Practical no. 1

Aim: To implement linear search (unsorted) to find an item in a list.

Theory: The process of identifying or finding a particular record is called searching.
There are two types of search :-
(A) Linear Search
(B) Binary search.
The linear search is further classified as
(i) Sorted
(ii) unsorted.
Here we look on the unsorted linear search:
Linear Search, also known as sequential until the list sequentially until the desired element is found.
when the elements to be searched are not specifically arranged in ascending or descending order. They are arranged in random number order. This is what is called unsorted linear search.
In this any random order list can be used for searching.
This ~~is the simplest~~ form of search.

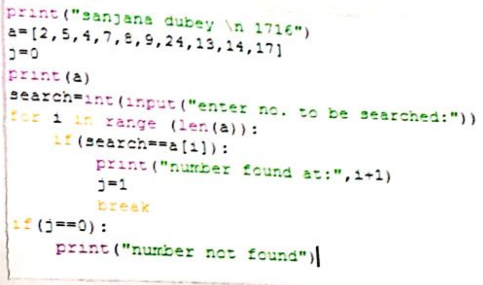
Advantage:
If the number is there in the list then you will find it.

Disadvantage:
If you are looking for the number which is at the end of the list then you need to search entire list.

Unsorted linear Search

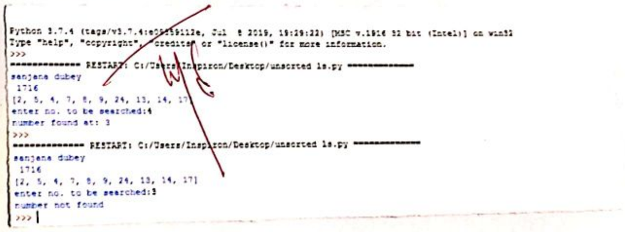
(A) The data is entered in random manner.
(B) user needs to Specify the element to be searched in the entered list.
(C) Check the condition that whether the entered list and the number matches. If it matches then display the location plus increment 1 as data is sorted from location zero.
(D) If all elements are checked one by one and element not found then prompt message number not found.

Source code:-



```
print("sanjana dubey \n 1716")
a=[2,5,4,7,8,9,24,13,14,17]
j=0
print(a)
search=int(input("enter no. to be searched:"))
for i in range (len(a)):
    if(search==a[i]):
        print("number found at:",i+1)
        j=1
        break
if(j==0):
    print("number not found")
```

Output:-



```
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: C:/Users/Inspiron/Desktop/unsorted ls.py =============
sanjana dubey
 1716
[2, 5, 4, 7, 8, 9, 24, 13, 14, 17]
enter no. to be searched:4
number found at: 3
>>>
============= RESTART: C:/Users/Inspiron/Desktop/unsorted ls.py =============
sanjana dubey
 1716
[2, 5, 4, 7, 8, 9, 24, 13, 14, 17]
enter no. to be searched:3
number not found
>>>
```

Practical no. 2.

Aim: TO implement linear search (sorted) to find an item in an list.

Theory :- Searching and sorting

These two are different modes or types of data structure.

Sorting: To basically sort the inputed data in ascending order or dexending order.

Searching: To search elements and to display the same.

In searching that too in linear search (sorted), the data is arranged in ascending to descending order or descending to ascending order.

That is all what is meant by searching through 'sorted' that is well arranged data.

Sorted linear Search

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

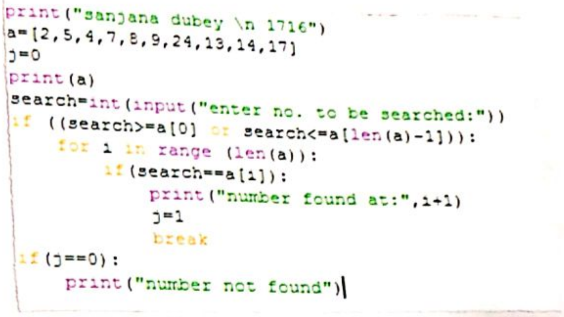
(B) user has to give an element for searching through sorted list.

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

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

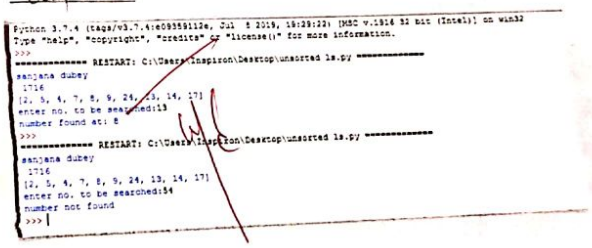
E) In sorted order list of elements we can check the condition that whether the entered number lies from starting point till the last element if not then without any processing we can say number not in the list.

Source code:-



```
print("sanjana dubey \n 1716")
a=[2,5,4,7,8,9,24,13,14,17]
j=0
print(a)
search=int(input("enter no. to be searched:"))
if  ((search>=a[0] or search<=a[len(a)-1])):
    for i in range (len(a)):
        if(search==a[i]):
            print("number found at:",i+1)
            j=1
            break
if(j==0):
    print("number not found")
```

output:-



```
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: C:\Users\Inspiron\Desktop\unsorted ls.py =============
sanjana dubey
 1716
[2, 5, 4, 7, 8, 9, 24, 13, 14, 17]
enter no. to be searched:13
number found at: 8
>>>
============= RESTART: C:\Users\Inspiron\Desktop\unsorted ls.py =============
sanjana dubey
 1716
[2, 5, 4, 7, 8, 9, 24, 13, 14, 17]
enter no. to be searched:54
number not found
>>> |
```

Practical no. 3

Aim: Implement binary search to find an item in an ordered list.

Theory: Binary search also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Advantage:-

If you are looking for the number which is at the end of the list then you need to search entire list in leanear search, which is time consuming. This can be avoided using binary search.

Binary Search

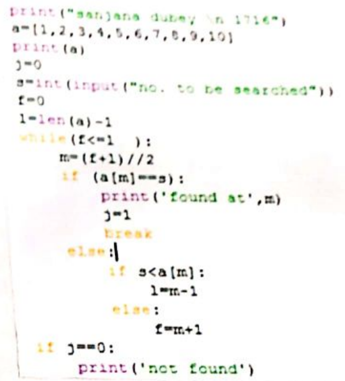
The user is supposed to enter the data in the list. Then enter the element to be searched from the list. Compare the entered data with the middle element. If the number matches the middle element, print number found at that position.

Else if the number is greater than the middle element then it can only lie in right half subarray after the middle element.

r else if the number is smaller then for the left half.

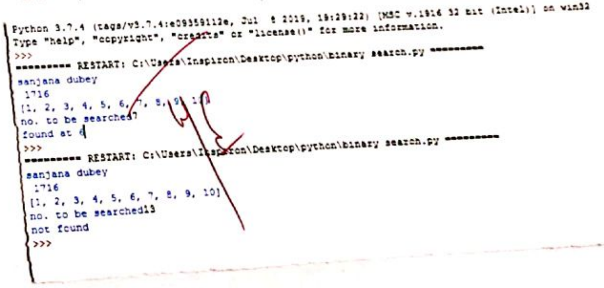
f it doesn't match then print number not found.

Source code:-



```
print("sanjana dubey \n 1716")
a=[1,2,3,4,5,6,7,8,9,10]
print(a)
j=0
s=int(input("no. to be searched"))
f=0
l=len(a)-1
while(f<=l  ):
    m=(f+l)//2
    if (a[m]==s):
        print('found at',m)
        j=1
        break
    else:
        if s<a[m]:
            l=m-1
        else:
            f=m+1
if j==0:
    print('not found')
```

Output:-



```
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: C:\Users\Inspiron\Desktop\python\binary search.py =========
sanjana dubey 
 1716
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
no. to be searched7
found at 6
>>> 
========= RESTART: C:\Users\Inspiron\Desktop\python\binary search.py =========
sanjana dubey 
 1716
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
no. to be searched13
not found
>>> 
```

Practical no. 4.

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. Is a Simple Sorting algorithm that repeatedly Steps through the lists, compares adjacent elements and Swaps them if they are in wrong order.

The pass through the list is repeated until the lists is Sorted.

The algorithm which is a comparision Sort is named for the way Smaller or larger elements "bubble" to the top of the list.

Although the algorithm is Simple, it is too Slow as compared to one element checks if condition fails then only swaps otherwise goes on.

Examples:-

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) Swaps since 5>4

(1 4 5 2 8) → (1 4 2 5 8) Swaps 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)

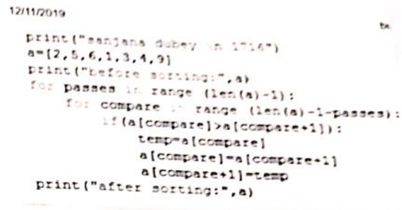
(1 4 2 5 8) → (1 2 4 5 8) Swap since 4> 2

(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.

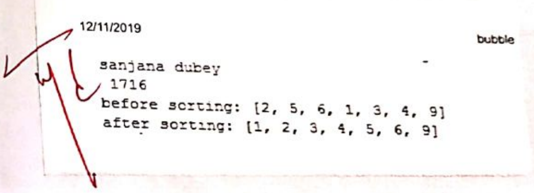
Source code:-



12/11/2019

```
print("sanjana dubey \n 1716")
a=[2,5,6,1,3,4,9]
print("before sorting:",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 sorting:",a)
```

Output:-



12/11/2019

bubble

```
sanjana dubey
 1716
before sorting: [2, 5, 6, 1, 3, 4, 9]
after sorting: [1, 2, 3, 4, 5, 6, 9]
```

Source code:

```
print("STACK\nSA
class stack:
        global t
        def __in
                sel
                sel
        def push

                n=l
                if

                els
```

Output:

```
STACK
SANJANA DUBEY
1716
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty
>>> |
```

Aim: Implement working of Stack.

Theory: In computer science, a stack is an abstract data type that serves as a collection of elements with two principal operations push, which adds an element to the collection and pop, which removes the most recently added element that was not yet removed.

The order may be LIFO (Last in first out) or FILO (First in Last out)

3 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 reversed order in which they are pushed. If the stack is empty, then it is said to be an underflow condition.

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

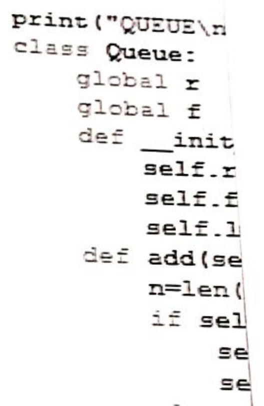
4. Is empty: Returns true if stack is empty else false.

DAO: Practical no. 6.

Aim: Implement the queue as a list.

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 at both of its ends.
Enqueue () can be termed as add () in queue i.e adding a element in queue.
Dequeue () can be termed as delete or Remove. i.e deleting or removing of element.
Front is used to get the front data item from a queue.
Rear is used to get the last item from a queue.

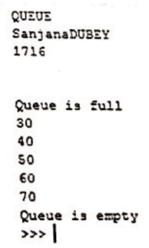
Source code:



```
print("QUEUE\n
class Queue:
    global r
    global f
    def __init
        self.r
        self.f
        self.l
    def add(se
        n=len(
        if sel
            se
            se
```

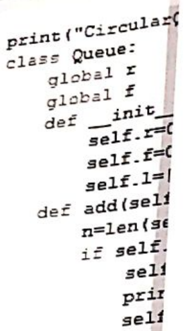
4E

Output:



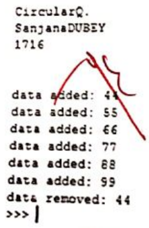
```
QUEUE
SanjanaDUBEY
1716


Queue is full
30
40
50
60
70
Queue is empty
>>> |
```

Source code:



```
print("Circular(
class Queue:
    global r
    global f
    def __init__
        self.r=(
        self.f=(
        self.l=[
   def add(self
       n=len(se
       if self.
           self
           prir
           self
```

output:



```
CircularQ.
SanjanaDUBEY
1716

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: Implement a circular queue.

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 actuality there might be empty slots at the beginning of the queue.

To overcome this limitation we can implement queue as circular queue.

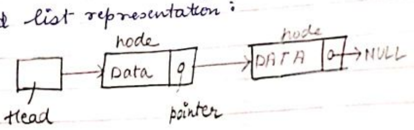
In circular queue we go on adding the element to the queue and reach the end of the array. The next element is sorted in the first slot of the array.

P.P.O Practical no. 8.

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 containg items each link contains a
1) Link - Each link of a linked list can store a data called an element.
2) Next - Each link of a linked list contains a link to the next link called NEXT.
3) Linked list - A linked list contains the connection link to the first link called first.

Linked list representation:



Types of linked list:

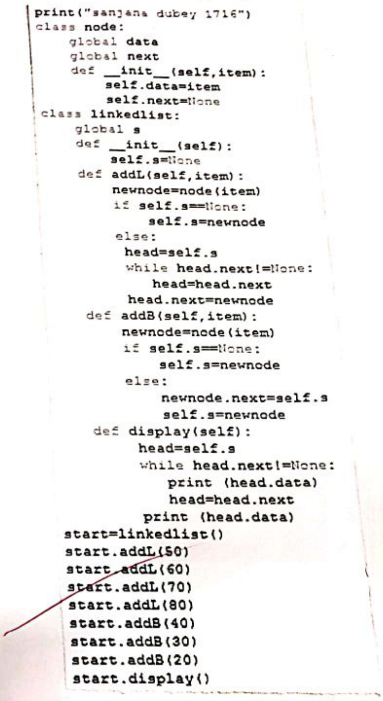
1. Simple
2. Doubly
3. circuler

Basic operations:

1. Insertion
2. Deletion
3. Display
4. Search
5. delete.

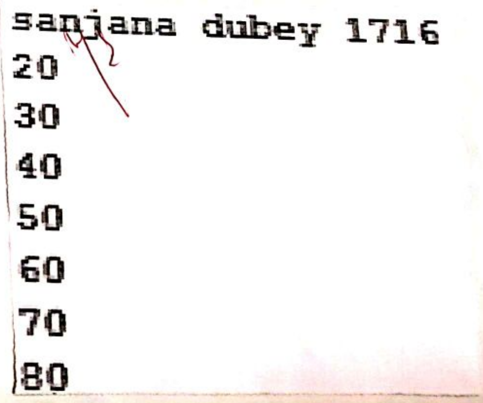
Source code:-

050



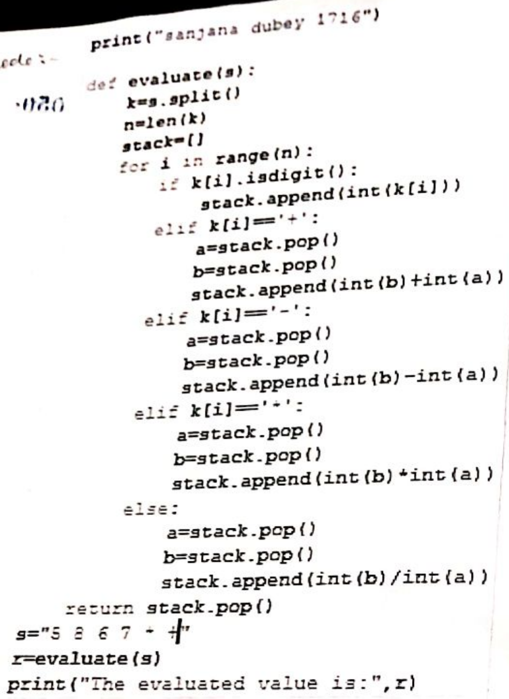
```
print("sanjana dubey 1716")
class node:
    global data
    global next
    def __init__(self,item):
        self.data=item
        self.next=None
class linkedlist:
    global s
    def __init__(self):
        self.s=None
    def addL(self,item):
        newnode=node(item)
        if self.s==None:
            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:-



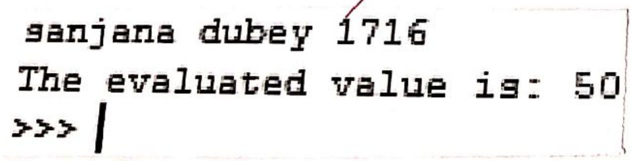
```
sanjana dubey 1716
20
30
40
50
60
70
80
```

Source code:-



```
print("sanjana dubey 1716")
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(b)*int(a))
        else:
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)/int(a))
    return stack.pop()
s="5 8 6 7 * +"
r=evaluate(s)
print("The evaluated value is:",r)
```

Output:-



```
sanjana dubey 1716
The evaluated value is: 50
>>> |
```

Practical no. 9.

Aim:- To evaluate postfix expressions using stack.

Theory: Stack is an (ADT) and works on LIFO (Last in first out) i.e PUSH and POP operations.

A postfix expression is a collections of operators and operands in which the operator is placed after the operands.

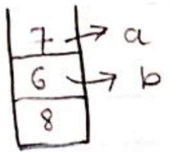
Steps to be followed:-

1. Read all the symbols one by one from left to right in the given postfix expressions.
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 operations and store the two popped operands in two different variables (operand 1 and operand 2). Then perform reading symbol operator using operand 1 and 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:-

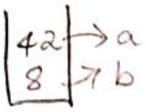
" 8 6 7 * + "

Stack :-

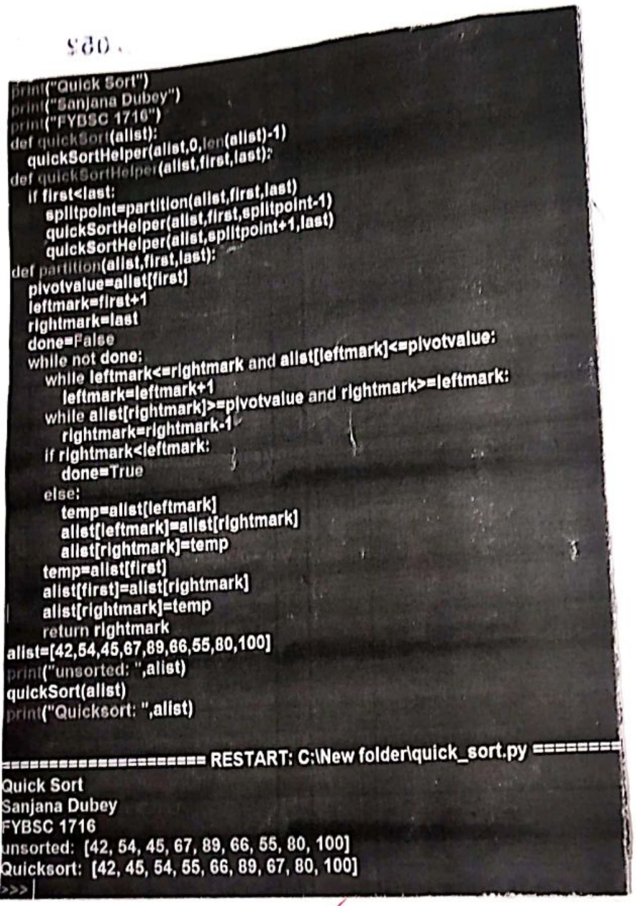


a *b = 7×6 : 42

Store 42 in stock.



a+b = 42+8 = $\underline{\underline{50}}$



```
print("Quick Sort")
print("Sanjana Dubey")
print("FYBSC 1716")
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("unsorted: ",alist)
quickSort(alist)
print("Quicksort: ",alist)

===================== RESTART: C:\New folder\quick_sort.py =========
Quick Sort
Sanjana Dubey
FYBSC 1716
unsorted:  [42, 54, 45, 67, 89, 66, 55, 80, 100]
Quicksort:  [42, 45, 54, 55, 66, 89, 67, 80, 100]
>>>
```

Practicalno. 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 medias 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, put x at its correct position is sorted array and put all smaller elements (smaller than x) before x and put all greater elements (greater than x) after x. All this should be done in linear time.

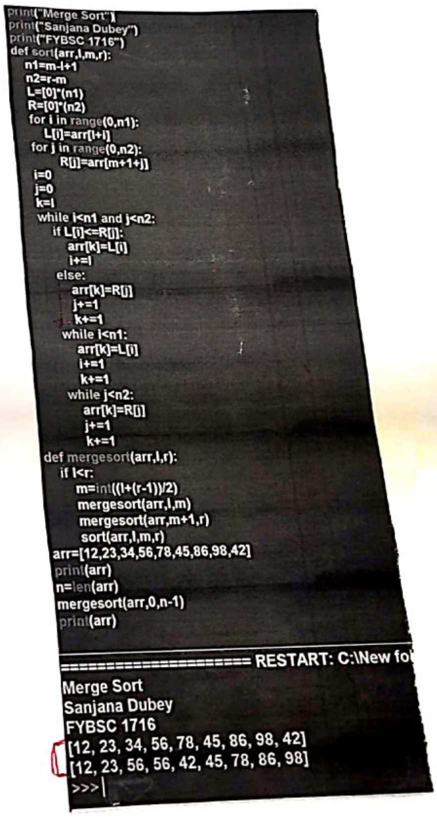
Practical no: 11

Aim: Merge Sort.

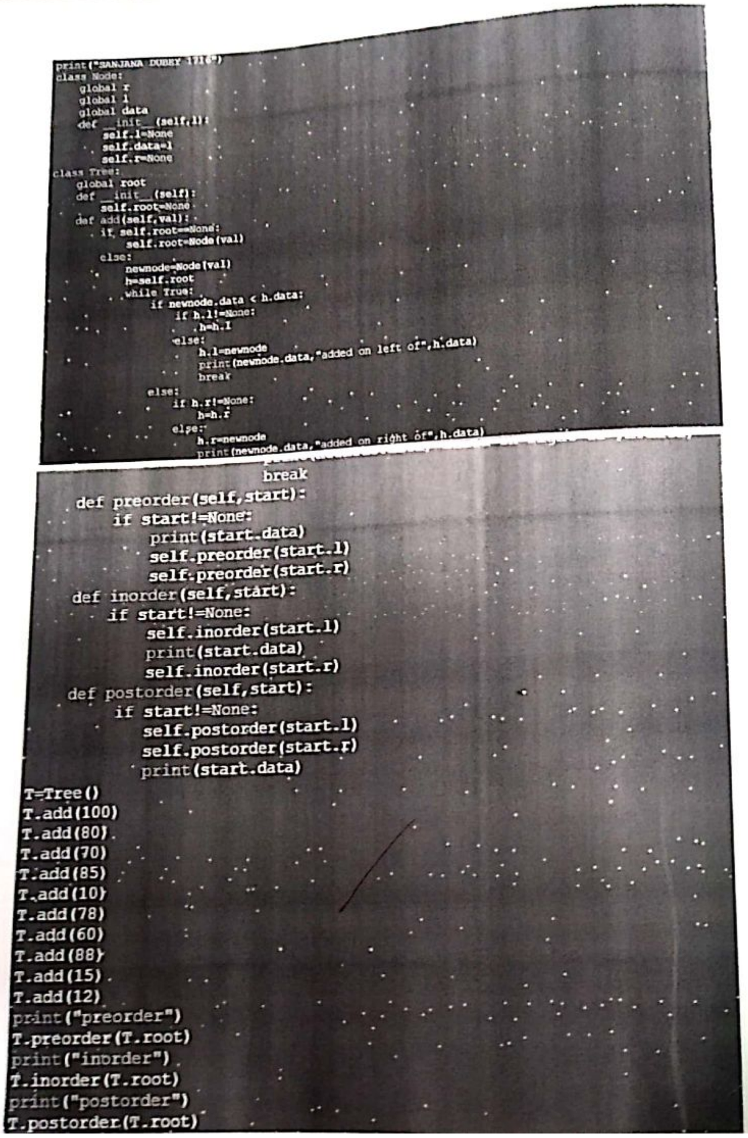
Theory: conceptually a merge sort looks as
1. Divide the unsorted list into n
Sublists; each containing one element.
A list of one element is inherely Sorted.
2. Repeadly merge sort sublists to produce
longer runs until there is only one run
remaining. This is the sorted list.

The patterns of merge form tree Structure
in the common case that binary (two-way)
merge is used, forming binary tree.
The merge must be performed accordingl
to a post order traversal, but that
still leaves considerable flexibility.
Most common choices are depth - first or
breadth first order, but practical
implementation on tend to use depth -
first Order, as that gives better locality
of reference.

It is also possible to interface the merges,
generating the output of a smaller merge
as needed by a large merge.
Taken to extreme, the result resembles
heap Sort.



```
print("Merge Sort")
print("Sanjana Dubey")
print("FYBSC 1716")
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=[12,23,34,56,78,45,86,98,42]
print(arr)
n=len(arr)
mergesort(arr,0,n-1)
print(arr)

======================= RESTART: C:\New fo

Merge Sort
Sanjana Dubey
FYBSC 1716
[12, 23, 34, 56, 78, 45, 86, 98, 42]
[12, 23, 56, 56, 42, 45, 78, 86, 98]
>>>
```

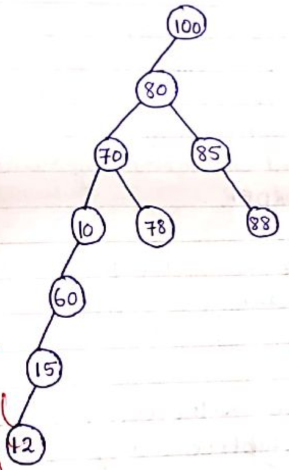


```
print("SANJANA DUBEY 1716")
class Node:
    global r
    global l
    global data
    def __init__(self,l):
        self.l=None
        self.data=l
        self.r=None
class Tree:
    global root
    def __init__(self):
        self.root=None
    def add(self,val):
        if self.root==None:
            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)
            self.postorder(start.r)
            print(start.data)
T=Tree()
T.add(100)
T.add(80)
T.add(70)
T.add(85)
T.add(10)
T.add(78)
T.add(60)
T.add(88)
T.add(15)
T.add(12)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
```

Practical no. 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 atmost two children.



Diagramatic Representation of Binary Search Tree.

Traversal is a process to visit all the node of a tree and may print their value too.

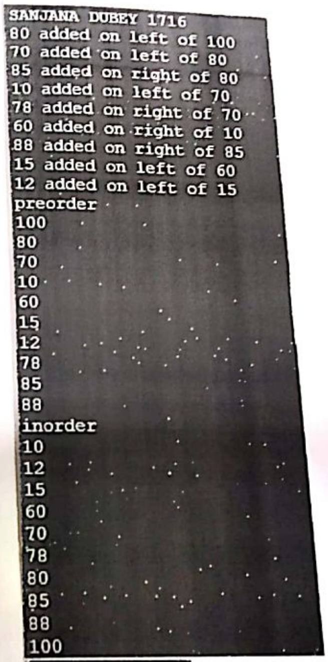
There are 3 ways which we use to travere a tree.

→ INORDER
→ PREORDER
→ POST ORDER

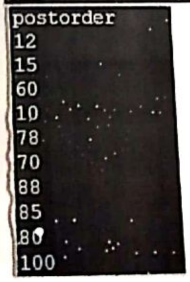
INORDER: The left-Subtree is visited 1st then the root and later the right Subtree. we should always remember that every node may represent a subtree itself. output produced is sorted key values in ASCENDING ORDER.

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

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



```
SANJANA DUBEY 1716
80 added on left of 100
70 added on left of 80
85 added on right of 80
10 added on left of 70
78 added on right of 70
60 added on right of 10
88 added on right of 85
15 added on left of 60
12 added on left of 15
preorder
100
80
70
10
60
15
12
78
85
88
inorder
10
12
15
60
70
78
80
85
88
100
```



```
postorder
12
15
60
10
78
70
88
85
80
100
```