PRACTICAL -01

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

Theory:- The process of identify or finding a particular record is called searching.

There are two type of search
- → linear search
- → Binary search

The linear search is further classified as
1. SORTED
2. UNSORTED

Here we will look on the UNSORTED linear search. Linear search also known as sequential search is a process that checks every elements in the list sequential until the desired element is found.

When the element to be searched can not specially arranged is ascending or descending order. They are arranged in random manner. that is what it calls unsorted linear . search unsorted linear search.

→ The data linear search entered in random manner.
→ User needs to specify the elements to be searched in the entred list.
→ check the condition that wheather the entered number matches if it matches then display the location place increament 1 as data is sorted the location zero.

→ If all elements are checked one by one a element not found then prompt message not found.

PRACTICAL 02

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 SORT the inputed data in ascending or descending manner.

SEARCHING ⇒ To search elements and to display the same

In searching that two in linear sorted search the data in arranged to ascending to deseending order. that is all what it meant by searching through sorted that ~~is~~ it will arranged data.

SORTED LINEAR SEARCH :-

→ the user is supposed to enter data in sorted manner.

→ user has to given as element for searching through sorted list.

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

→ 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 that then without any processing ~~with~~ can we can say that number is not in the list.

Source code:

```
print("Abhishek")
a=[3,12,23,36,49,66,78]
print(a)
search=int(input("Enter number to be searched from the list:"))
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
```

PRACTICAL - 03

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 linear the array must be sorted in either ascending or descending order.

If each step of the algorithm a comparision is made and the procedure branches into one of two direction. specially the key value is compound to the middle elements of array.
If the key value is ~~use~~ less than or greater than this middle elements the algorithm knows which half of the array to continue searching in leaves the array. is sorted. This proceos is repeated on progressivery smaller segments of the array untill 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.

Output:

```
Case1:
Abhishek
1706
[3, 12, 23, 36, 49, 66, 78]
Enter number to be searched from the list:36
Number found at location: 4
Case2:
Abhishek
1706
[3, 12, 23, 36, 49, 66, 78]
Enter number to be searched from the list:99
Number not in RANGE!
Case3:
Abhishek
1706
[3, 12, 23, 36, 49, 66, 78]
Enter number to be searched from the list:11
Number not in  given list!
```

Source code:

```
print("Abhishek jha")
a=[11,37,10,61,12,8]
print("Before BUBBLE SORT elemets list: \n ",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 elemnts list: \n",a)
```

Output:

```
Abhishek jha
1740
Before BUBBLE SORT elemets list:
 [11, 37, 10, 61, 12, 8]
After BUBBLE SORT elemnts list:
[8, 10, 11, 12, 37, 61]
```

PRACTICA - 04

Aim:- TO sort given random data type by using ~~lea~~ linear short.

Theory:- SORTING is type in which any random data is sorted i.e arranged in ascending or descending order. BUBBLE sort sometimes refered ~~to~~ to as sinking sort.
is a simple sorting algorithm that repeatedly steps through the list compares adjacent elements 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 very smaller or larger elements " " to the top of the list.

Although the algorithm is simple it is also a ~~two~~ too slow as compares one elements 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 compare the first two elements and swap since 5 > 1

(1 5 4 2 8) → (1 4 5 2 8) swap 5 > 4

(1 4 5 2 8) → (1 4 2 5 8) swap 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 in 4 > 2
(1 4 1 2 4 5 8) ⇒ (1 2 5 4 5 8)

Third pass

(1 2 4 5 8) get checks and gives the data in sorted order.

```
## Stack ##
print("abhishek")
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()
```

Output

Abhishek

Stack is full

Data= 70

Data= 60

Data= 50

Data= 40

Data= 30

Data= 20

Data= 10

Stack empty

Aim To demonstrate the use of stack.

Theory: In computer Science a stack serves as collection of elements which two principle operation push which adds an elements to the which the most recently added element that was not yet removed the order may be FIFO (last first in first out) (last order may be LIFO) (last in first out) or FIFO (first in last out)

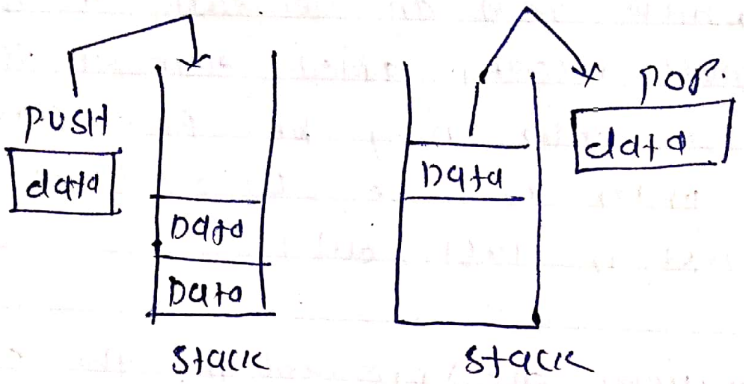
There basic operations are provided in the stack push adds an item in the if the stack is full then if it is said to be over flow condition

pop:

Stack the item are process in the reverse over in which they are pushed of the stack is empty then it is said to be an underflow condition.

- Peek on TOP:- Returns top elements of stack

isempty:- Returns ~~the~~ true if stack is empty else false.



Last in - First - out.

Queue add and Delete

```
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)
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(44)
Q.add(55)
Q.add(66)
Q.add(77)
Q.add(88)
Q.add(99)
Q.remove()
Q.add(66)
```

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

PRACTICAL – 06

Aim: To demonstrate queue add and delete.

Theory :- Queue is a linear data structure where the first element is inserted for one end called REAR and deleted from other end called as FRONT.

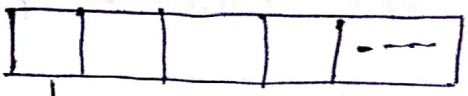
Front point to the beginning of the queue and rear points to the end of the queue. Queue follow the FIFO (First – in – First – out) structure

Ans- Ascending to its FIFO structure elements insted First file also be pronounced First
In a queue one end is always used to input data. (Inqueue) and the other is used to delete data (dequeue) because queue is open on both of its ends.

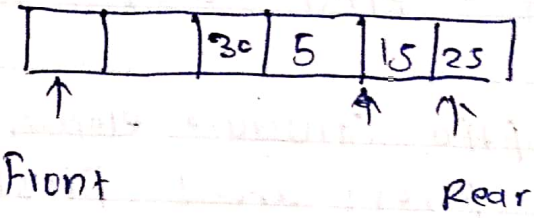
enqueue() can be turned as add() in queue i.e adding a element in queue. Dequeue() can be tamed as delet or Recor.. i.e deleting or renouncing of elements.

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

Reard is used to get the last iter from a queue.



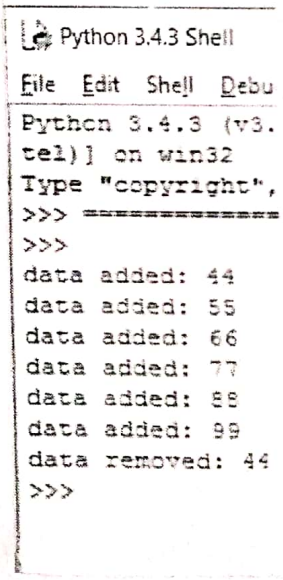
on both sides ↲ Queue ↳ can have end.



Front = 2

Rear = 5

```
#(circular queue)
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
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:
self.r=s
print("Queue is full")
def remove(self):
        n=len(self.l)
ifself.f<=n-1:
print("data removed:",self.l[self.f])
self.f=self.f+1
else:
            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(44)
Q.add(55)
Q.add(66)
Q.add(77)
Q.add(88)
Q.add(99)
Q.remove()
Q.add(66)
```



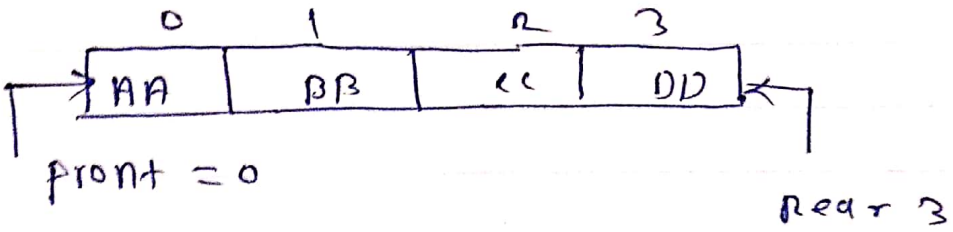
PRACTICAL - 7

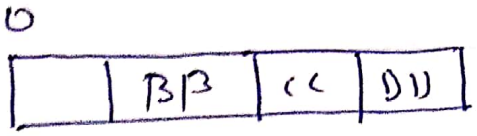
Aim To demonstrate the use of circular queue is data - structure.

Theory: The queue that are implement using an array suffer from one limitations in that implementation there is a possibility that the queue is seperated there might the empty stats. at the beginning of the queue.

To queue come this limitations are use can implement queue as circular queue. in circular queue are go on adding element to the queue and near the end of the array, the next element is sorted in the first sort of array.

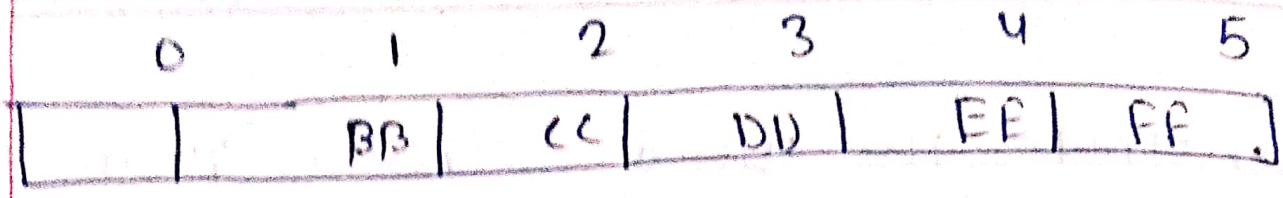
Example:



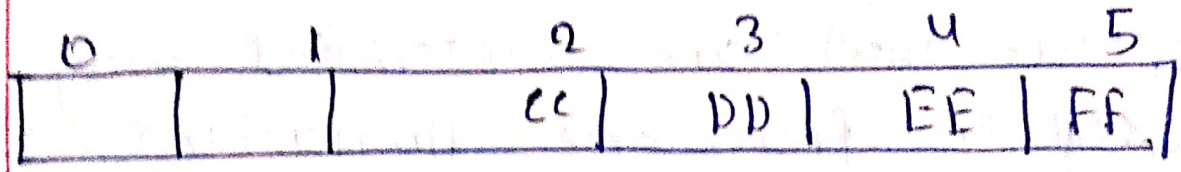


Front = 1 Rear = 3

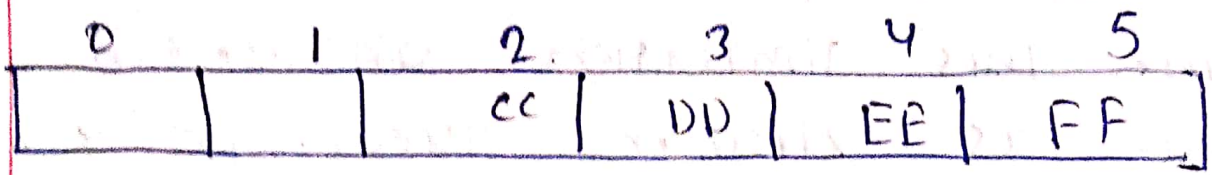
A.



Front = 1 Rear = 5.



Front = 2 Rear = 5



Front = 2 Rear = 0

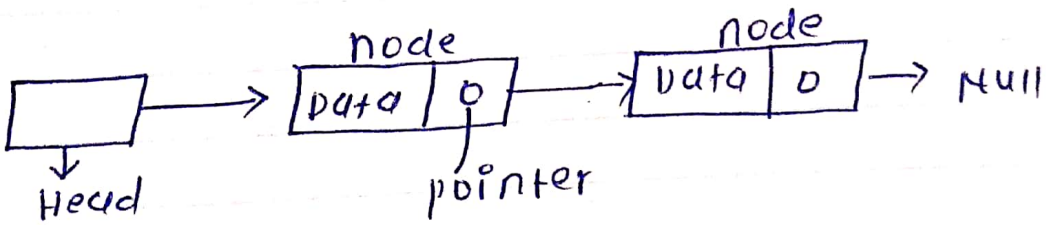
PRACTICAL – 8

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 items. If 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 first link called first.

LINKED LIST Representation:-



TYPES of LINKED LIST:-

① Simple
② Doubly
③ Circular.

BASIC Operations

① Insertions
② Deletion
③ Display
④ search
⑤ Delete.

SOURCE CODE:

```
print("Abhishek")
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
defaddL(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
defaddB(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:

```
Abhishek
20
30
40
50
60
70
80
```

```
## Postfix Evaluation ##

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="8 6 9 * +"
r=evaluate(s)
print("The evaluated val

print("Abhishek")
```

Output

```
The evaluated value is: 62
Abhishek
```

PRACTICAL 9

Aim To evaluate postfix expression 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 collection of operators is placed after the operands.

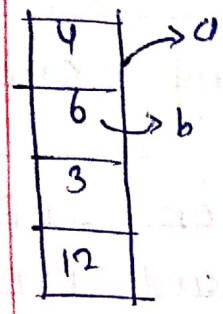
Steps to the followed:-

1) Read all the symbols one by one from left to right in the given postfix expression.
2) If the reading symbols is ~~operations~~ operand then push it on the stack.
3) If the reading symbols is operators (+, -, *, /, etc.) then perform two pop operations and store the two popped operands in two different variables (operand 1 and operand 2). Then perform reading symbols operator using operand 1 & operand 2 and push result back on to the stack.
4) Finally perform a pop operation and display the popped value as final result.

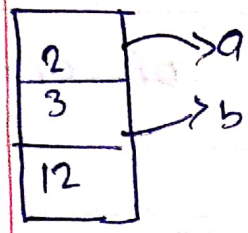
Value of ~~postfi~~ postfix expression.

$$5 = 12 \quad 3 \quad 6 \quad 4 \quad - \quad + \quad *$$

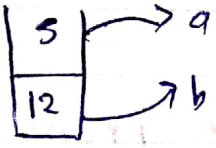
Stack :



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

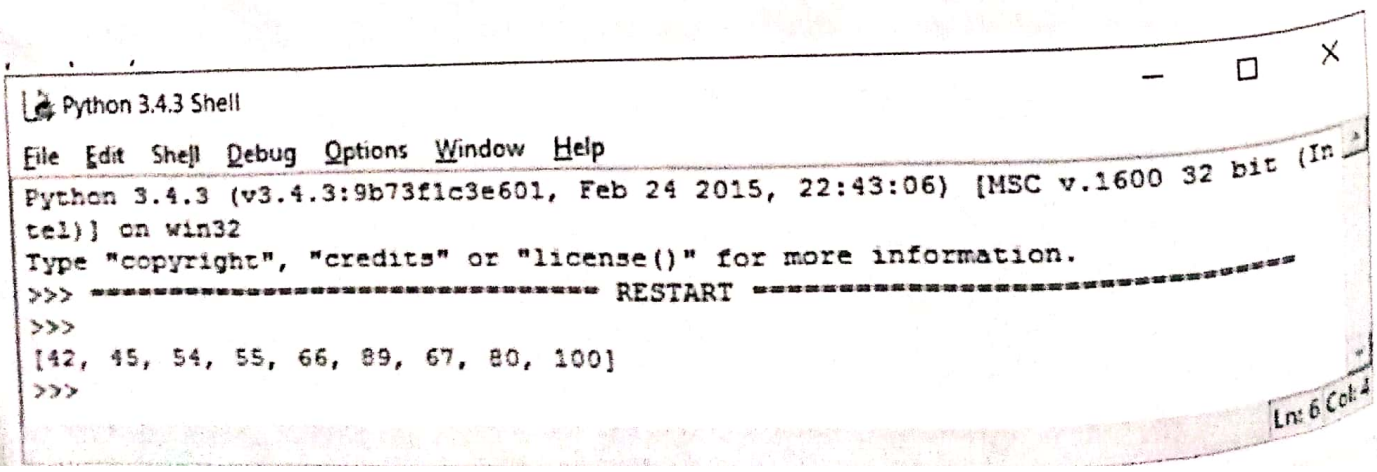


b + a = 3 + 2 = 5 // Store result in stack.



b * a = 12 × 5 = 60

```
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)
```



PRACTICAL – 10

Aim:- TO 'evaluate' i.e 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 point and partitions the given array around the picked. There are many different versions of quick sort that pick point in different ways.

1) Always pick first element as .

2) Always pick last element as .

3) pick a random element as pivot

4) pick median as pivot.

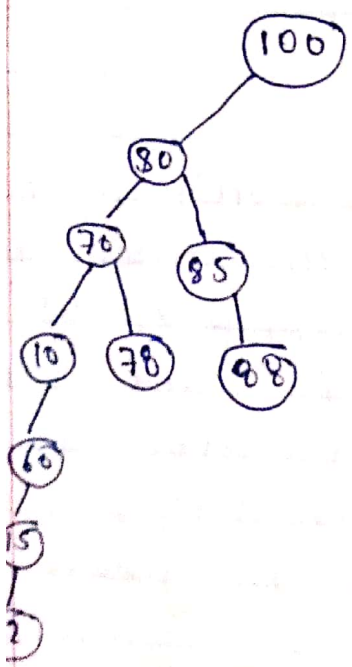
The key process in quicksort is partitions ().
Target of partitions is given an array and an element 'x' of array as pivot, put x at its correct positions in sorted array and put all smaller element (smaller than x) before x, & put all greater elements (greater than x) after x. All this should be done in linear time.

PRACTICAL – 11

Aim :- Binary tree and Traversal.

THEORY :- A binary tree is a special type of tr in which every node or vertex has either no child or one child node or two chi nodes.

A binary tree is an important class of a tree data structure in which a node can ha at most two children.



Digramatic Representation o BINARY SEARCH TREE

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):
        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.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)
T.add(12)
print("preorder")
```

PRACTICAL - 12

AIM:- MERGE SORT

THEORY:- MERGE SORT is a sorting technique based on divide and conqure technique with worst case time complexity being $\theta(n \log n)$ it one of the most respected algorithms.

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

It divides input array in into halves, calls itself for the two halves and then merge the two sorted halves. The merge() function is used for (arr, l, m, r) is key process that assumes that are [l --- m] and arr [m+1 -- r] are sorted and merges the two sorted sub- array into one.

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

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

1. INORDER
2. PREORDER
3. POSTORDER

① 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

② PRE-ORDER :- ~~The left subtree~~ In root node is visited first then the left subtr and finally the right subtree.

③ POST-ORDER :- The root node is visited last left subtree, then the right subtree and finally root node.

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

Output

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

Pre-order

100
80
70
10
60
15
12
78
85
88

Inorder

10
12
15
16
70
78
80
88
100

Postorder

10
12
15
60
78
80
85
88
100

Abhishek