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 UN-SORTED 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 what it calls unsorted linear Search.
Unsorted linear Search
↳ The data is entered in random manner
↳ User needs to Specify the element to be Searched in the entered list.
↳ Check the condition that whether the entered number matches if it matches then display the location plus increment 1, as data is Stored from location Zero.

```
print("kushal")
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 ==========
kushal
[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 ==========
kushal
[5, 4, 6, 24, 56, 24, 5, 2]
enter no to be searched: 01
number not founded
>>>
```

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

Practical-2

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

THEORY:- SEARCHING and SORTING are different modes or types of data-structure

SORTING - To basically SORT the inputed data in ascending or descending manner

SEARCHING :- To search elements and to display the same

In searching that too in LINEAR SORTED :- Seach the data is arranged in ascending to descending or descending to ascending that is all what it meant by searching through 'sorted' that is well arranged data.

Sorted linear Search:

- The user is supposed to enter data in sorted manner
- User has to given an element for searching through sorted list
- If element is found display with an updation as value is stored from location '0'.

```
print("kushal")
a=[1,3,4,6,7,9,10]
j=0
print(a)
s=int(input("enter no to be searched: "))
if((s<a[0]))or((s>a[6])):
   print("number absent")
else:
   for i in range(len(a)):
      if(s==a[i]):
         print("number founded at: ",i+1)
         j=1
         break
   if(j==0):
         print("number not founded")
```

output:-

```
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p2 ds.py ==========
kushal
[1, 3, 4, 6, 7, 9, 10]
enter no to be searched: 3
number founded at:  2
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p2 ds.py ==========
kushal
[1, 3, 4, 6, 7, 9, 10]
enter no to be searched: 55
number absent
```

- If data or element not found print the same.
- In sorted order list of elements we can check the condition that whether the entered number lies from starting point till the last element if not then without any processing we can say number not in the list.

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

This process is repeated on progressively smaller segments of the array until the value is located.
Because each step in the algorithm divides the array size in half a binary search will complete successfully in logarithmic time.

```
print("kushal")
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 ==========
kushal
[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 ==========
kushal
[3, 6, 9, 25, 58, 78, 99]
enter the no to searched10
no absent
>>>
```

Practical-14

Aim:- To demonstrate the use of Stack

THEORY:- In computer science, a Stack is an abstract data type that serves as a collection of element with two principal operations 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 In First Out) or FILO (First In Last Out)

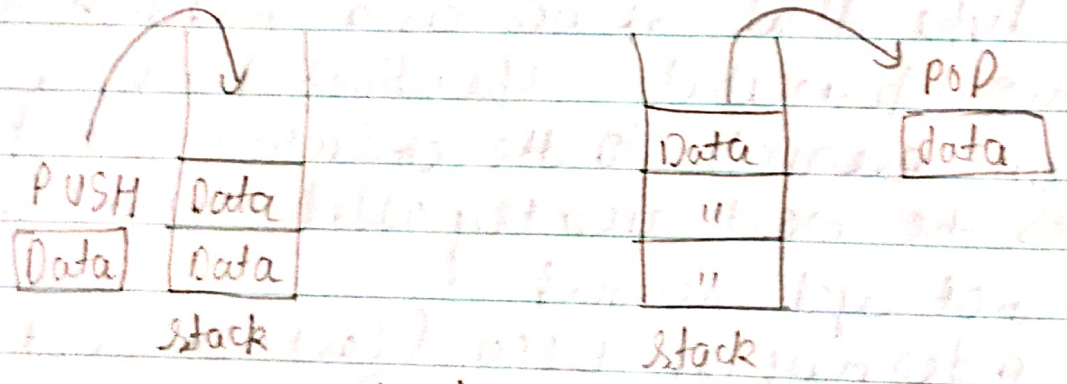
Three basic operations are performed in the Stack

- PUSH
- POP
- Peek or TOP
- is Empty

- PUSH :- Adds an item in the 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 reversed order in which they are pushed If the Stack is empty then it is said to be an underflow condition

- Peek or Top :- Return top element of Stack
- is Empty :- Return true if Stack is empty else false.



Last in - First - out

```
## Stack ##
print("KUSHAL SINGH")
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:-

```
KUSHAL SINGH
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty
```

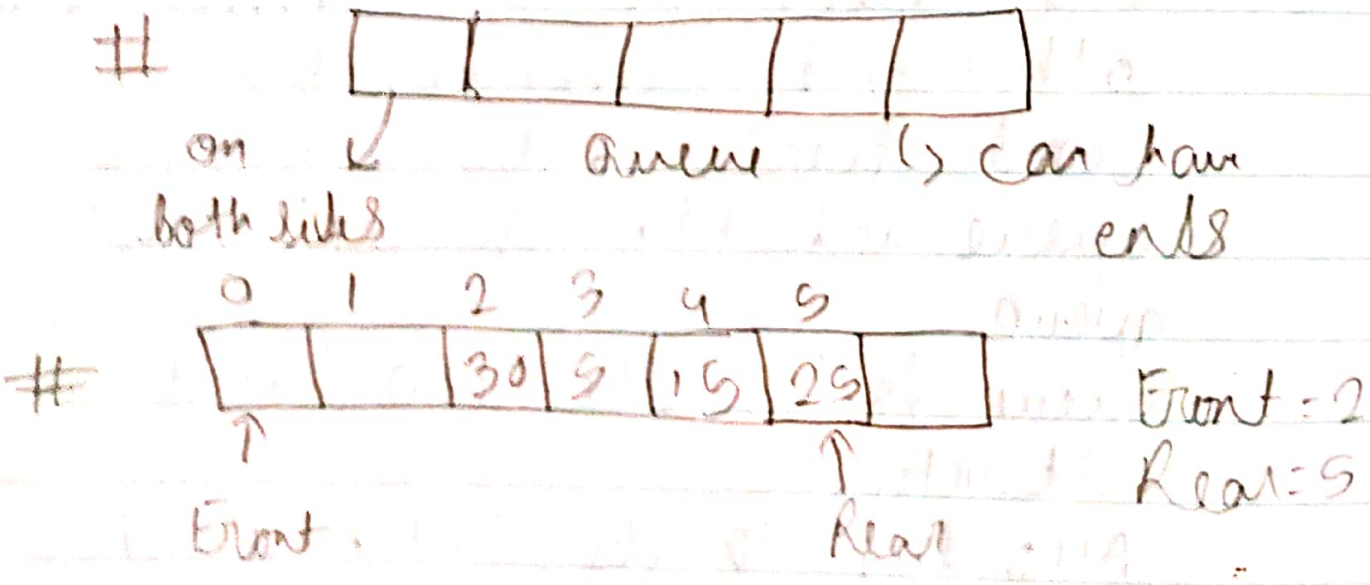
Practical - 05

AIM :- To demonstrate Queue add and delete.

THEORY :- Queue is a linear data structure where the first element is inserted from one end called REAR. and deleted from the other end called as FRONT.
Front points to the beginning of the queue and Rear points to the end of the queue.
Queue follows the FIFO (First In - 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 is queue.
Dequeue () can be termed as delete or Remove i.e deleting or removing of element.

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

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



```
## Queue add and Delete #
print("kushal")
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:-

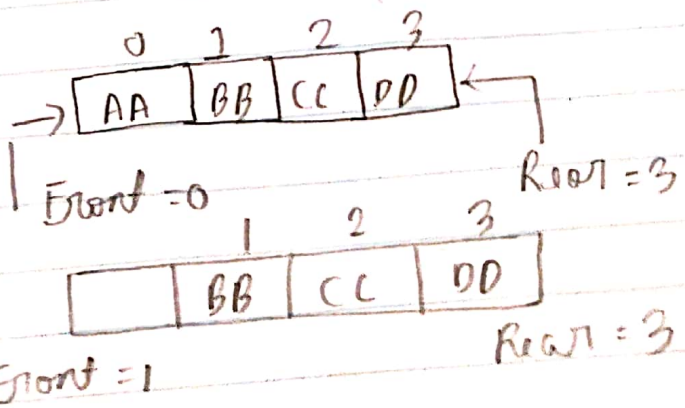
```
>>>
 RESTART: C:/Users/Shree/AppData/Local/Programs/Python/Python37-32/que n delete.py
kushal
Queue is full
30
40
50
60
70
Queue is empty
```

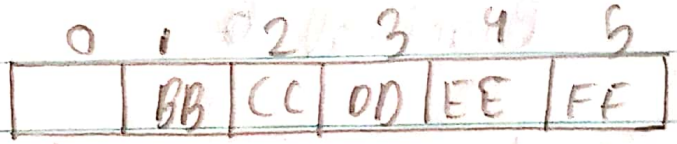
Practical -6

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

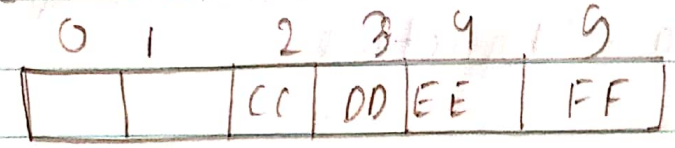
THEORY :- The queue that we implement using an array suffer from one limitation In that implementation there is a possibility that the queue is reported as full, even though in actually there might be empty slots at the begining of the queue.
To overcome this limitations we can implement queue as circular queue In circular queue we go on adding the element to the queue and reach the end of the queue and reach the end of the array.
The next element is stored in the first slot of the array.

Example :-

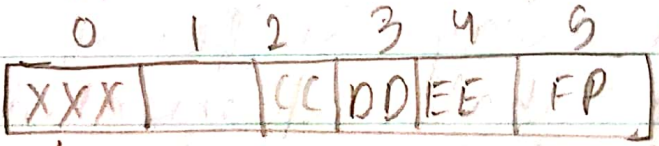




Front = 1 Rear = 5



Front = 2 Rear = 5



Front = 2 Rear = 0

No.

```
#(circular queue)
print("kushal")
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
kushal
data added: 44
```

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

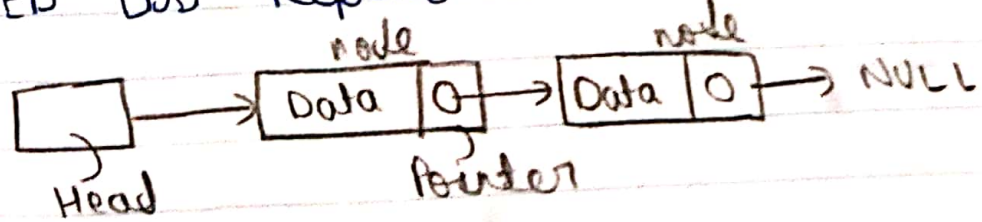
Practical-7

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- Each link of a linked list contains a link to the next link called NEXT.
- LINKED LIST - A linked list contains the connection link to the first link called First

LINKED list Representation:-



TYPES of LINKED LIST :-

- Simple
- Doubly
- circular

BASIC operations

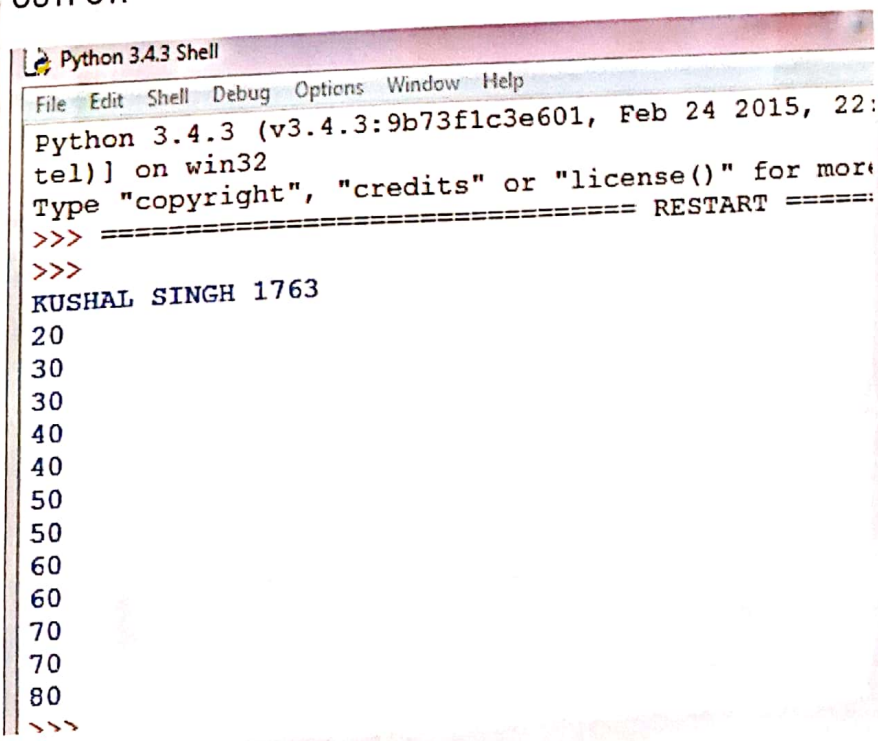
- Insertion
- Deletion
- Display
- Search
- Delete

PROGRAM:

```
### After Before limkedlist(simple)###
print("KUSHAL  SINGH 1763")
class node:
  global data
  global next
  def __init__(self,item):
    self.data=item
    self.next=None
class linkedlist:
  global s
  def __init__(self):
    self.s=None
  def addL(self,item):
    newnode=node(item)
    if self.s==None:
      self.s=newnode
    else:
      head=self.s
      while head.next!=None:
        head=head.next
      head.next=newnode
  def addB(self,item):
    newnode=node(item)
```

```
        if self.s==None:
            self.s=newnode
        else:
            newnode.next=self.s
            self.s=newnode
    def display(self):
        head=self.s
        while head.next!=None:
            print(head.data)
            head=head.next
            print(head.data)
start=linkedlist()
start.addL(50)
start.addL(60)
start.addL(70)
start.addL(80)
start.addB(40)
start.addB(30)
start.addB(20)
start.display()
```

OUTPUT:



Practical 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 operands in which the operator is placed after the operand.

Step to be followed

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

If the reading symbol is operand then push it on to the stack

If the reading symbol is operator (+, -, *, / etc) then perform two pop operations and store the two popped operands in two different variables (operand 1 & operand 2)

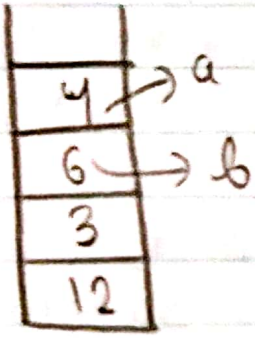
Then perform reading symbol operator using operand 1 & operand 2 and push reset back on to the stack

Finally ! Perform a pop operation and display the popped value as final reset.

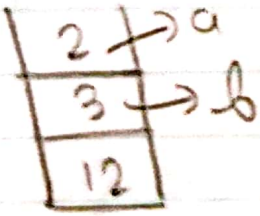
Value of postfix expression

S = 12 3 6 4 - + *

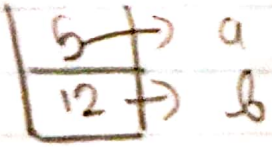
Stack



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



b + a = 3 + 2 = 5 // Store result in stack



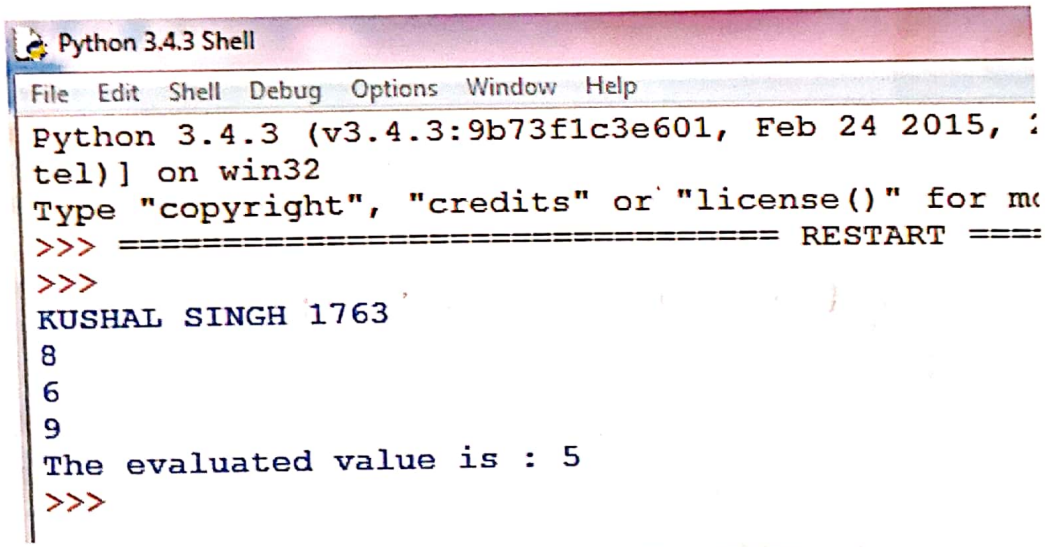
b * a = 12 * 5 = 60

PROGRAM:

```
##Postfix Evaluation##

print("KUSHAL SINGH 1763")

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

OUTPUT:



Practical-9

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 list, compares adjacent element and Swaps them if they are in wong order They pass through the list is repeated until the list is Sorted.
The algorithm which is a comparision Sort is named for the way Smaller or larger element's "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 order.

```
print("kushal ")
a=[56,45,23,12,2,1]
print("before bubble sorting element list are ",a)
for passes in range(len(a)-1):
   for compare in range (len(a)-1-passes):
      if(a[compare]>a[compare+1]):
         temp=a[compare]
         a[compare]=a[compare+1]
         a[compare + 1]=temp
print("after bubble sort elements list",a)
 output:-
>>>
========== RESTART: C:\Users\Shree\Desktop\DS\p4.py ==========
kushal
before bubble sorting element list are  [56, 45, 23, 12, 2, 1]
after bubble sort elements list [1, 2, 12, 23, 45, 56]
>>>
```

Practical 10

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

THEORY:- Quick Sort is an efficient Sorting algorithm Type of a Divide & 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) Pic median as pivot.

The key process in quick Sort is partition() Target of partitions is given an array and an element X of array as pivot but X at its current position in Sorted array and put all Smaller element (Smaller than X) before X, & put all greater elements (greater than X) after X All this should be done in linear time.

QUICK SORT

```
## QUICK SORT ##
printf("KHUSAL 1763")
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:-

```
KHUSAL 1763
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-11

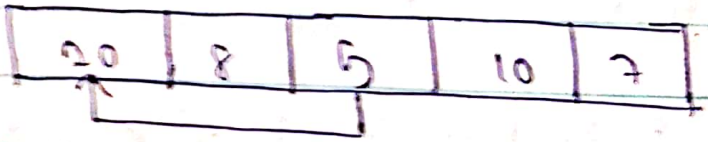
Aim:- To Sort given random data by using Selection Sort

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 beginning. The algorithm maintains two sub arrays in a given array. The sub array which is already sorted

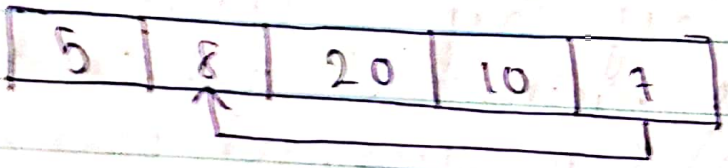
The Selection Sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the smallest value as it makes a pass and after completing the pass places it in the proper location. As with a bubble sort, after the first pass,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

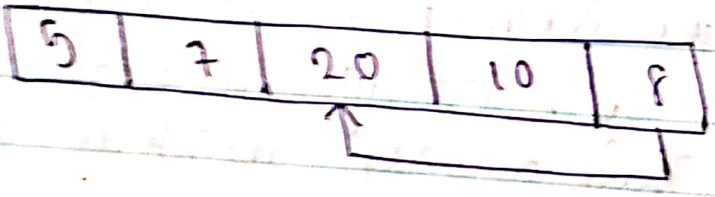
eg. In each pass the smallest remaining item is selected and then placed in its proper location.



5 is smallest

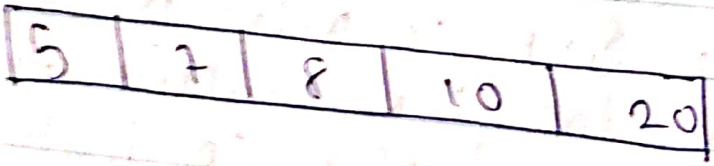


7 is smallest



8 is smallest

10 ok list is sorted



20 ok list is sorted

```
## SELECTION SORT ##
print("kushal 1763")
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)
```

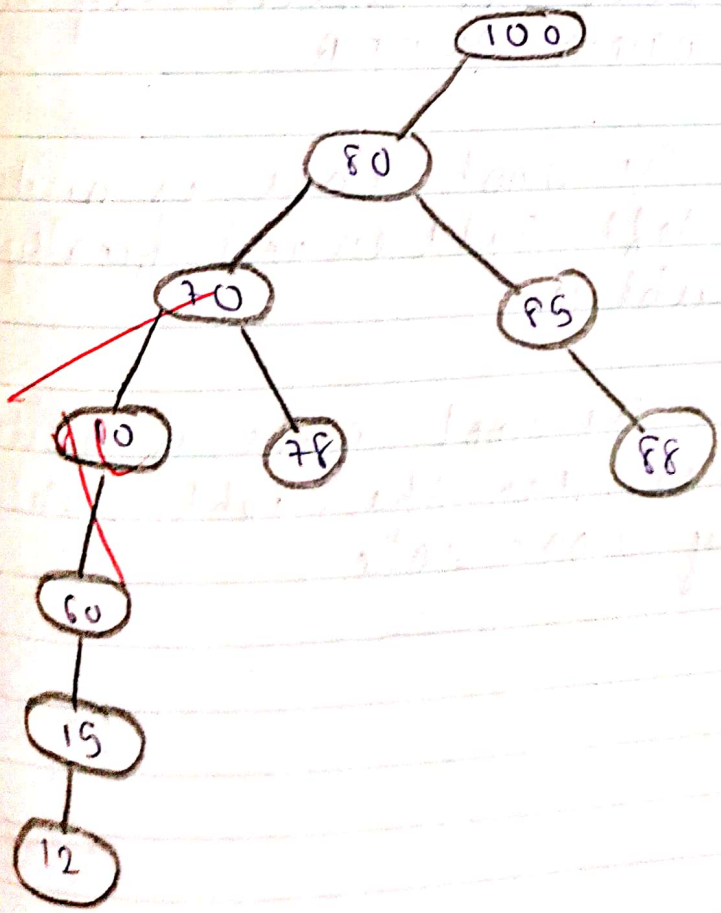
```
========== RESTART: C:/Users/Shree/Desktop/class/selection sort.py ==========
kushal 1763
Before sorting
 [23, 22, 18, 96, 56, 60]
After selection sort
 [18, 22, 23, 56, 60, 96]
>>>
```

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 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.

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

PRE-ORDER:- The root node is visited first then the left subtree and finally the right subtree.

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

```
## BINARY SEARCH TREE ##
class Node:
    print("kushal 1763")
    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(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 ==============
kushal 1763
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
>>>
```

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 [l...m] and arr [m+1...r] are Sorted and merges the two Sorted Sub-array into one.

```
print("kushal singh 1763")
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 =========
kushal singh 1763
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]
>>>
```