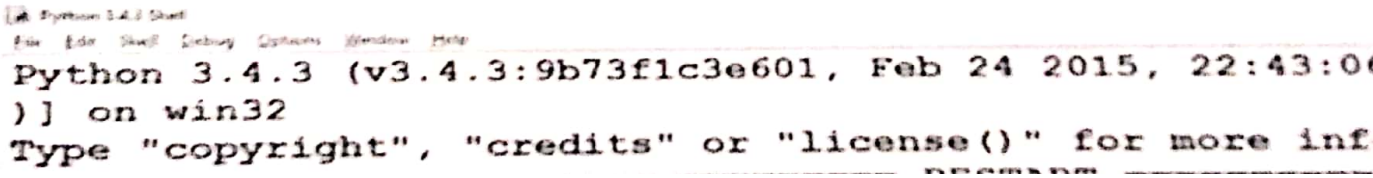
#Linear Search Unsorted

SOURCE CODE:

```
print("abhay singh")
print("Roll No.1750")
found=False
a=[21,31,41,51,61,71,81,91]
print(a)
search=int(input("Enter the number to be searched: "))
for i in range(len(a)):
  if(search==a[i]):
        print("The number is found at" ,i+1)
        found=True
       break;
if(found==False):
        print("The number is not found")
```

OUTPUT:



Practical :

Aim :- 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

i) Linear search

ii) Binary search

The linear search is further classified as unsorted. Here we will look of the UNSORTED linear search, also known as Sequential search, also is a process that checks every element in the list sequentially until the desired element is found.

When the element we searched are not specified arranged in ascending or arranged order. They are arranged in random that is what it calls unsorted linear search

UNSORTED linear search.

- The data is entered in random manner.
- User needs specified the element to the 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.

- If all element are checked one by one and not found then prompt message number not f

#linear search sorted

```
found=False
a=[12,45,66,86,98]
print("Abhay singh")
print("Roll No. 1750")
search=int(input("Enter a Number to be searched : "))
if(search<a[0] or search>a[len(a)-1]):
    print("Number does not exists ")
else:
    for x in range(len(a)):
        if(search==a[x]):
            print("The number is found at",x,"index number")
            found=True
            break
    if(found==False):
        print("Number not in the list")
```

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (
Type "copyright", "credits" or "license()" for more information.
```

Practical 2.

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

Theory: Searching and sorting are different modes are type of data. Structure

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

SEARCHING - To search element and to display the same

In searching that too in Linear sorted se

The data is arranged in ascending and descen that is all what it mean by searching through 'so that is well arranged data.

SORTED linear search.

The user is supposed to enter data in sorted man User has to given an element for searching through sort If element is found display with an updation as val is stored from location 0.

If data or element not found print the same.

Practical 3

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

Theory: A linear search also know as a half interval search, is an algorithm used in computer science to locate. Jos the search to the linear, the array must list sorted in either ascending or descending order.

At each step of the algorithm a comparision node 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 know which half of the array to continues searching in he the array is sorted.

This process is represented on progressive smaller segments of the array. Until the value is located.

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

#BINARY SEARCH:

```
print("Abhay singh")
print("Roll No.1750")
a=[12,25,89,57,99]
print(a)
search=int(input("Enter a number to be searched : "))
l=0
r=len(a)-1
while(True):
  m=(l+r)//2
  if(l>r):
    print("Number is not found")
    break
  if(search==a[m]):
    print("Number found at:",m+1)
    break
  elif(search<a[m]):
    r=m-1
  else:
    l=m+1
```

```
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
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Abhay singh
Roll No.1750
[12, 25, 89, 57, 99]
Enter a number to be searched : 99
Number found at: 5
>>> ========== RESTART ==========
>>>
Abhay singh
Roll No.1750
[12, 25, 89, 57, 99]
Enter a number to be searched : 66
Number is not found
>>>
```

Aim : To demonstrate the use of ~~stak~~ stack.

Theory:
In computer science, a stack is an abstract data that serves as a collection of elements with two principal operation push, which adds an element to the collection and pop, which removes the most recently added element that was not yet remmoved. The order may be LIFO (Last in First out) or FILO (First In Last out).

Three Basic Operation are performed in the ...

- Push : Adds an item in the if the stack full th... is said to be over flow condition.

- Pop : Removes an item from the stack the items a... Popped in the reversed order in which they are pu... Pushed. If the stack is empty, then it is said to be a underflow Condition.

- Peek or top : Returns top element of stack.

- Is empty : Returns true if stack is empty else false

Push
Data ~~Data~~
Data
Stack

Data
"
"

Pop
data

Last_in first_out

```
print("Abhaysingh 1750")
class stack:
globaltos
def __init__(self):
self.l=[0,0,0,0,0,0,0,0]
self.tos=-1
def push(self,data):
    n=len(self.l)
ifself.tos==n-1:
print("stack is full")
else:
self.tos=self.tos+1
self.l[self.tos]=data
def pop(self):
ifself.tos<0:
print("stak 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)
```

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

```
>>>
====================
Abhay singh 1750
queue is full
3
40
50
60
70
queue is full
>>>
```

```
print("Abhaysingh 1750")
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)
ifself.r<n-1:
self.l[self.r]=data
self.r=se!f.r+1
else:
print("queue is full")
def remove(self):
    n=len(self.l)
ifself.f<n-1:
print(self.l[self.f])
self.f=self.f+1
else:
print("queue is full")
q=queue()
q.add(3)
q.add(40)
```

s.pop()

s.pop()

s.pop()

s.pop()

s.pop()

s.pop()

s.pop()

s.pop()

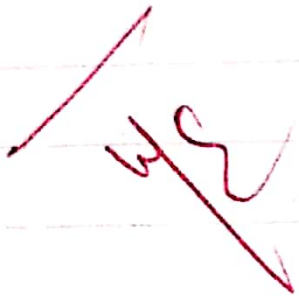
```
Abhay singh 1750
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack is empty
>>>
```

In a queue, one end is always
(deque) because queue is open on 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

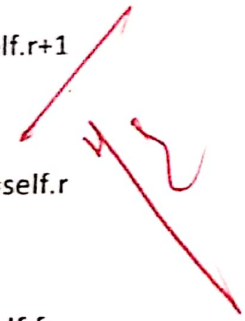
Front is use to get the front data item from a queue
Rear is used to get the last item from a queue



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

Theory: The queue that use 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 be empty slots at the ~~beng~~ beging 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 stored in the first slot of the array

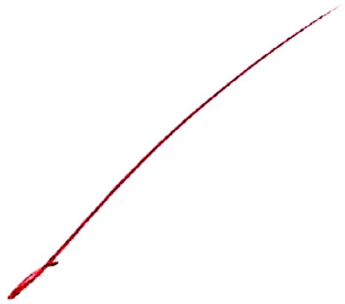
```
print("Abhaysingh 1750")
class queue:
global r
global f
def __init__(self):
self.r=0
self.f=0
self.l=[0,0,0,0,0,0]
def add(self,data):
    n=len(self.i)
ifself.r<n-1:
self.l[self.r]=data
print("data added",data)
self.r=self.r+1
else:
        s=self.r
self.r=0
ifself.r<self.f:
self.l[self.r]=data
self.r=self.r+1
else:
print("queue is full")
def remove(self):
    n=len(self.l)
ifself.f<n-1:
```

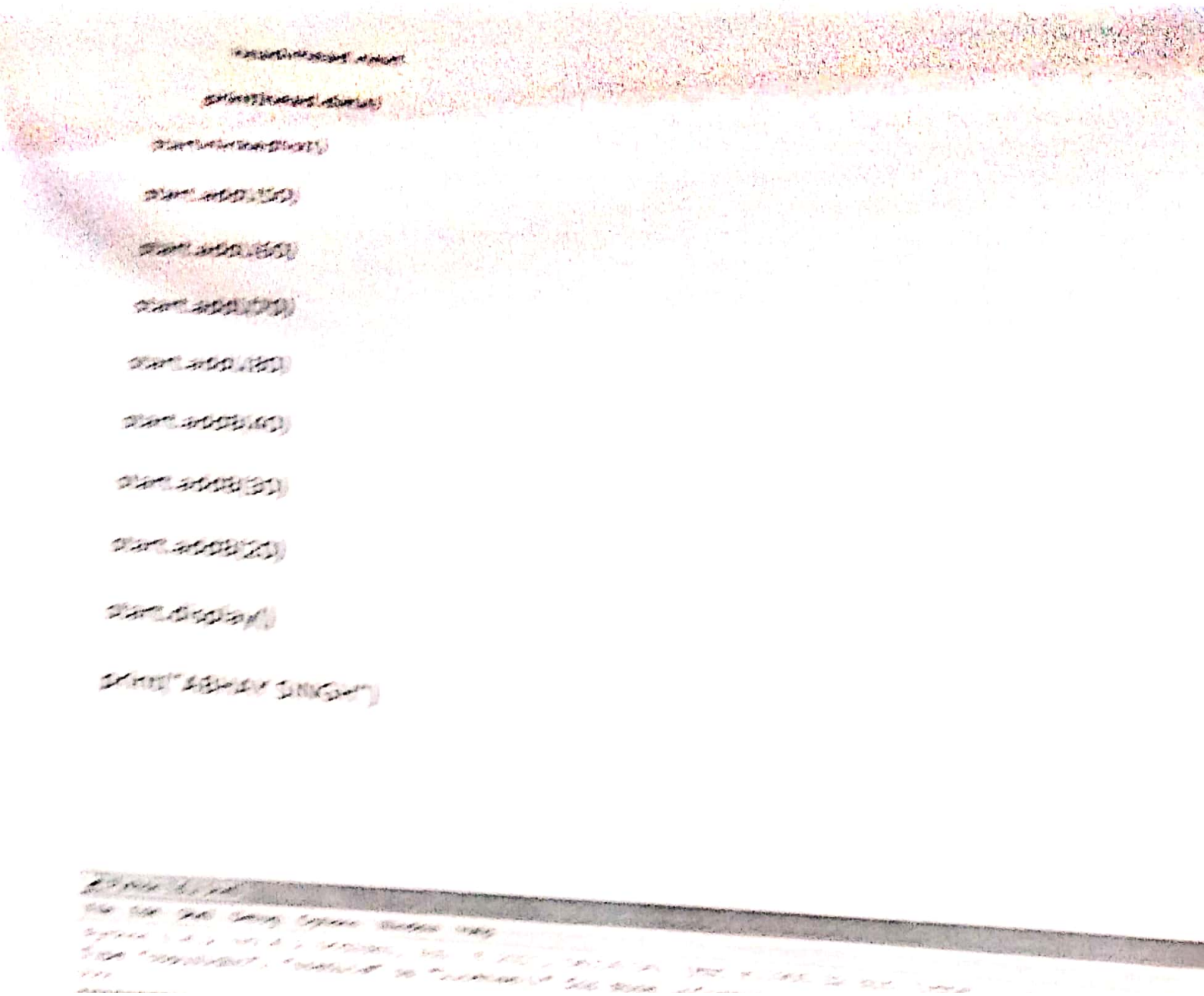


```
        s=self.f
        self.f=0
        ifself.f<self.r:
        print(self.l[self.f])
        self.f=self.f+1
        else:
        print("queue is empty")
        self.f=s
q=queue()
q.add(33)
q.add(44)
q.add(55)
q.add(66)
q.add(77)
q.add(88)
q.remove()
q.add(66)
```

```
>>>
======================
Abhay singh 1750
data added 33
data added 44
data added 55
data added 66
data added 77
queue is full
data removed: 33
data added 66
>>>
```

```
### After before linkedlist(simple)###
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)
```





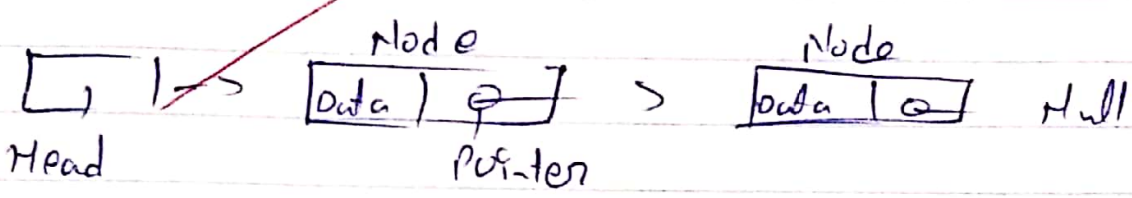
Practical 7

Aim: To demonstrate the use of Linkedlist in data structure.

Theory: A Linked list is a sequence of data structure 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 contain a link the no. link called next.
- Linked - A linked list contains the connection link t the first link called first.

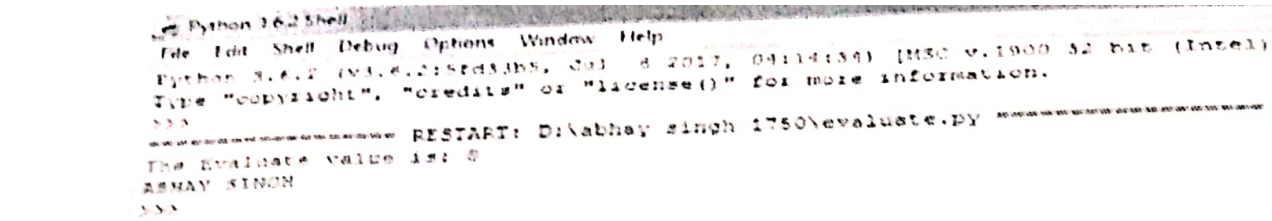
Linked List Representation -



Types of Linked List :

-> Simple
-> Doubly
-> Circular.

```
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=stcak.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="8 6 9 * +"
```



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 operation.

A Postfix expression is a collection of ~~operation~~ operator and operands in which the operator is placed after the operands.

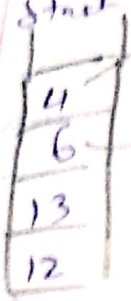
Step to be followed:

1. Read all the symbols one by one from left to right in the given Postfix expression.
2. If the reading symbol is operand the push it on the stack.
3. If the reading symbol is operator (+, -, *, /, etc). then perform Two Pop operation and store the two Popped operand in two different variable (operand 1 & operand 2 perform reading symbol operator using operand 1 & operand 2 & Push result back onto the stack.
4. Finally! Perform a Pop operation and display the Pop value as final result

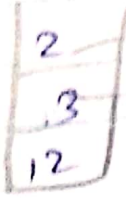
Value of postfix expression

S = 12 3 6 4 − + 3

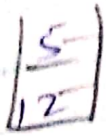
Stack:



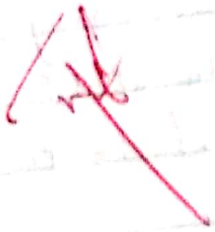
b − a = 6 − 4 = 2 // Store again in stack



b + a = 3 + 2 = 5. // store result in st



b * a = 12 * 5 = 60



```
>>>
ABHAY SINGH 1750
[6, 5, 3, 2, 1, 4]
[1, 2, 3, 4, 5, 6]
>>>
```

Practical 4

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

Theory: SORTING is type in which any random data is sorted ie arranged in ascending or descending order.
BUBBLE sort sometime refferd a sinking sort.
Is a simple sorting algorithm that repeatly step through the list, compare adjacent element and swaps them if they are in wrong order. The pass through the list is repeated until the list is sorted. The algorithm which is a comparision sort 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 compare one element checks if condition fail that only swap otherwise goes on

eg:
First Pass
(5 1 4 2 8) → (1 5 4 2 8) Here algorithm compare the first two elements and swaps 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 element are already order (8 > 5) algorithm does not swap them.

Second Pass:

(1 4 2 5 8) → (1 4 2 5 8)

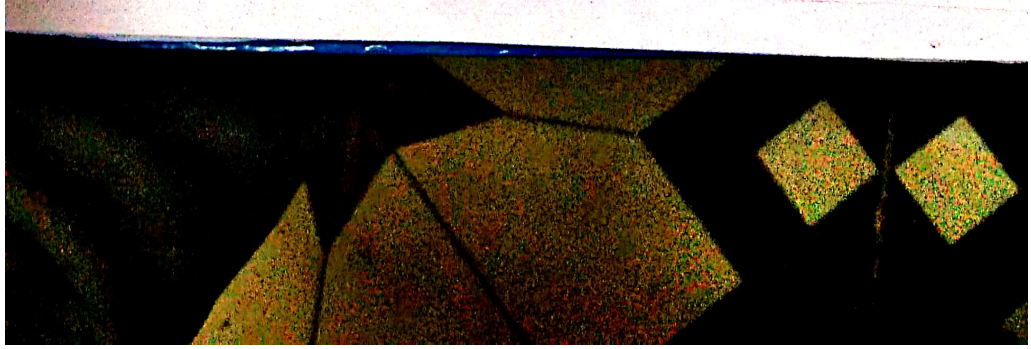
(1 4 2 5 8) → (1 2 4 5 8) swap 4 > 2

(1 2 4 5 8) → (1 2 4 5 8)

Third Pass

(1 2 4 5 8) It check and give the data in sorted order

```
print("ABHAY SINGH 1750")
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]
```



```
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)
Type "copyright", "credits" or "license()" for more infor
>>> ========== RESTART ==========
>>>
ABHAY SINGH 1750
[42, 45, 54, 55, 66, 67, 80, 89, 100]
>>>
```

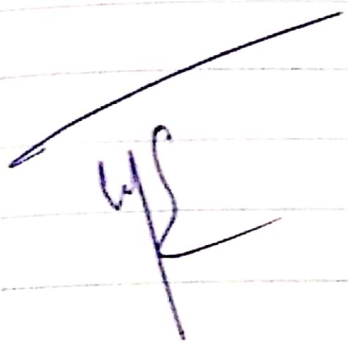
Practical 10

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 version 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 given an array and an element x of array as pivot, put x at correct position, put x at its correct position is sorted and put all smaller element (smaller than x) before x, & put all greater (greater than x) after x All this should be done in linear time

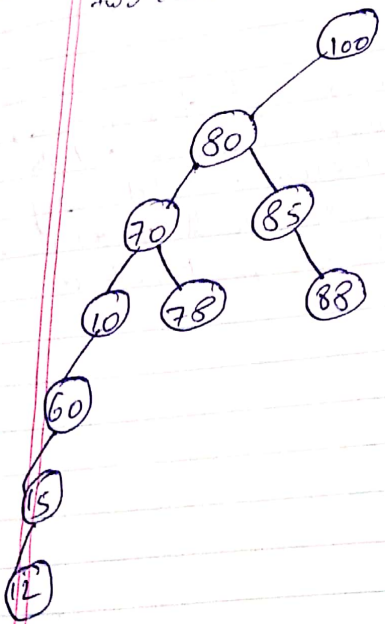


Practical: 11

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 children.



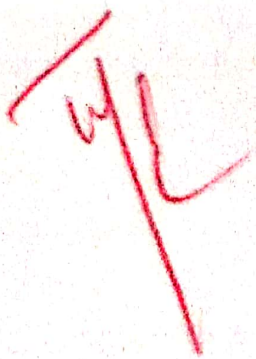
Diagramatic Representation of BINARY SEARCH TREE

```
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.r)
            self.postorder(start.r)
            print(start.data)
print("Name : Abhay")
t=tree()
t.add(12)
t.add(90)
t.add(13)
t.add(85)
t.add(35)
t.add(44)
t.add(71)
t.add(82)
t.add(16)
t.add(12)print("Preorder is :")
t.preorder(t.root)
print("Inorder is :")
```

t.postorder(t.root)

```
Name : Abhay
90  added on right of  12
13  added on left of  90
85  added on right of  13
35  added on left of  85
44  added on right of  35
71  added on right of  44
82  added on right of  71
16  added on left of  35
12  added on left of  13
Preorder is :
12
90
13
12
85
35
16
44
71
82
Inorder is :
12
12
13
16
35
44
71
82
85
90
Postorder is :
12
16
82
71
44
35
85
13
90
12
>>>
```



Practical - 12

Aim : merge sort.

Theory : Merge sort is a sorting technique ba
on divide and conquer technique. with worst-
time complexity being $O(n \log n)$, it one of the mos
desirded algorithm.

Merge sort first divide the array into e
halves and then combine them in a sorted manne

It divide given array in two halves
itself for the two halves and then merge the to
orted halves the merge() function is used fo
merging 2 halves the merge (arr, l, m, r) is ke
roces that assumes that arr [l... m] & arr
irst sorted and merge the two sorted sub
nto one.

```
print("ABHAY SINGH 1750")
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
```

```
        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)
```

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [M
Type "copyright", "credits" or "license()" for more informat
```

```
print("ABHAY SINGH 1750")
a=[23,22,18,96,56,60]
print("Beore 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 seletion sort\n",a)
```

output:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
ABHAY SINGH 1750
Beore sorting
 [23, 22, 18, 96, 56, 60]
After seletion sort
 [60, 96, 96, 96, 96, 96]
>>>
```

Practical 13

Aim: To evaluate : o to sort given random data by using Selection sort.

Theory: Selection Sort is a simple sorting algorithm. This sorting algorithm is an in place comparision based algorithm in which the list is divided into two parts, the sorted part at the left and end. Initially the sorted part is empty and the unsorted part is the entire list. The smallest element is selected from the unsorted array & swapped with the left most element & that element becomes a part of sorted array. This process continues moving unsorted array boundry by one element to the right.

This algorithm is not suitable for large data as its average and worst case complexity are of $O(n^2)$, where n is the number of items.