```
print("pinank")
a=[5,4,6,24,56,24,5,2]
j=0
print(a)
s=int(input("enter no to be searched: "))
for i in range(len(a)):
    if(s==a[i]):
        print("no founded at ",i+1)
        j=1
        break
if(j==0):
    print("number not founded")
```

output:-

```
>>>
================ RESTART: C:\Users\Shree\Desktop\DS\p1 ds.py =============
pinank
[5, 4, 6, 24, 56, 24, 5, 2]
enter no to be searched: 4
no founded at  2
>>>
================ RESTART: C:\Users\Shree\Desktop\DS\p1 ds.py ================
pinank
[5, 4, 6, 24, 56, 24, 5, 2]
enter no to be searched: 01
number not founded
>>>
```

PRACTICAL- 1

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

Theory : The process of identifying or finding a particular record is called searching.
There are two types of search
↳ Linear search
↳ Binary search
The linear search is further classified as
* Sorted * UNSORTED
Here we will look on the UNSORTED linear search, also known as sequential search, is a process that checks every element in the list sequentially until the desired element is found.
when the element is found.
When the element 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 in entered in random manner.
↳ User needs to specify the element to be searched in entered list.

↳ Check the condition that whether the entered list num matches, if it mathaes then displ the location plus increment 1 as data is stored from location 2
→ If all element are checked one one and element not found then prompt message number not found.

PROGRAM:

```
### After Before limkedlist(simple)###
print("PINANK RATHOD 1762")
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:
```

```
print("pinank")
a=[5,4,6,24,56,24,5,2]
j=0
print(a)
s=int(input("enter no to be searched: "))
for i in range(len(a)):
    if(s==a[i]):
        print("no founded at ",i+1)
        j=1
        break
if(j==0):
    print("number not founded")
```

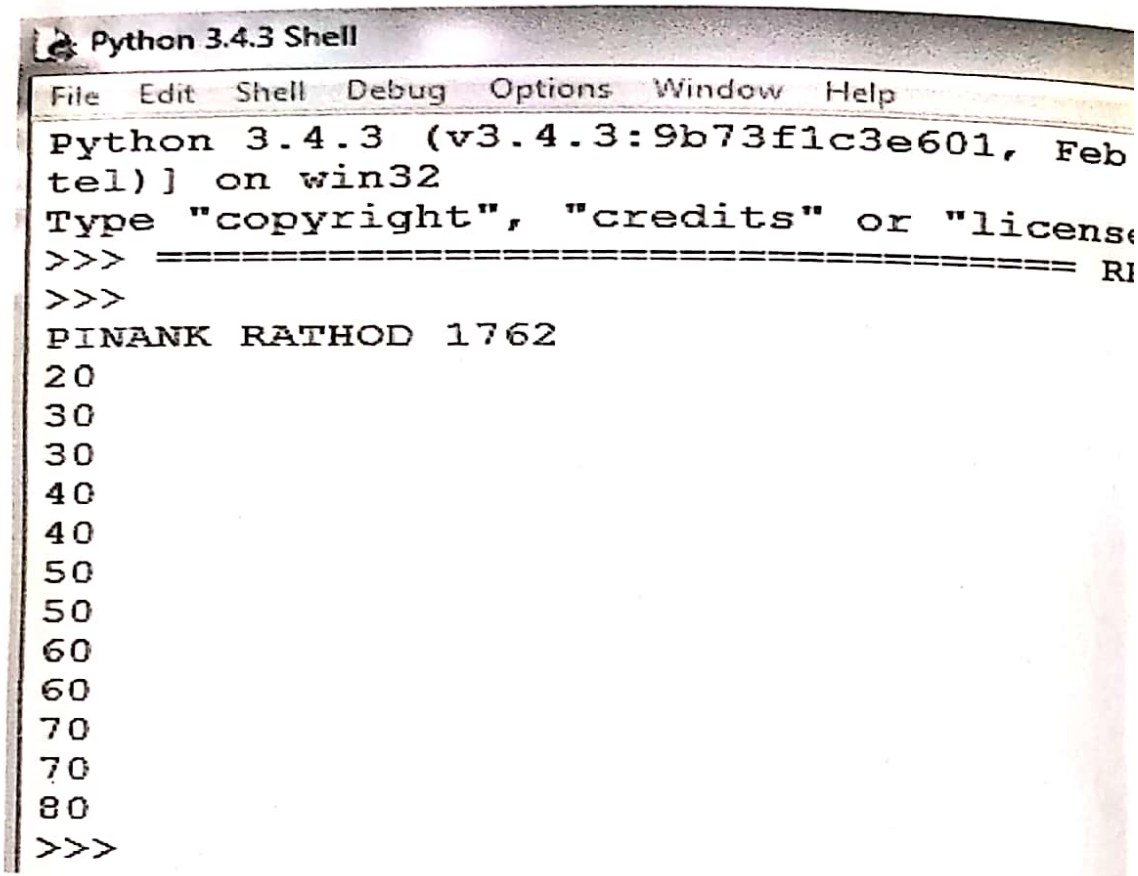
output:-

```
>>>
================ RESTART: C:\Users\Shree\Desktop\DS\p1 ds.py ============
pinank
[5, 4, 6, 24, 56, 24, 5, 2]
enter no to be searched: 4
no founded at  2
>>>
================ RESTART: C:\Users\Shree\Desktop\DS\p1 ds.py ================
abhinav singh
[5, 4, 6, 24, 56, 24, 5, 2]
enter no to be searched: 01
number not founded
>>>
```

SORTED LINEAR SEARCH

→ The user is supposed to ente data in sorted manner.
→ User has given an element f searching through sorted f list.
→ If element is found display u an updation as value is sorte listed.
→ If element is found display u an updation as value is sto from location '0'.
→ If data or element not fo point the same.
→ In sorted order list order elements we can check the condition that whether the entered number lies from starting point till the last e if not then without any processing we can say number not in the list.

OUTPUT:



1

```
print("pinank")
a=[3,6,9,25,58,78,99]
print(a)
s=int(input("enter the no to searched"))
b=0
c=len(a)-1
d=int((b+c)/2)
if((s<a[b])or(s>a[c])):
   print("number not in range")
elif(s==a[c]):
   print("no founded at :  ",c+1)
elif(s==a[b]):
   print("no founded at",b+1)
else:
   while(b!=c):
      if(s==a[d]):
         print("no founded at ",d+1)
         break
      else:
         if(s<a[d]):
            c=d
            d=int((b+c)/2)
         else:
            b=d
            d=int((b+c)/2)
      if(s!=a[d]):
         print("no absent")
         break
```

output:-

```
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p3 ds.py ==========
pinank
[3, 6, 9, 25, 58, 78, 99]
enter the no to searched25
no founded at  4
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p3 ds.py ==========
pinank
[3, 6, 9, 25, 58, 78, 99]
enter the no to searched10
no absent
>>>
```

PRACTICAL - 3

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

Theory : A binary search also known as a half - interval search is an algorithm used in computer science to locate a specified value (key) within an array. For the search to be binary the array must be sorted in either ascending or descending order.

At each step of the algorithm a comparison 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 than or greater that this middle element, the algorithm knows which half of the array.

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

This process is repeated on progressively smaller segments of the array until the value is located.

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

P (1)

```
## BUBBLE SORT ##
print("PINANK RATHOD 1762")
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)
```

```
OUTOUT:-
PINANK RATHOD 1762
List before BUBBLE SORT:
[8,5,1,2,3,6]
List after BUBBLE SORT:
[1,2,3,5,6,8]
```

PRACTICAL - 4

AIM: To sort given random data by usin bubble sort.

THEORY: A binary search also known as a half-interval search, is an algorithm used in computer science to be binary, the array must be sorted in either ascending or descending order.

BUBBLE sort sometimes referred as sinking sort.

It a simple sorting algorithm that repeatly steps through the lists, compares adjacent elements & 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 in 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 the 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 & swap 5 > 1
(1 5 4 2 8) → (1 4 5 2 8) → 5 > 4
(1 4 5 2 8) → (1 4 2 5 8) 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) since 4 > 2
(1 2 4 5 8) → (1 2 4 5 2)

Third Pass
(1 2 4 5 8) it checks & gives the data in sorted order.

```
## Stack ##
print("pinank")
class stack:
    global tos
    def __init__(self):
        self.l=[0,0,0,0,0,0,0,0]
        self.tos=-1
    def push(self,data):

        n=len(self.l)
        if self.tos==n-1:
            print("stack is full")
        else:
            self.tos=self.tos+1
            self.l[self.tos]=data
    def pop(self):
        if self.tos<0:
            print("stack empty")
        else:
            k=self.l[self.tos]
            print("data=",k)
            self.tos=self.tos-1
s=stack()
s.push(10)
s.push(20)
s.push(30)
s.push(40)
s.push(50)
s.push(60)
s.push(70)
s.push(80)
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
```

OUTPUT:-

pinank
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty

PRACTICAL : 5

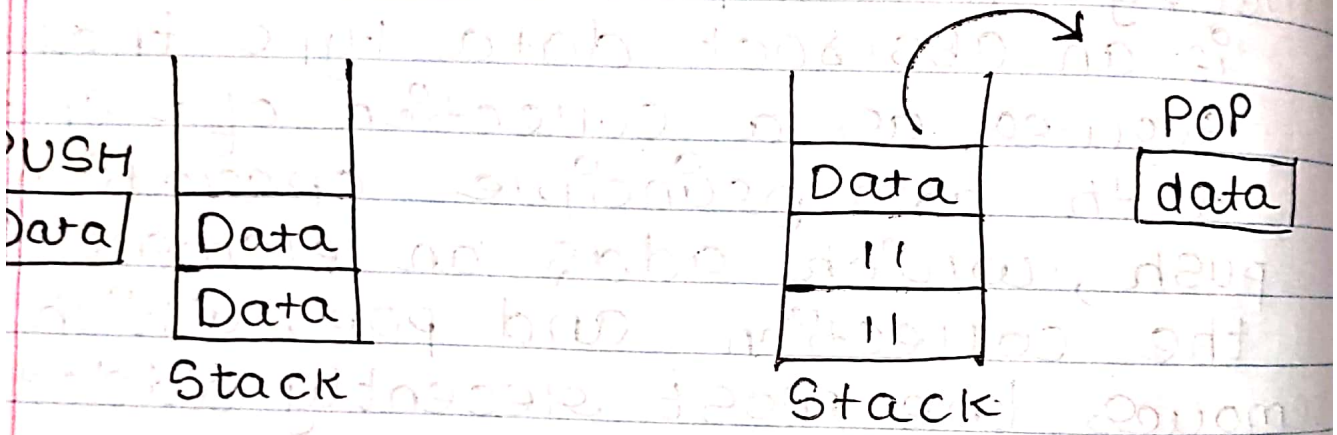
Aim : To demmonstrate the use of stack.

Theory : In computer Science, a stack is an abstract data type that serves as a collection of elements with two principle operation. 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 the stack.

- PUSH : Adds an item in the. If the stack is full said be over flow codition.
- POP : Remove an item from the stack. The item are popped in the reversed order in which they are pushed. If the stack is empty, them ~~from~~ it is said to be an underflow condition.

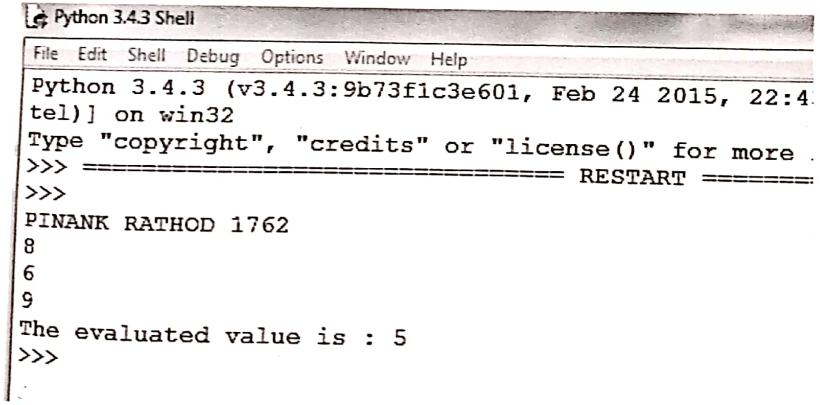
- Peek or top : Returns top elem of stack.
- isEmpty : Returns true if Stack empty else false



Last-in - Fao First-out.

```
            b=stack.pop()
            stack.apend(int(b)/int(a))
    return stack.pop()
s="8 6 9 - +"
r=evaluate(s)
print("The evaluated value is :",r)
```

OUTPUT:



```
## Queue add and Delete #
print("pinank")
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()
```

output:-

```
>>>
 RESTART: C:/Users/Shree/AppData/Local/Programs/Python/Python37-32/que n delete.py
pinank
Queue is full
30
40
50
60
70
Queue is empty
```

PRACTICAL - 6

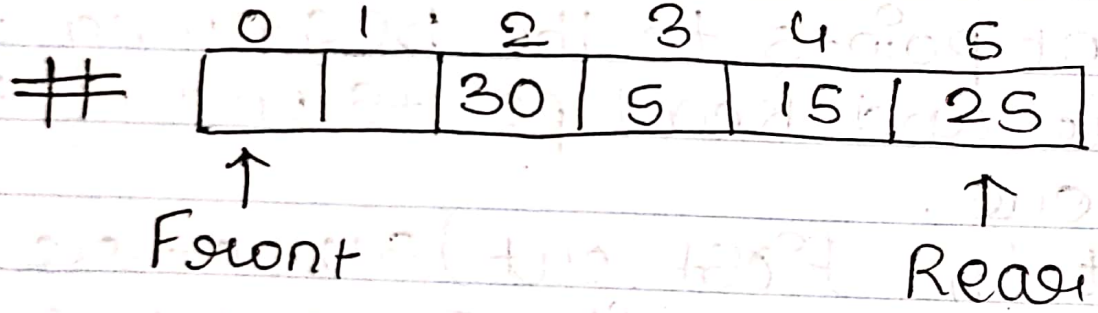
Aim: To demonstrate Queue add and delete

Theory: Queue is a linear data structure where the first element is inserted from one end called REAR and deleted from the others end called as FRONT
Front points to the beginning of the queue and Read points end of the queue.
(First-in - First-out) Structure.
According to its FIFO structure element.
In a queue, one end is always use to insert data (enqueue) and the other is used to data at both of its end.
Enqueue() can be termed as add() in queue i.e adding a element in queue
Dequeue() can be termed as delete or Remove. ie deleting or removing of element.

Front is used get the front da
item from a queue.

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

or ↲ queue. ↳ can have
both end.
side



Front
Rear=

```
#(circular queue)
print("pinank")
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)

output:-
>>>
 RESTART: C:/Users/Shree/AppData/Local/Programs/Python/Python37-32/circluar que.py
pinank
data added: 44
```

PRACTICAL - 7

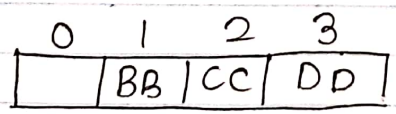
Aim: To demonstrate the use of circular queue is data structure.

Theory: The queue that we implement using an array suffer from one limitation. In that implementation there is a possible 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 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:

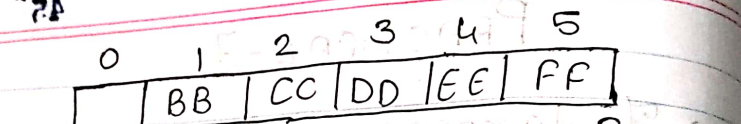
Front = 0

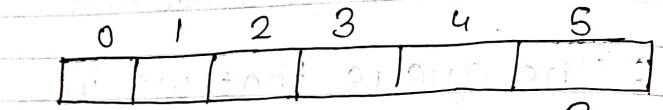
Rear = 3

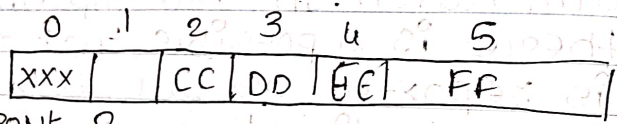


Front = 1

Rear = 3







```
data added: 55
data added: 66
data added: 77
data added: 88
data added: 99
data removed: 44
```

PROGRAM:

```
### After Before limkedlist(simple)###
print("PINANK RATHOD 1762")
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:
```

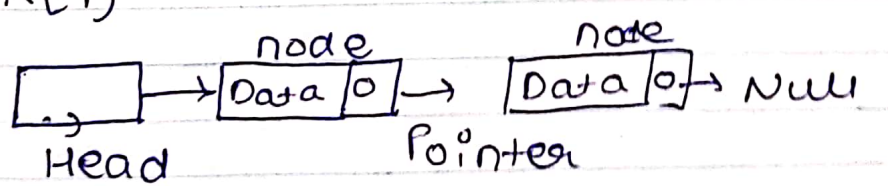
PRACTICAL - 8

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. Each link contains a connection to another click.

- Link: Each link of a linket list can store a data an element
- Next: Each link of a linked list contains a link to the next link called NEXT
- LINKED: A Linked list contains the connection link to the first link called first.

LINKED



5A

TYPES OF LINKED LIST.

- ↳ SIMPLE
- ↳ DOUBLY
- ↳ CIRCULAR

BASIC operation

- ↳ Insection
- ↳ Delection
- ↳ Display
- ↳ Search
- ↳ Delete

PROGRAM:

```
##Postfix Evaluation##
print("PINANK RATHOD 1762")
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()
```

PRACTICAL : 9

Aim : To evaluate postfix expression using stack

Theory : Stack is a (ADT) and work & LIFO i.e PUSH & POP operation.

Step to be followed

1) Read all the symbols & one by one left to right in the given by 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 two poopped operands in two different variables.

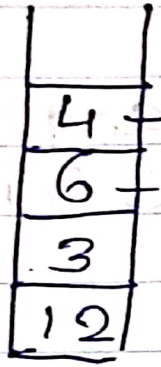
Then perform reading symbols operator using operand & operand 2 & push reset back on to the stack.

4) Finally : Perform a pop operation & display the popped value as final.

value of postfix expression

S = 12 3 6 4 - + *

Stack

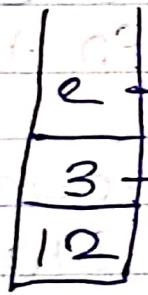


a

b

b - a = 6 - 4 = 2 ||

store again in Sta

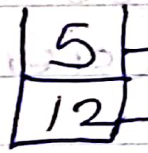


e → a

3 → b

b + a = 3 + 2 = 5 ||

store reset in Sta



a

b

b * a = 12 * 9 = 6

/

```
            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()
```

```
list are ",a)
ses):
,a)
: C:\Users\Shree\Desktop\DS\p4.py ========
 [56, 45, 23, 12, 2, 1]
2, 23, 45, 56]
```

```
## QUICK SORT ##
printf("PINANK 1762")
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:-
PINANK 1762
List before QUICK SORT
[42,54,45,67,89,66,55,80,100]
List after QUICK SORT:
[42,45,54,55,66,67,80,89,100]
```

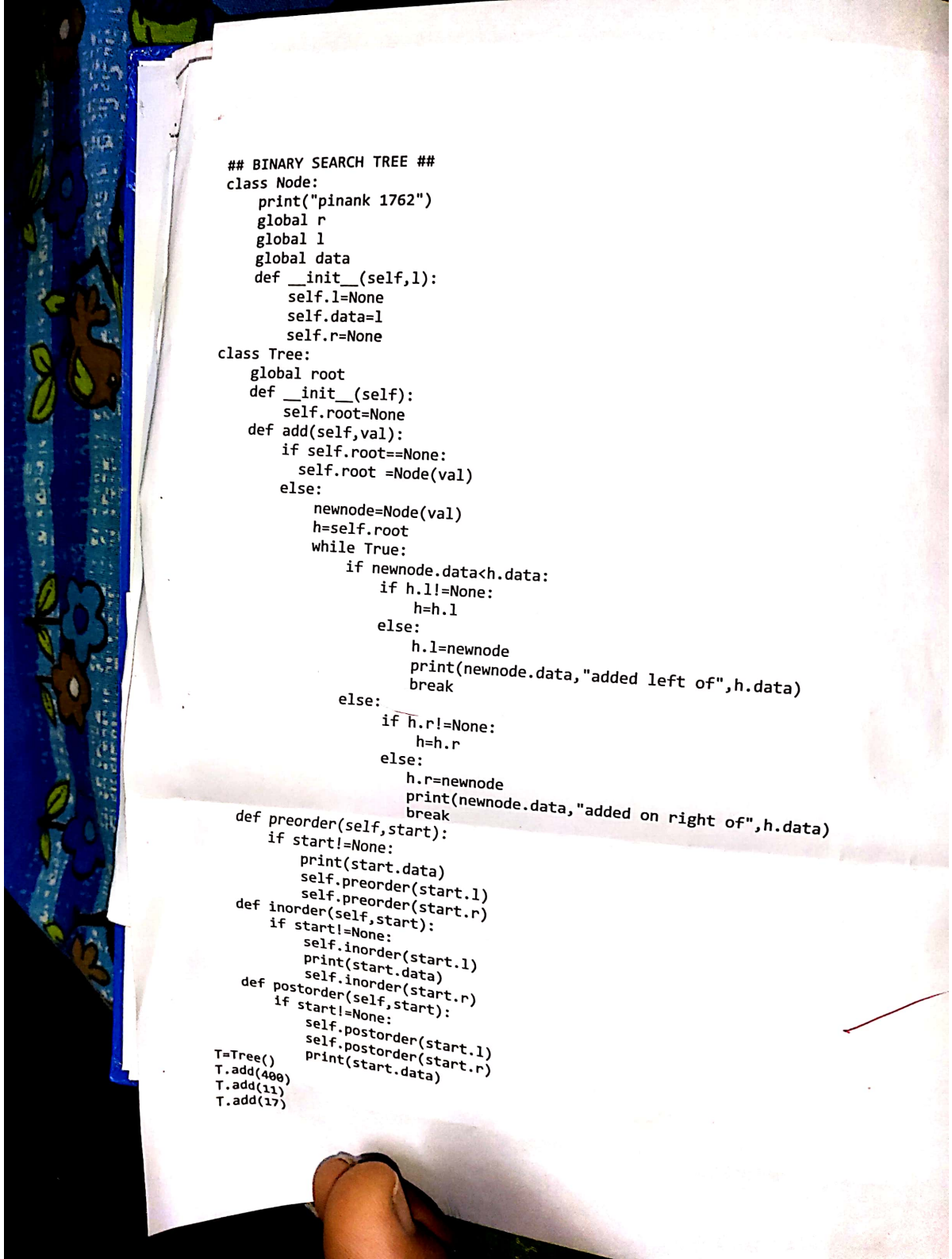
PRACTICAL - 10

Aim: To 'evaluate' ie ~~the sort~~ sort the given data in quick sort.

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

1) Always pick first element as pivot
2) Always pick the 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 partitions in given an array and an element 'x' of array as pivot; put x ~~at~~ its correct position in sorted array and put all smaller elements (smaller than x) before x, & put all greater elements (greater than x) after x. All this should be done in linear time.

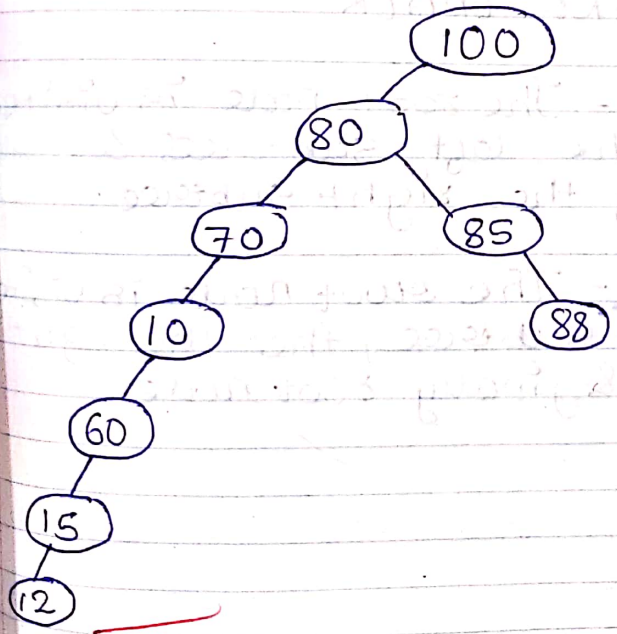


```
## BINARY SEARCH TREE ##
class Node:
    print("pinank 1762")
    global r
    global l
    global data
    def __init__(self,l):
        self.l=None
        self.data=l
        self.r=None
class Tree:
    global root
    def __init__(self):
        self.root=None
    def add(self,val):
        if 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 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)
            self.postorder(start.r)
            print(start.data)
T=Tree()
T.add(400)
T.add(11)
T.add(17)
```

PRACTICAL - 11

Aim : Binary Tree & Traversal

Theory: A binary tree is a special type of tree which every node or vertex has either no child or one child node or two child nodes.



diagramatic Representation of BINARY SEARCH TREE

Traversal is process to visit all node of a tree & print their value too

3 ways we use traverse a tree

a) IN-ORDER: The left-subtree is visited 1st then the root & later the right subtree we should always remember that every node may represent a subtree values in ASCENDING ORDER

b) PRE-ORDER - The root node is visited 1st then the left subtree & finaly the right subtree.

c) POST-ORDER - The root node is visited last left subtree, then the right subtree & finally root node.

T=1
T.i
T.i
T.a

```
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)

=============== RESTART: C:/Users/Shree/Desktop/ppp abhinav.py ===============
pinank 1762
11 added left of 400
17 added on right of 11
50 added on right of 17
30 added left of 50
47 added on right of 30
preorder
400
11
17
50
30
47
inorder
11
17
30
47
50
400
postorder
47
30
50
17
11
400
>>>
```

```
print("pinanak 1762")
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)

>>>
========== RESTART: C:/Users/Shree/Desktop/class/merage sort.py ==========
pinank 1762
array before sorting
 [12, 23, 34, 56, 78, 45, 86, 98, 42]
array after merge sorting
 [12, 23, 34, 56, 42, 45, 78, 86, 98]
>>>
```

PRACTICAL - 12

Aim: Merge Sort

Theory: Merge sort is a sorting technique based on divide & conquer technique. With worst-case time complexity being (n log n), its one of the most respected algorithm.

Merge Sort first divides the array into equal halves and then combines them in sorted manner.

It divides input array in two halves calls itself for two halfes & then merges sorted halves. The merge (arr(), m, r) is key process that assumes that arr[l....m] & arr[m+1..r] are sorted & merges the two sorted sub array into one.

PRACTICAL - 13

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

Theory: The selection sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.

The selection sort improved on the bubble 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.

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

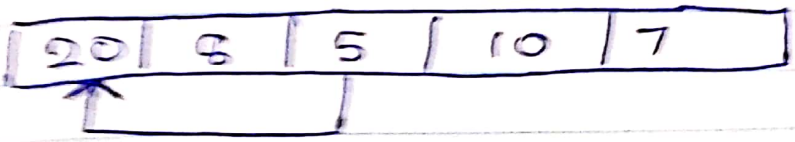
```
de
arr
pri
n=1
me
pri

>>>
=====
pinar
array
[12,
array
[12,
>>>
```

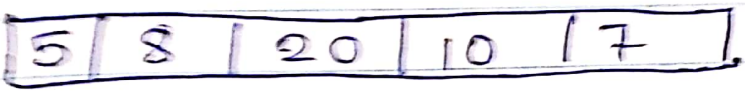
```
## SELECTION SORT ##
print("pinank 1762")
a=[23,22,18,96,56,60]
print("Before sorting \n",a)
for i in range(len(a)-1):
 for j in range(len(a)-1):
  if(a[j]>a[i+1]):
    t=a[j]
    a[j]=a[i+1]
    a[i+1]=t
print("After selection sort \n",a)

========== RESTART: C:/Users/Shree/Desktop/class/selection sort.py ==========
pinank 1762
Before sorting
 [23, 22, 18, 96, 56, 60]
After selection sort
 [18, 22, 23, 56, 60, 96]
>>>
```

eg. on each pass the smallest remaining item is selected, and then placed in its proper location



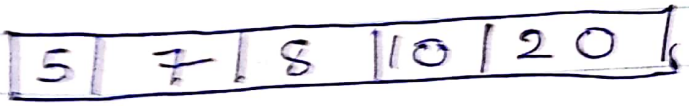
5 is smallest



7 is smallest

8 is smallest

10 ok list is sorted.



20 ok list is sorted.