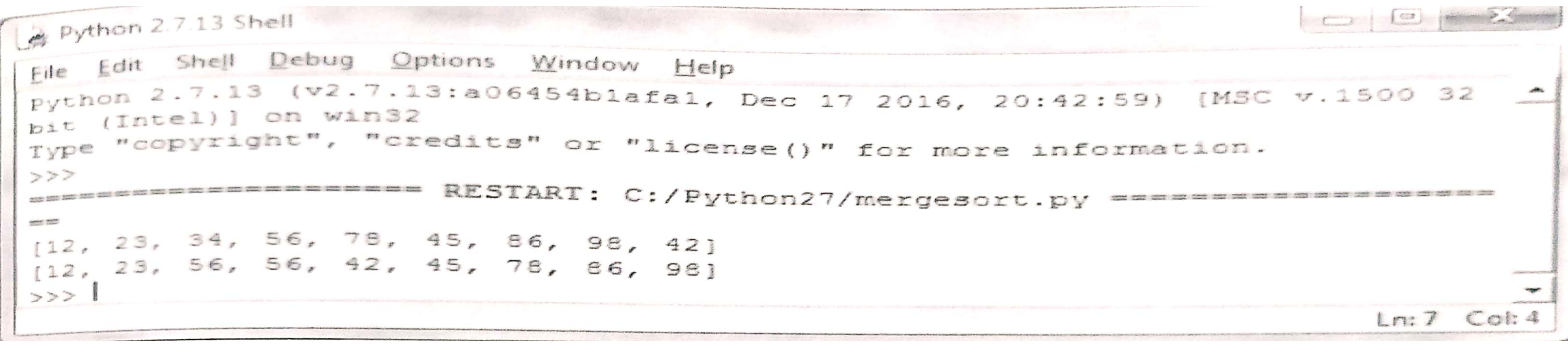


Input -

```
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[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)
```

PRACTICAL - 12

Aim:- Merge Sort

Theory:- Merge sort is a sorting technique based on divide and conquer technique with worst-case time complexity being $O(n \log n)$. It is one of the most respected algorithms.

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-arrays into one.

```
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
print("postorder")
T.postorder(T.root)
```

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
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 ==========
>>>
60 added on left of 100
70 added on left of 60
65 added on right of 60
10 added on left of 70
75 added on right of 70
80 added on right of 10
88 added on right of 65
15 added on left of 80
12 added on left of 15
preorder
100
60
70
10
80
15
12
75
65
88
inorder
10
12
15
80
70
75
60
88
65
100
postorder
10
12
15
80
70
75
60
65
88
100
```

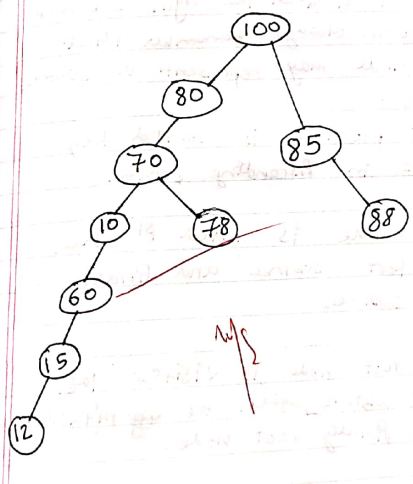
```
## Binary Tree and Traversal ##

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):
ifself.root==None:
self.root=Node(val)
else:
newnode=Node(val)
        h=self.root
while True:
ifnewnode.data<h.data:
ifh.l!=None:
                h=h.l
else:
h.l=newnode
print(newnode.data,"added on left of",h.data)
break
else:
h.r=newnode
print(newnode.data,"added on right of",h.data)
break
defpreorder(self,start):
if start!=None:
print(start.data)
self.preorder(start.l)
self.preorder(start.r)
definorder(self,start):
if start!=None:
self.inorder(start.l)
print(start.data)
self.inorder(start.r)
defpostorder(self,start):
if start!=None:
self.inorder(start.l)
self.inorder(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)
```


PRACTICAL - 11

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 search Tree.

PRACTICAL-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 & 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

The key process in quicksort is partition(). Target of partitions is, 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.

Input →

```
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)
```

Output →

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
[42, 45, 54, 55, 66, 89, 67, 80, 100]
>>>
```

SOURCE CODE:

```
print("Vedant Sharma 1710")
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))
returnstack.pop()
s="4 8 5 2 / * +"
r=evaluate(s)
print("The evaluated value is:",r)
```

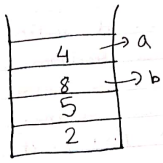
OUTPUT:

Vedant Sharma 1710
The evaluated value is: 12

Value is of Postfix expression:-

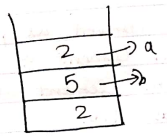
S = 2 5 8 4 / * +.

Step (1).



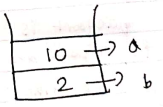
∴ b/a = 8/4 = 2.

Step (ii).



b * a = 5 * 2 = 10.

Step (iii).



b + a = 2 + 10 = 12.

SOURCE CODE:

```
print("Vedant Sharma 1710")
class node:
global data
global next
def __init__(self,item):
self.data=item
self.next=None
classlinkedlist:
global s
def __init__(self):
self.s=None
defaddl (self,item):
newnode=node(item)
ifself.s==None:
self.s=newnode
else:
head=self.s
whilehead.next!=None:
head=head.next
head.next=newnode
defaddlB(self,item):
newnode=node(item)
ifself.s==None:
self.s=newnode
else:
newnode.next=self.s
self.s=newnode
def display(self):
head=self.s
whilehead.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:

```
Vedant Sharma
20
30
40
```

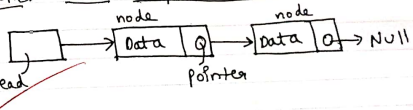

PRACTICAL-8

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 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 :-



```
            print("Data removed:",self.l[self.f])
            self.f=self.f+1
        else:
            print("queue is empty")
            self.f=s
q=queue()
q.add(44)
q.add(55)
q.add(66)
q.add(77)
q.add(88)
q.add(99)
q.remove()
```

OUTPUT:

VEDANT SHARMA

1710

data added: 44

data added: 55

data added: 66

data added: 77

data added: 88

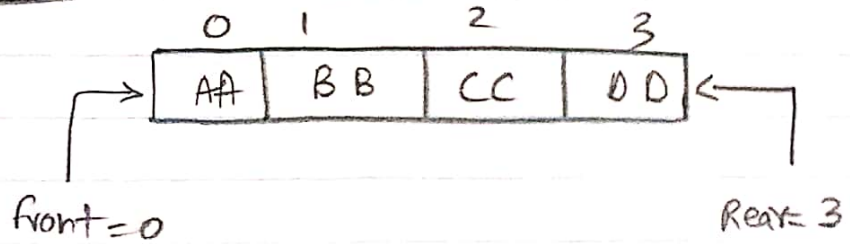
data added: 99

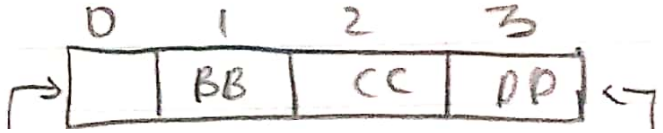
Data removed: 44

THEORY:- The queue that we implement using an array suffer from one limitation. In that implementation there is a ~~particular~~ possiblity 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 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. ~~In~~ the next element is stored in the first slot of the array.

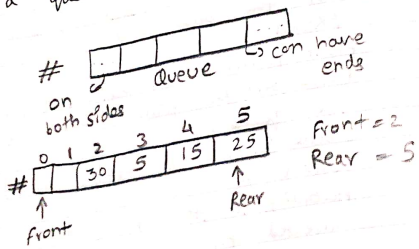
Example:-





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

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



on both sides

Front = 2
Rear = 5

```
q.add(30)
q.add(40)
q.add(50)
q.add(60)
q.add(70)
q.add(80)
q.remove()
q.remove()
q.remove()
q.remove()
q.remove()
q.remove()
```

OUTPUT:

```
VEDANT SHARMA
1710
queue is full
queue is full
30
40
50
60
queue is empty
queue is empty
```

SOURCE CODE:

```
print("VEDANT SHARMA \n1710")
class queue:
  global r
  global f
  def __init__(self):
    self.r=0
    self.f=0
    self.l=[0,0,0,0,0]
  def add(self,data):
    n=len(self.l)
    if(self.r<n-1):
      self.l[self.r]=data
      self.r=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()
```


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 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.

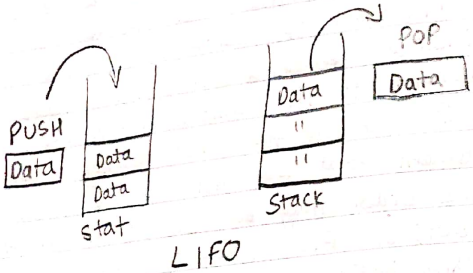
According to its FIFO structure, element inserted first will also be removed first.

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.

- Peek or Top : Returns top element of stack.
- Is empty : Returns true if stack is empty else false.



LIFO

```
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:

```
VEDANT SHARMA
1710
Stack is full!
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
Stack is empty
```

```
print("VEDANT SHARMA \n1730")
class stack
  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 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)
```

PRACTICAL-5

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 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: Adds an item in the stack, 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 reverse order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

PRACTICAL - 4

Aim: To sort given random data by

THEORY: Sorting is type in which any is sorted i.e. arranged in or descending order.

Bubble sort sometimes referred to as sinking sort.

It is a simple algorithm that repeatedly steps through the lists, compares adjacent elements and swaps them if they are in wrong order.

They pass through the list repeatedly until the list is sorted. the algorithm which is a comparis sort is named for the way smaller or large elements to the top of the list.

Although the algorithm is simple, it is too as it compares one element checks if cond is ~~false~~ failed then only swaps, otherwi on.

For eg, for (17, 13, 22, 1, 7, 32) compare the first element other element.

```
print("VEDANT SHARMA\n1710")
a=[3,4,21,24,25,29,64]
print(a)
search=int(input("Enter number:"))
l=0
h=len(a)-1
m=int((l+h)/2)
if((search<a[l])or(search>a[h])):
    print("Number not in range")
elif(search==a[h]):
    print("Number found at location:",h+1)
elif(search==a[l]):
    print("Number found at location:",l+1)
else:
    while(l!=h):
        if(search==a[m]):
            print("Number found at location:",m+1)
            break
        else:
            if(search<a[m]):
                h=m
                m=int((l+h)/2)
            else:
                l=m
                m=int((l+h)/2)
    if(search!=a[m]):
        print("Number not in given list!")
        break
```

```
OUTPUT
CASE1:
VEDANT SHARMA
1710
[3,4,21,24,25,29,64]
Enter number:24
Number found at location:5
CASE2:
VEDANT SHARMA
1710
Enter number:90
Number not in range
CASE3
VEDANT SHARMA
1710
Enter number:15
Number not in given list!
```


PRACTICAL -3.

Prm° To find an item from list by binary search.

THEORY°

Binary search finds the position of target value of sorted array. The binary search can be classified as dichotomies divide and conquer search algorithm and executes in logarithmic time.

The idea of binary search is to use the info. is that the array is sorted and reduce the time complexity to O(logn). We basically ignore half of elements just after one comparison.
Compare 'x' with the middle element. If 'x' matches with middle element, we return the mid index.
Else if 'x' is greater than mid element, then 'x' can only lie in right half subarray after the mid element. So we recur for right half.

Else recur for left half.
Its time complexity can be written as:
$T(n) = T(n/2) + C$.
Auxillary space is O(1), In case of iterative implemention,
O(logn) recursion call stack space.

PRACTICAL -3

Aim: To find an item from list by binary search.

THEORY:

Binary search finds the position of target value of sorted array. The binary search can be classified as dichotomies divide and conquer search algorithm and executes in logarithmic time.

The idea of binary search is to use the info. is that the array is sorted and reduce the time complexity to O(log n). We basically ignore half of elements just after one comparison.
Compare 'x' with the middle element. If 'x' matches with middle element, we return the mid index.
Else if 'x' is greater than mid element, then 'x' can only lie in right half subarray after the mid element. So we recur for right half.

Else recur for left half.
Its time complexity can be written as:
$T(n) = T(n/2) + C$.
Auxiliary space is O(1), In case of iterative implementation,
O(log n) recursion call stack space.

```
print("VEDANT SHARMA\n1710")
a=[3,4,21,24,25,29,64]
print(a)
search=int(input("Enter number:"))
l=0
h=len(a)-1
m=int((l+h)/2)
if(search<a[l]or(search>a[h])):
    print("Number not in range")
elif(search==a[h]):
    print("Number found at location:",h+1)
elif(search==a[l]):
    print("Number found at location:",l+1)
else:
    while(l!=h):
        if(search==a[m]):
            print("Number found at location:",m+1)
            break
        else:
            if(search<a[m]):
                h=m
                m=int((l+h)/2)
            else:
                l=m
                m=int((l+h)/2)
    if(search!=a[m]):
        print("Number not in given list!")
        break
```

```
OUTPUT
CASE1:
VEDANT SHARMA
1710
[3,4,21,24,25,29,64]
Enter number:24
Number found at location:5
CASE2:
VEDANT SHARMA
1710
Enter number:90
Number not in range
CASE3
VEDANT SHARMA
1710
Enter number:15
Number not in given list!
```

Time complexity is O(n). This is because in worst case we need to scan the complete array. But in average case it reduces the complexity even though the growth rate is some.
Its space complexity is O(1).

```
j=0
print(a)
search=int(input("Enter the number to be searched: "))
if((search<a[0]) or (search>a[6])):
print("Number doesnt exist!")
else:
for i in range (len(a)):
if(search==a[i]):
print("Number found at: ",(i+1)
    j=1
break
if(j==0):
print("Number NOT found!")
```

output:

VEDANT SHARMA

1710

[3, 10, 21, 59, 69, 74, 80]

Enter the number to be searched: 80

Number found at: 7

VEDANT SHARMA

1710

[3, 10, 21, 59, 69, 74, 80]

Enter the number to be searched: 5

Number NOT found!

PRACTICAL - 2

Aim: To find an item from list by sorted linear search.

Theory: Linear search is a method for finding a particular value in a list. It starts searching the value from the beginning of the list and continues till the end of the list until the value is found.

If the elements of the array are already sorted then in many cases we don't have to scan the complete array to see if the element is there in the given array or not.

In the algorithm, it can be seen that, at any point if the value at arr[i] is greater than the data to be searched then we just return -1 without searching the remaining array.

It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of element in the array, else it ret. -1.

Expressions like while (n) (n--;) also contribute to the count of comparisons as value of n being compared internally so as to decide whether or not to terminate the loop. It has a time complexity of O(n) and has very simple implementation.

Expressions like while (n) (n--;) also contribute to the count of comparisons as value of n is being compared internally so as to decide whether or not to terminate the loop. It has a time complexity of O(n) and has very simple implementation.

PRACTICAL-1

m: To find an item from list by unsorted linear search.

Theory: Linear search is very basic and simple search algorithm. In this, we search an element or value in a given array by traversing the array from the starting, till desired element or value is found.

Let us assume that given an array whose elements of the array ~~are~~ ~~is not in~~ order is known. That means elements are not sorted. In this case if we want to search for an element then we have to scan the complete array and see if the element is there in the given list. or not. This is called sorted linear search.
This requires to scan the array completely and check each element for the array that we need to search.

Given an array of n distant integers and an element x. Search the element x in array using minimum number of comparisons. Any sort of comparison ~~will~~ will contribute 1 to count of comparisons. For eg, the condition used to terminate a loop, will also contribute 1 to the cont of comparisons each time it gets executed.