☆☆ INDEX ☆☆

PRACTICAL - 1

AIM : To search a number from the list using Linear Unsorted.

THEORY : Searching is the process of identifying or finding a particular record.
It is divided into two parts,
* Linear Search
* Binary Search
Linear Search is further divided into two parts,
* Linear Unsorted and * Linear Sorted
In this practical we will look on the Linear Unsorted search. It is also ~~known~~ as sequential search. Sequential search is a process that checks every element present in the list sequentially until the desired element is found. When the elements to be searched are not arranged in ascending or descending ~~order~~ order. They are arranged in random manner. That is what is meant as Unsorted Linear Search.
Some characteristics of linear search unsorted include :-
→ The elements are entered in random manner.

→ The user needs to specify the element that is to be searched.
→ It checks the condition whether the entered number matches if it matches.
→ If the elements are checked and the desired element is not found, prompt message number not found.

Source Code :

```
#Linear Unsorted
print(" \n Dharan Shah \n CS-1701 \n Batch A")
j=0
A=[4,29,21,64,25,24,3]
search=int(input("Enter Number To Be Searched"))
for i in range (len(A)):
    if(search==A[i]):
        j=1
        print("Number Found At :",i)
        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/Dharan/Desktop/Python Programmes/SEM 2/Data
Structures/LinearUnsorted.py

 Dharan Shah
 Roll No-1701
 Batch A
Enter Number To Be Searched 21
Number Found At : 2
>>>
```

PRACTICAL-2

AIM: To search a number from the list using Linear sorted method.

THEORY:- The different modes of data structures are Searching and Sorting.

* Searching is to search an element from given list and display it.
* Sorting is to sort the data ascending or degending manner.

As there are two types of Linear searches → Unsorted And Sorted.

Here, in this practical we will talk about Linear sorted method.

In linear sorted method the data is arranged is ascending or descending. Now, the element is searched from the sorted data.

It is known as Sorted Linear Search.

* Some characteristics are as below:-

The user is supposed to enter data in sorted manner.

* If the element is found, display the location from 'zero'.
* If element is not found, print the same,

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

SOURCE CODE :

```
#Linear Sorted
print(" \n Dharan Shah \n Roll No-1701 \n Batch A")
j=0
A=[3,4,21,24,25,29,64]
search=int(input("Enter Number To Be Searched"))
if ((search<A[0]) or (search>A[6])):
    print("Number Does Not Exist")
else:
    for i in range(len(A)):
        if (search==A[i]):
            j=1
            print("Number Found At :",i)
            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/Dharan/Desktop/Python Programmes/SEM 2/Data
Structures/LinearSorted.py

 Dharan Shah
 Roll No-1701
 Batch A
Enter Number To Be Searched4
Number Found At : 1
>>>
```

SA.

SOURCE CODE:

```
#Binary Search
print(" \n Dharan Shah \n CS-1701 \n Batch A")
A=[3,4,21,24,25,29,64]
print (A)
search=int(input("Enter Number To Be Searched :"))
l=0
h=len(A)-1
m=int((l+h)/2)
if((search<A[l]) or (search>A[h])):
    print("Does Not Exist")
else:
    while l!=h:
        if(search==A[m]):
            print("Number Found At",m)
            break
        else:
            if(search<A[m]):
                h=m-1
                m=(l+h)/2
                print("Number Found At",m)
            else:
                l=m+1
                m=(l+h)/2
                print("Number Found At",m)
```

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/Dharan/Desktop/Python Programmes/SEM 2/Data Structures/Binary


 Dharan Shah
 CS-1701
 Batch A
[3, 4, 21, 24, 25, 29, 64]
Enter Number To Be Searched :24
Number Found At 3
>>>
```

PRACTICAL - 3

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

THEORY: A binary search is also known as a half-interval search. It is an algorithm used in computer science to locate a specified value key within an array. For the search to be binary, the array must be sorted in either ascending or descending order. At each step, the algorithm makes a comparison and the procedure branches in one or two directions. Mainly, the key value is compared to the middle element of the array. If the key value is less than or greater than this middle element, the algorithm knows which half of the array to continue searching because the array is sorted.

The process is repeated on progressively smaller segments of the array, until the value is located, because each step in the algorithm divides the array size in half, a binary search when ended, it will take a logarithmic time.

PRACTICAL - 4

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

THEORY: Bubble Sort, it is also refered as sinking sort, it 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 comparison sort, is named for the way smaller or larger elements "bubble" to the top of list. Although bubble sort is one of the simplest sorting algorithm, to understand its complexity means that its efficiency decreases dramatically on lists.

Due to its simplicity, bubble sort is often used to introduce the concept of algorithm, or a sorting algorithm to introductory computer science subjects. Odd-even sort is a parallel version of bubble sort, for message passing systems.

SOURCE CODE :-

```
#Bubble Sort
print(" \n Dharan Shah \n CS-1701 \n Batch A")
A=[3,6,5,8,10,2,1]
print("Before Bubble Sort elements list :",A)
for passes in range (len(A)-1):
    for compare in range (len(A)-1-passes):
        if(A[compare]>A[compare+1]):
            temp=A[compare]
            A[compare]=A[compare+1]
            A[compare+1]=temp
print("After Bubble Sort elements list :",A)
```

OUTPUT :-

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 b
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/Dharan/Desktop/Python Programmes/SEM 2/Data
Structures/BubbleSort.py

 Dharan Shah
 CS-1701
 Batch A
Before Bubble Sort elements list : [3, 6, 5, 8, 10, 2, 1]
After Bubble Sort elements list : [1, 2, 3, 5, 6, 8, 10]
>>>
```

SOURCE CODE:-

```
#STACK#
print("\nDharan Shah\n1701\nBatch A")
class stack:
        global tos
        def __init__(self):
             self.l=[0,0,0,0,0,0,0]
             self.tos=-1
        def push(self,data):

             n=len(self.l)
             if self.tos==n-1:
                  print("stack is full")
             else:
                  self.tos=self.tos+1
                  self.l[self.tos]=data
       def pop(self):
            if self.tos<0:
               print("stack empty")
            else:
                 k=self.l[self.tos]
                 print("data=",k)
                 self.tos=self.tos-1
s=stack()
s.push(10)
s.push(20)
s.push(30)
s.push(40)
s.push(50)
s.push(60)
s.push(70)
s.push(80)
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
```

```
Dharan Shah
1701
Batch A
stack is full
data= 70
data= 60
data= 50
data= 40
data= 30
data= 20
data= 10
stack empty
>>>
```

PRACTICAL-5

AIM:- To demostrate use of Stack.

THEORY:- A Stack is an abstract data type that serves 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 basic operations are performed in the Stack are:-

1) PUSH 2) POP

⇒ PUSH

Adds an item in the if the stack is full then it is said to be over flow condition.

⇒ POP

Removes an item from the stack the items are popped in the reversed order in which they are pushed. If the Stack is empty, then it is said to be an underflow condition.

PRACTICAL-6

AIM: To demostrate 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 other end called FRONT.

FRONT points to the begining of the queue and REAR points to the end of the queue.

It follows First In First Out structure, according to its structure, element which is inserted first will also be removed first.

In a queue, one end is always used to insert data and other is used to delete because queue is open at both of its ends.

Adding in queue is termed as Enqueue() or Add(). Deleting is termed as dequeue() or delete(). or remove().

```
# Queue
print("\nDharan Shah\n1701\nBatch A")
class Queue:
    global r
    global f
    def __init__(self):
        self.r=0
        self.f=0
        self.l=[0,0,0,0,0,0]
    def add(self,data):
        n=len(self.l)
        if self.r<n-1:
            self.l[self.r]=data
            self.r=self.r+1
        else:
            print("Queue is full")
    def remove(self):
        n=len(self.l)
        if self.f<n-1:
            print(self.l[self.f])
            self.f=self.f+1
        else:
            print("Queue is empty")
Q=Queue()
Q.add(30)
Q.add(40)
Q.add(50)
Q.add(60)
Q.add(70)
Q.add(80)

Q.remove()
Q.remove()
Q.remove()
Q.remove()
Q.remove()
Q.remove()
```

```
== RESTART: C:/Users/Dharan/AppData/Local/Programs/Python/Python37/Queue.py ==

Dharan Shah
1701
Batch A
Queue is full
30
40
50
60
70
Queue is empty
>>>
```

```
#CIRCULAR QUEUE
print("\nDharan Shah\n1701\nBatch A")
class Queue:
    global r
    global f
    def __init__(self):
        self.r=0
        self.f=0
        self.l=[0,0,0,0,0,0]
    def add(self,data):
        n=len(self.l)
        if self.r<=n-1:
            self.l[self.r]=data
            print("data added:",data)
            self.r=self.r+1
        else:
            s=self.r
            self.r=0
            if self.r<self.f:
                self.l[self.r]=data
                self.r=self.r+1
            else:
                self.r=s
                print("Queue is full")
    def remove(self):
        n=len(self.l)
        if self.f<=n-1:
            print("data removed:",self.l[self.f])
            self.f=self.f+1
        else:
            s=self.f
            self.f=0
            if self.f<self.r:
                print(self.l[self.f])
                self.f=self.f+1
            else:
                print("Queue is empty")
                self.f=s
=Queue()
add(21)
add(31)
add(57)
add(67)
add(92)
add(96)
remove()
add(56)
```

```
a added: 57
a added: 67
a added: 92
a added: 96
a removed: 21
```

PRACTICAL - 07

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

THEORY :- The queue that we implement using an array suffer from one limitation. In that implementation, there is possibility that the que is full, though there might be empty slots. At the begining, To overcome this limitation we can implement using circular queue.

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

Circular Queue is a linear data structure in which the operations are performed based on FIFO principle and last position is connected back to the first position to make a circle.

Application of circular Queue includes,

1. Computer Controlled Traffic Signal System.
2. CPU scheduling and memory management.

PRACTICAL - 08

AIM: To demonstrate the use of Linked List in data structure.

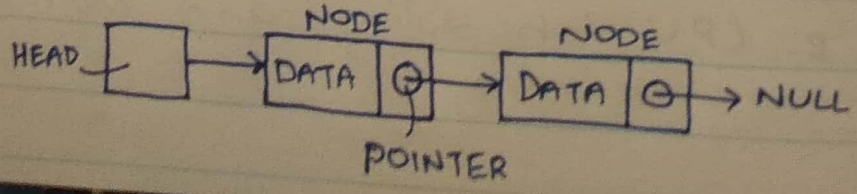
THEORY: A Linked List is a sequence of data structures. Linked List is a sequence of Links which contains items. Ech Link contains a connection to other Link.

* 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.

A Linked List is a Linear data structure where each element is a seperate object.

There are 3 types of Linked Lists,
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

REPRESENTATION:



SOURCE CODE :-

```
#Simple LinkedList
print("Dharan Shah\n1701\nBatch A")
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(45)
start.addL(55)
start.addL(65)
start.addL(75)
start.addB(35)
start.addB(25)
start.addB(15)
start.display()
```

```
15
25
35
45
55
65
75
>>>
```

PRACTICAL - 09

AIM : To evaluate postfix expression using Stack.

THEORY : Stack is an Abstract Data Type and works on LIFO (Last_In First_Out) i.e. PUSH & POP operations.

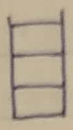
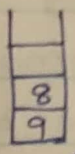
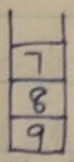
A postfix operation is a collection of operators and operands in which the operator is placed after the operands.

To evaluate a postfix expression, we can use the following steps :

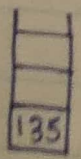
1. Read all the symbols one by one from left to right in the given postfix expression.
2. If the reading symbol is an operand, then push it on to the Stack.
3. If reading symbol is an operator (+, -, *, /) then perform two pop operation and store the two popped operands in two different variables. Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally, perform a pop operation and display the popped value as final result.

value of post fix expression:
"9 8 7 + *"

Stack:

Reading	Stack	Evaluated
Initially	 Stack is Empty	Nothing
9	 push(9)	–
8	 push(8)	–
7	 push(7)	–

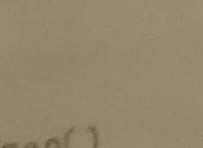
+	 push(result)	value1 = pop(); // 7
		value2 = pop(); // 8
		result = 7+8 // 15
		push(15)
		(7 + 8)

*	 Push(result)	value1 = pop(); //15
		value 2 = pop(); //9
		result = 15*9 // 135
		push(135)

```
#Postfix
print("\nDharan Shah\n1701\nBatch A")
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="9 8 7 + *"
r=evaluate(s)
print("The evaluated value is:",r)
```

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/Dharan/Desktop/Python Programmes/SEM 2/Data
Structures/PostfixCode.py

Dharan Shah
1701
Batch A
The evaluated value is: 135
>>>
```



Display(result)
135

02

Source Code:-

```
#QuickSort
print("\nDharan Shah\n1701\nBatch A")

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=[41,56,48,63,84,68,52,87,100]
quickSort(alist)
print(alist)
```

PRACTICAL : 10

AIM : Quick Sort

THEORY : The quicksort algorithm uses divide and conquer technique. However, in quick sort we ~~dev~~ divide the list into 2 parts based on a preselected pivot as opposed to dividing into 2 equal parts in merge sort. Now, once the lists are divided based on the pivot, the left sublist contains all the elements lesser that the pivot element and the right sublist contain elements greater than the pivot element. The quick sort is a recursive algorithm based on the divide and conquer technique. The quick sort algorithm implements in following steps:

(i) The first element from the list is selected as pivot element. We focus on placing the pivot element at its correct position. Once we placed the pivot elemen at its correct position, the sublist is divided into 2 parts L and R not included pivot element. L list contains all elements lesser than pivot element and R contains all elements greater to than pivot elements.

(ii) Step i) is again applied to the 2 sublists recursively until there is not more than one element in the sub element.

(iii) Once, we place elements all the elements in the correct position we combine the individual elements in the original ~~xxxxxxxxx~~array.

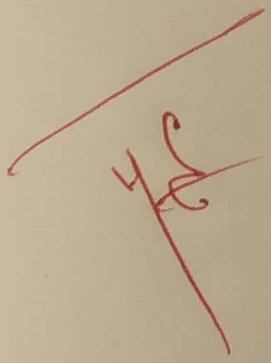
Efficiency Analysis :-

The worst case time complexity for quick sort is $O(n^2)$. However for average case for quick sort is $O(n \log n)$. If we compare the average case for merge sort and quick sort we find that they are same. However the advantage of quicksort over merge sort is that quick sort does not require any extra lists to store the sublists as in case of merge sort.

Output :-


```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/Dharan/Desktop/Python Programmes/SEM 2/Data
Structures/QuickSort.py

Dharan Shah
1701
atch A
, 48, 52, 63, 68, 84, 56, 87, 100]
```



```
#Binary Tree & Traversal

print("\nDharan Shah\n1701\nBatch A")
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("Inorder")
T.inorder(T.root)
print("Postorder")
T.postorder(T.root)
```

PRACTICAL - 11

AIM: BINARY TREE AND TRAVERSAL

THEORY: Binary Search Tree can be defined as a class of binary tree, in which the nodes are arranged in specific order. This is also called as ordered binary search tree. In a binary search tree, the values of all the nodes in the left sub-tree is less than the value of root. Similary, value of all the nodes in the right sub-tree is greater than or equal to the value of root. This rule will be recursively applied to all the left and right sub-trees of the root.

Traversing the tree :-

There are mainly three types of tree traversals,

1) Pre-order traversal
2) Post-order traversal
3) In-order traversal

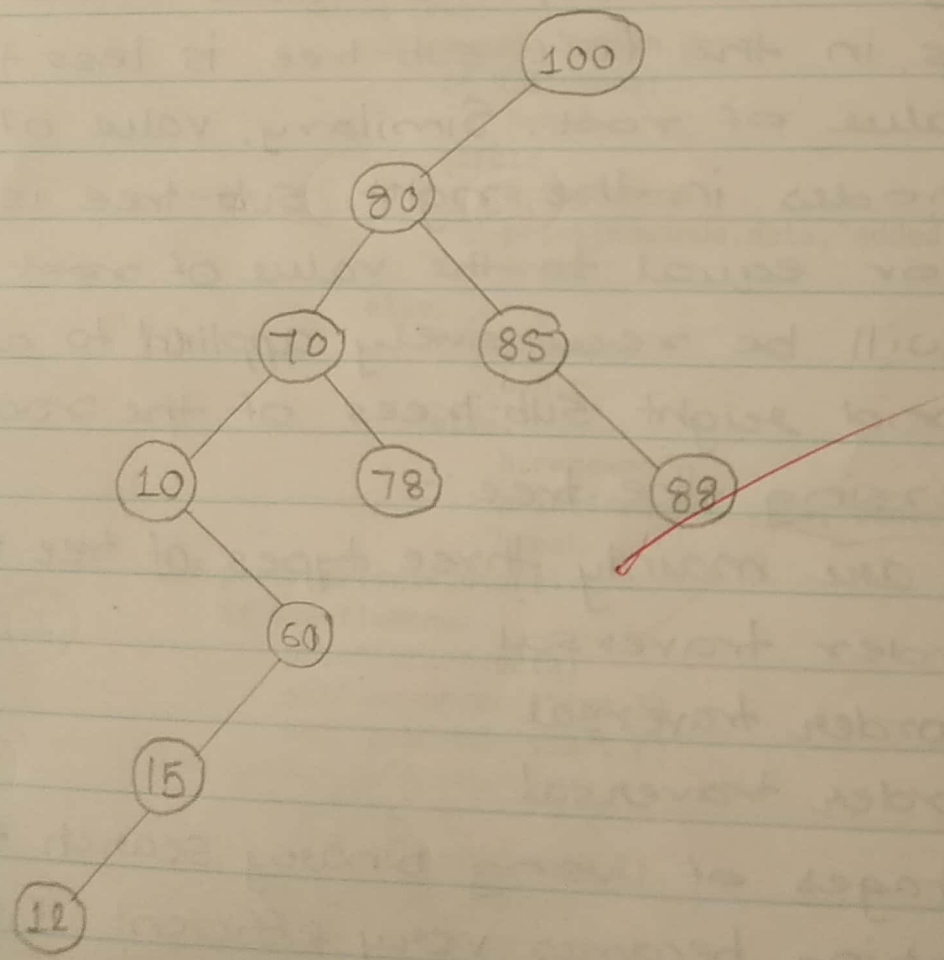
Advantages of using binary search tree,

1) Searching becomes very efficient in a binary tree since we get a hint at each step, about which sub-tree contains the desired element.
2) The Binary Search Tree is considered as efficient data structure in compare to arrays and Link Lists.

3) It also speed up the insertion and deletion operations as compare to that in array and linked list.

Searching for an element in a binary search tree takes $O(\log_2 n)$ time. In worst case, the time is takes to search an element is $O(n)$.

* Binary Tree :-



```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: D:\DS P12.py ==========

Dharan Shah
1701
Batch A
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
10
12
15
60
70
78
80
85
88
```

SOURCE CODE:

```
#MergeSort
print("\nDharan Shah\n1701\nBatch A")
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:
```

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(nlogn), it is one of the most respected algorithms.
In merge sort, the given unsorted array with n elements, is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these sub arrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.
Merge Sort is quite fast, and has a time complexity of O(n*logn). It is also a stable sort, which means the equal elements are order in the same order in the sorted list.
Time complexity of MergeSort is O(n*logn) in all the 3 cases (worst, average and best) as merge Sort always divides the array into two halves and takes linear time to merge two halves. It requires equal amount of additional spaces as the unsorted array. Hence, its not at all recommended for searching large unsorted arrays. It is the best sorting technique used for sorting Linked Lists.

```
            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=[12,11,13,5,6,7,52,47,21]
n=len(arr)
mergesort(arr,0,n-1)
print("After Mergesort\n",arr)
```

OUTPUT:

```
>>>
```

Dharan Shah

1701

Batch A

After Mergesort

[5,6,7,11,12,13,21,47,52]