PRACTICAL NO- 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

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 element to be searched are not specifically arranged in ascending or descending order. That is why it calls unsorted Linear Search.

08

Program:

```
print("Jai Doshi:1761")
found=False
A=[14,31,30,22,85]
search=int(input("No.To Be Searched:"))
for i in range(len(A)):
  if(search==A[i]):
    print("No. Is Found At The Location:",i)
    found=True
    break;
if (found==False):
  print(search,"Donot Exists")
```

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
====== RESTART: D:/Softwares/College/Semester 2/DS/Program/Unsorted.py ======
Jai Doshi:1761
No.To Be Searched:30
No. Is Found At The Location: 2
>>>
====== RESTART: D:/Softwares/College/Semester 2/DS/Program/Unsorted.py ======
Jai Doshi:1761
No.To Be Searched:15
15 Donot Exists
>>>
```

1. The data is entered in random manner.
2. User needs to specify the elem to be searched
3. check the condition whether the entered number matches, if the entered number matches display the location plus increment by 1 as data is stored from location zero.
4. If all elements are checked one by one and element not found then prompt message number not found.

PRACTICAL NO- 2

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

Theory: SEARCHING & SORTING are different modes of data structure.

SORTING: To basically sort the input data in ascending or it in the descending order.

SEARCHING: To search & display the desired element.

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 the well arranged data.

1. The user is supposed to enter the data in sorted manner.

2. User has to give an element for searching through sorted list.

Program:

```
print("Jai Doshi:1761")
found=False
A=[14,30,31,45,85]
search=int(input("No.To Be Searched:"))
if(search<A[0] or search>A[len(A)-1]):
  print(search,"DoesNot Exists")
else:
  for i in range(len(A)):
    if(search==A[i]):
      print("No. Is Found At The Location:",i)
      found=True
      break;
if (found==False):
  print(search,"Donot Exists")
```

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======= RESTART: D:/Softwares/College/Semester 2/DS/Program/Sorted.py =======
Jai Doshi:1761
No.To Be Searched:80
80 Donot Exists
>>>
======= RESTART: D:/Softwares/College/Semester 2/DS/Program/Sorted.py =======
Jai Doshi:1761
No.To Be Searched:14
No. Is Found At The Location: 0
>>>
```

3. If element is found display position of an operation as value is stored from location zero.

4. If data or element not found print the same.

5. 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.

PRACTICAL No- 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 a 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 less than or greater than from this middle element, the algorithm knowns which half of the array to continue searching in, because the array is sorted.

18.

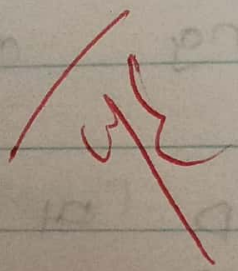
Program:

```
print("Jai Doshi:1761")
A=[21,39,49,58]
search=int(input("Enter The Number Search:"))
l=0
r=len(A)-1
while True:
  m=(l+r)//2
  if(l>r):
    print(search,"Not Found")
    break;
  if(search==A[m]):
    print("Number Is Found At Location:",m)
    break;
  else:
    if(search<A[m]):
      r=m-1
    else:
      r=m+1
```

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: D:/Softwares/College/Semester 2/DS/Program/Binary.py ========
Jai Doshi:1761
Enter The Number Search:14
14 Not Found
>>>
======== RESTART: D:/Softwares/College/Semester 2/DS/Program/Binary.py ========
Jai Doshi:1761
Enter The Number Search:39
Number Is Found At Location: 1
>>>
```

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.



PRACTICAL NO-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 & pop, which removes the most recently added element that was not yet removed. The order may be LIFO (Last In First Out) or the FILO (First In Last Out).

Three Basic operations are performed in the stack.

1. Push: Adds an item in the stack. If the stack is full then it is said to be over flow condition.

2. Pop: Removes an item from the stack. The items are popped in the required order in which they are pushed.

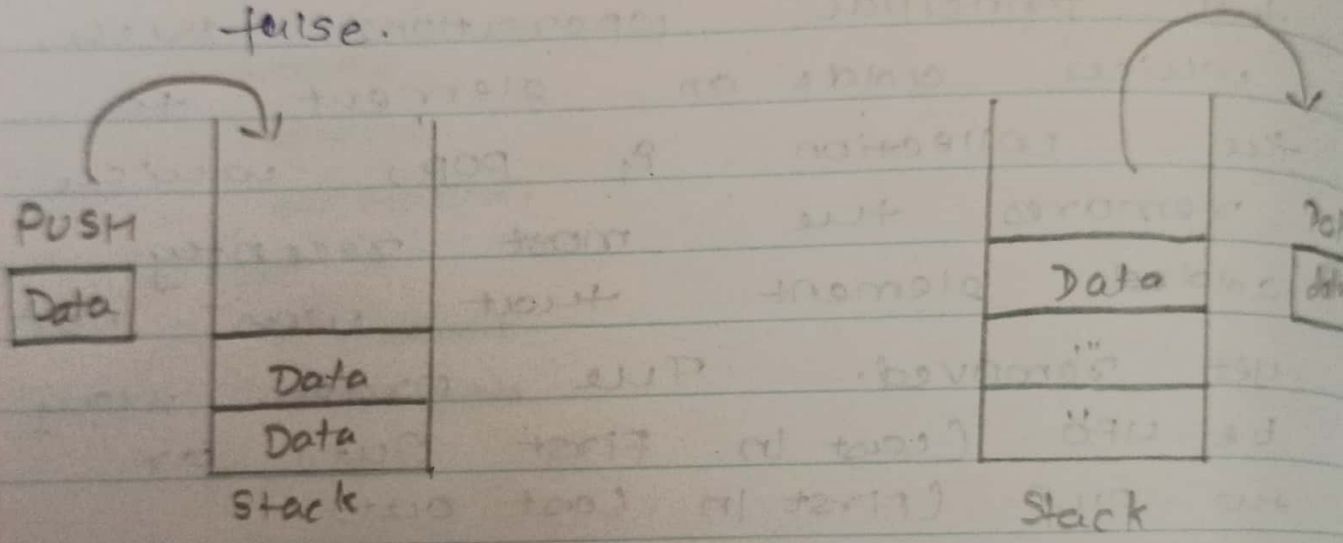
Program:

```
print("Jai Doshi:1761")
class stack:
    global tos
    def __init__(self):
        self.l=[0,0,0,0,0,0,0]
        self.tos=-1
    def push(self,data):

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

3. Peek or TOP: Returns top element of stack.

4. isEmpty: Return true if if stack is empty else false.



Last_in_first_Out

```
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
```

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: D:/Softwares/College/Semester 2/DS/Program/Stack.py ========
Jai Doshi:1761
Stack Is Full
Data= 70
Data= 60
Data= 50
Data= 40
Data= 30
Data= 20
Data= 10
Stack Is empty
>>>
```

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 delete from the other end called as FRONT.

Front points to the begining of the queue & Rear points to the end of the queue.

Queue follows the FIFO (FIRST IN first out) structure.

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

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

Program:

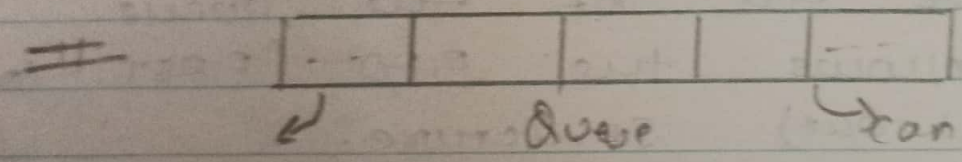
```
class Queue:
    global r
    global f
    def __init__(self):
        self.r=0
        self.f=0
        self.l=[0,0,0,0,0,0]
    def add(self,data):
        n=len(self.l)
        if self.r<n-1:
            self.l[self.r]=data
            self.r=self.r+1
        else:
            print("Queue is 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()
```

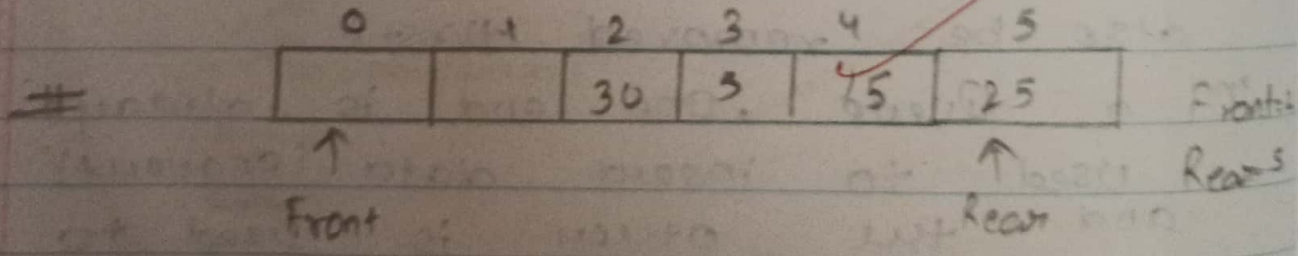
enqueue() can be termed as add() in queue i.e to add a element in the queue.

Dequeue() can be termed as delete or remove i.e deleting or removing element.

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

Rear is used to Get last data item from a queue.





Q.remove()

Q.remove()

Q.remove()

Q.remove()

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: D:/Softwares/College/Semester 2/DS/Program/Queue.py ========
Queue is full
30
40
50
60
70
Queue is empty
>>>
```

4/2

PRACTICAL NO-6

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

Theory:

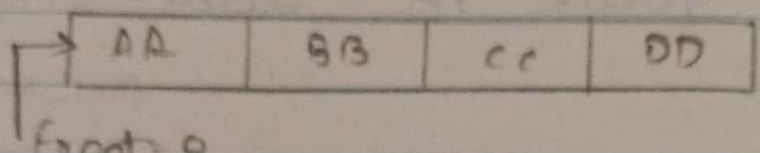
The queue that we implement using an array suffer from one limitation. In that implemen-tation there is a possibility that the queue is reported as full, even though in actuality there might be empty slots at the beginning of the queue.

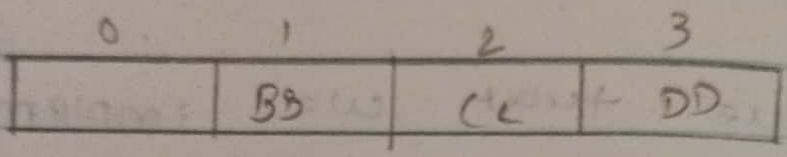
To overcome this limitation we can implement queue as the circular queue. In circular queue we go on adding the element to the queue & reach at the end of the array, the next element is stored in the first slot of the array.

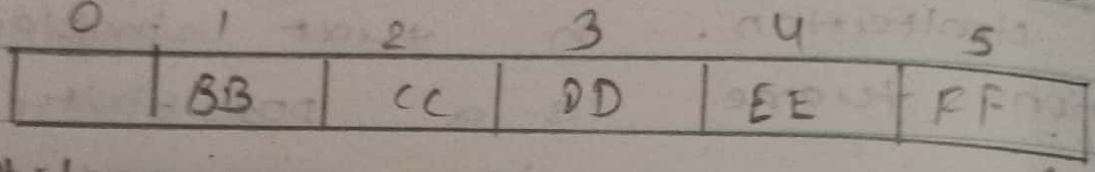
Program:

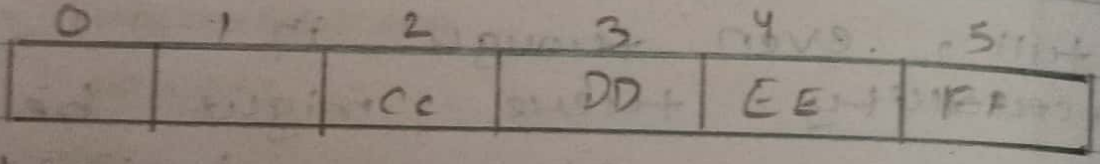
```
class Queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if self.r<=n-1:
      self.l[self.r]=data
      print("Data Added:",data)
      self.r=self.r+1
    else:
      s=self.r
      self.r=0
      if 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
```

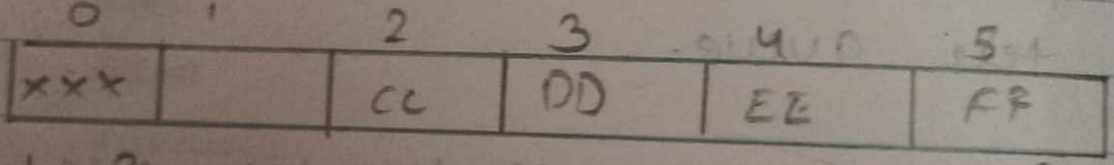
Example:-











```
        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)
```

```
Python 3.7.4 (tags/v3.7.4:09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: D:/Software/College/Semester 2/DS/Program/CircularQueue.py ====
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 linked list in data structure.

Theory:

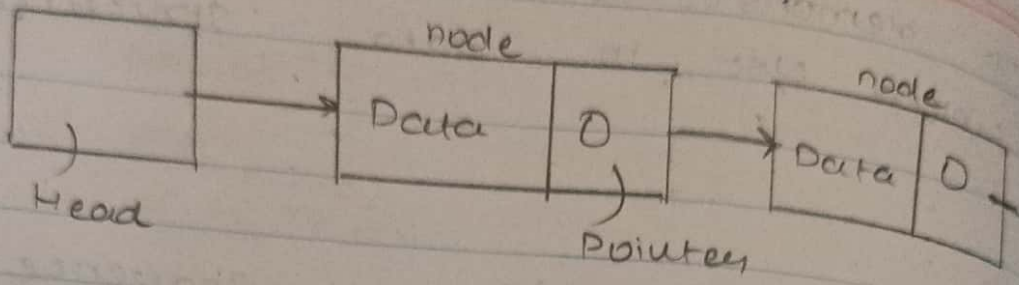
A linked list is a sequence of data structures. Linked list is a sequence of links which contains items. Each link contains a connection to another link.

- LINK - Each link of a linked list can store a data called an element.
- NEXT - Linked list contains a links to the next link called NEXT.
- LINKED LIST - A linked list contains the connection link to the first link called FIRST.

Program:

```
print("Jai Doshi:1761")
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
```

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

```
        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()
```

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Jai Doshi:1761
20
30
40
50
>>>
```

PRACTICAL NO-8

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 the Oporands in which the operators is placed after the operand.

STEPS:

1. Read all the symbols one by one from left to right in the given ~~POSTfix~~ expression.

2. If the reading symbol is operand then push it onto the stack.

3. If the reading symbol is operator (+, -, *, /, etc) then Perform two pop operations & Store the two popped Operands in two different variables (Operand 1 & Operand 2). Then

Program:

```
print("Jai Doshi:1761")
def evalute(s):
 k=s.split()
 n=len(k)
 stack=[]
 for i in range(n):
  if k[i].isdigit():
   stack.append(int(k[i]))
  elif k[i]=='+':
   a=stack.pop()
   b=stack.pop()
   stack.append(int(b)+int(a))
  elif k[i]=='-':
   a=stack.pop()
   b=stack.pop()
   stack.append(int(b)-int(a))
  elif k[i]=='*':
   a=stack.pop()
   b=stack.pop()
   stack.append(int(b)*int(a))
  else:
   a=stack.pop()
   b=stack.pop()
   stack.append(int(b)+int(a))
 return stack.pop()
```

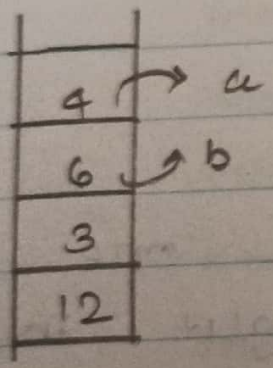
Perform reading Symbol operator using Operand1 & operand 2 & Push result back on the Stack.

4. Finally: Perform a pop operator and display the popped value as final result.

Value of postfix expression:

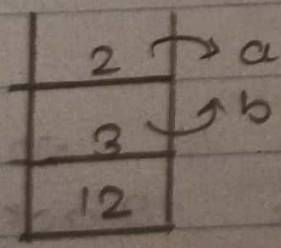
S = 12 3 6 4 - + *

Stack:



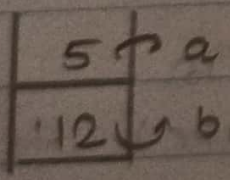
b - a = 6 - 4

= 2 // store answer in stack



b + a = 3 + 2

= 5 // store result in stack



b * a = 12 * 5

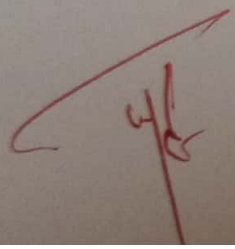
= 60

```
s="8 7 6 + *"

r=evalute(s)

print("The Evaluted Value is:",r)
```

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Jai Doshi:1761
The Evaluted Value is: 104
>>>
```



PRACTICAL NO. 9

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

Theory:

SORTING:

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

Bubble sort sometimes referred as sinking sort.

It is a simple sorting algorithm that repeatedly steps through the lists, compares adjacent element & 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 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 it condition fails then only swaps otherwise goes on.

Program:

```
print("Jai Doshi:1761")
a=[10,8,9,5,3,1,2]
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)
```

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Jai Doshi:1761
[10, 8, 9, 5, 3, 1, 2]
[1, 2, 3, 5, 8, 9, 10]
>>>
```

5A

PRACTICAL NO-10

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

Theory:

Quicksort 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

Program:

```
print("Jai Doshi:1761")
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)
```

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Jai Doshi:1761
[42, 45, 54, 55, 66, 67, 80, 89, 100]
>>>
```

Q. The key process is quicksort is partition (). Target of partitions is given as array and an element x of array as pivot, put x as 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.

PRACTICAL NO - 11

Aim: To sort Given raudom data by using Selection sort

Theory:

Selection sort is a simple sorting algoritum. This sorting algoritum is an inplace comparison based algoritum in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

Program:-

```
print("Jai Doshi:1761")
b=[15,14,13,12,11,10]
print(b)
for i in range (len(b)-1):
    for j in range(len(b)-1):
        if (b[j]>b[i+1]):
            t=b[j]
            b[j]=b[i+1]
            b[i+1]=t
print(b)
```

```
========== RESTART: C:/Users/aayus/Desktop/jaipractical3.py ==========
Jai Doshi:1761
[15, 14, 13, 12, 11, 10]
[10, 11, 12, 13, 14, 15]
>>>
```

PRACTICAL NO-12

Aim: Binary Tree & 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 childrens.

Program:-

```
print("Jai Doshi:1761")
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
```

Traversal is a process to visit all the nodes of a tree and may print their values

There are 3 ways in which we use to traverse a tree.

(A) Inorder
(B) Preorder
(C) Postorder

(A) Inorder:

The left - subtree is visited 1st then the root & 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.

(B) PRE ORDER:

the root node is visited 1st then the left subtree & finally the right subtree

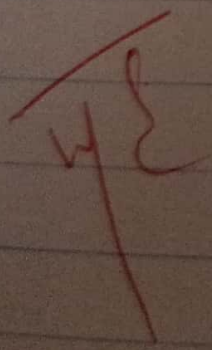
```
            else:
                h.r=newnode
                print(newnode.data,"added on right of",h.data)
                break
    def preorder(self,start):
        if start!=None:
            print(start.data)
            self.preorder(start.l)
            self.preorder(start.r)
    def inorder(self,start):
        if start!=None:
            self.inorder(start.l)
            print(start.data)
            self.inorder(start.r)
    def postorder(self,start):
        if start!=None:
            self.inorder(start.l)
            self.inorder(start.r)
            print(start.data)
T=Tree()
T.add(200)
T.add(80)
T.add(70)
T.add(85)
T.add(10)
T.add(79)
T.add(60)
T.add(88)
T.add(15)
T.add(14)
print("preorder")
```

```
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
========== RESTART: C:/Users/aayus/Desktop/jaipractical1.py ==========
Jai Doshi:1761
80 added on left of 200
70 added on left of 80
85 added on right of 80
10 added on left of 70
79 added on right of 70
60 added on right of 10
88 added on right of 85
15 added on left of 60
14 added on left of 15
preorder
200
80
70
10
60
15
14
79
85
88
inorder
10
14
15
60
70

70
79
80
85
88
200
postorder
10
14
15
60
70
79
80
85
88
200
>>>
```

(c) POST ORDER:

The root node is visited last, left sub-tree, then the right sub-tree & finally the root node.



PRACTICAL NO-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 Algoritem.

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

It divides input array into two halves, calls itself for the two halves & then merge the two sorted halves. The merge() function is used for merging two halves.

Program:-


```
print("Jai Doshi:1761")
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=[14,26,34,56,78,45,89,98,44]
print(arr)
n=len(arr)
mergesort(arr,0,n-1)
print(arr)
```

```
========== RESTART: C:/Users/aayus/Desktop/jaipractical2.py ==========
Jai Doshi:1761
[14, 26, 34, 56, 78, 45, 89, 98, 44]
[14, 26, 56, 56, 44, 45, 78, 89, 98]
>>>
```

The merge process (arr, l, r, m) is key process that assumes that arr [l...m] and arr [m+1...r] are sorted and merges the two sorted sub-array into one.