are a number of sorting techiques which have better time complexity than selection sort. One thing which distinguishes selection sort from other sorting algorithm is that it makes the minimum possible number of swaps n-1 in the worst case. It Compares each element of list and sort s accordingly cooking.



```
print("FAIZ")
print("1779)
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:
```

Practical no. 12
Quick Short

Aim: To demonstrate use of Quick Sort in data structure

Theory: Quicksort is an efficient sorting algorithm. Developed by British computer scientist Tony Hoare in 1959 and 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 heap sort.

Worst complexity : n^2
Average complexity : $n * \log(n)$
Best complexity : $n * \log(n)$
Method : partitioning

Quicksort is a divide and conquer algorithm. it works by selecting a pivot element from the array and partitioning the other element into two sub array according to wheather they are less than or greater than the pivot.

The sub-arrays are then sorted recursively. This can be done in place requiring small addition amount of memory to perform the sorting.

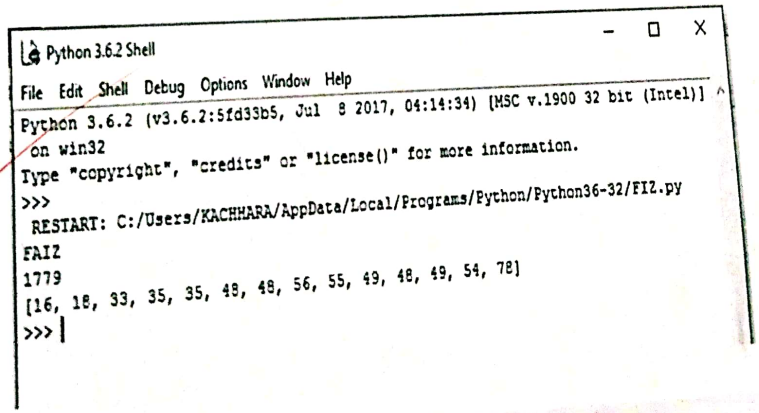
efficient implementations of Quicksort are not a stable sort meaning that the relative order of equal sort items is not preserved

Working

(a)	2	0	7	1	3	5	6	4
(b)	2	1	3	4	7	5	6	8
(c)	2	1	3	4	7	5	6	8
(d)	1	2	3	4	5	6	7	8
(e)	1	2	3	4	5	6	7	0

```
            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=[ 16,48,33,49,55,48,56,18,35,48,35,49,78,54]
quickSort(alist)
print(alist)
```

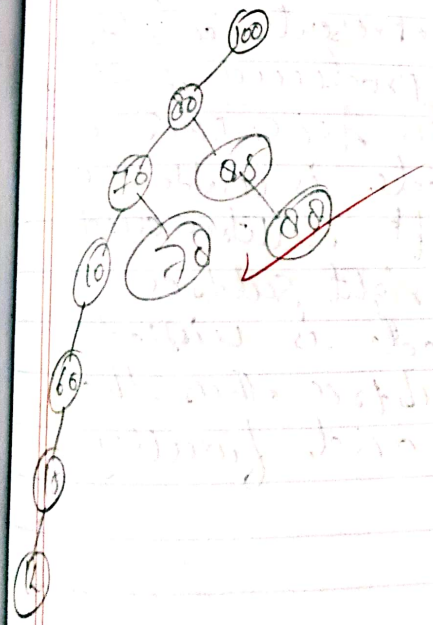
OUTPUT:



```
print("faiz")
print("1779")
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.postorder(start.l)
      print(start.data)
```

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 nodes.
A binary tree is an import class of a tree data structure in which a node can have at most two children



Diagramatic Representation of Binary search tree

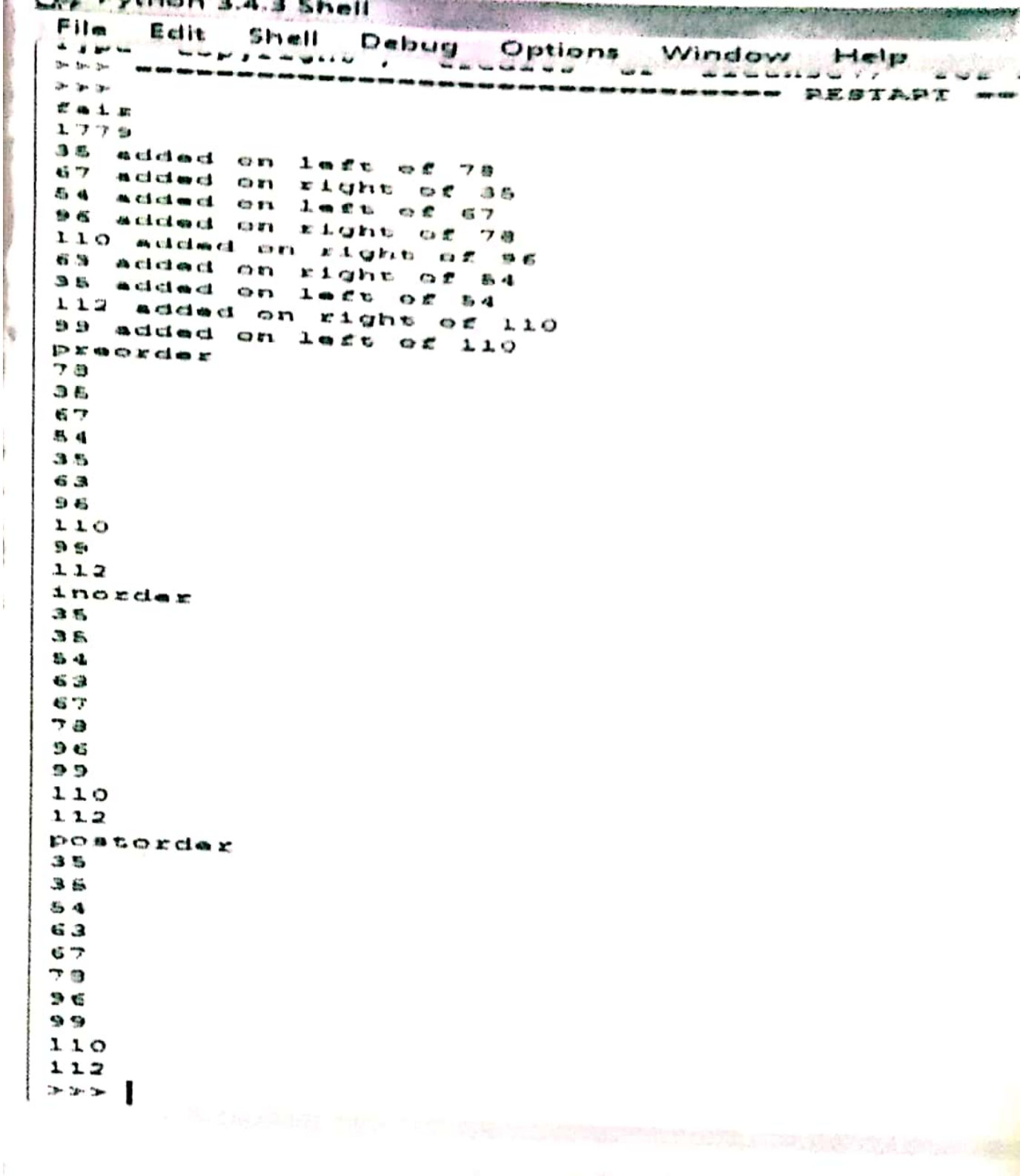
Traversal is a process to visit all the nodes of a tree and may print their values too.
There are 3 ways which we use to traverse a tree.

- INORDER
- PREORDER
- POSTORDER

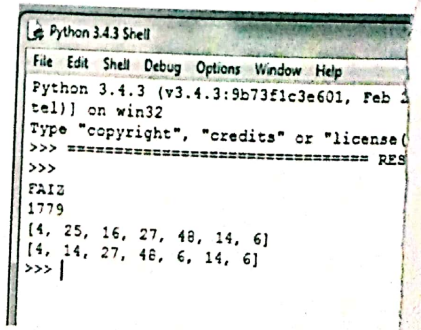
IN-ORDER : the left subtree is visited 1st then the root & later the right subtree we should always remember that every node may represents a subtree, itself. output produced is sorted key values in ASCENDING ORDER

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 rightsubtree and finally root node.



```
print("FAIZ")
print("1779")
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=1
    while i<n1 and j<n2:
        if L[i]<=R[j]:
            arr[k]=L[i]
            i+=1
        else:
            arr[k]=R[j]
            j+=1
            k+=1
    while i<n1:
        arr[k]=L[i]
        l+=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=[4,25,16,27,48,14,6]
print(arr)
n=len(arr)
mergesort(arr,0,n-1)
print(arr)
```



im: Merge sort

ory: 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 algorithem.

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 merge the two sorted halves. the merge() function is used for merging 2 halves the merge(arr, l, m, r) is key process that assume that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.