THEORY:

linear search is one of the simplest searching algorithm in which each item is subsequentially matched with each item in the list.

It is worst searching algorithm with worst case Run complexity. It is a force approch on the other hand increase if an ordered list, instead of searching the list in sequence. A binary search is used which will start by examining the middle term linear search is a technique to compare each other and every element with key

AIM: linear search

Q1.] Sorted array

ALGORITHM

STEP 1: Define a function with two parameter. Use for conditional statement with range in length of array to find index

STEP 2: Now use if conditional statement to check whether the given number by user is equal to the element in array

STEP 3: If the conditional in step 2 satisfy, return the index no. of the given array if the condition doesn't satisfies then get out of loop

STEP 4: Now initialize a variable to enter elements in the array from user. Now use split() method & to split the value

STEP 5: Now initialize a variable as empty array

STEP 6: Now use for conditional statement to append the elements given as input by user in the empty array

STEP 2: Now again initialize another variable to ask to find the element in array.

STEP 3: Again initialize a variable to call the define function.

STEP 4: Use if condition statement to check if the variable in step 3 matches with the element you want to find


```
# sorted array
# CODE
def linear(arr,x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
inp = input("Enter elements in array").split()
array = []
for ind in inp:
    array.append(int(ind))
print("Element in array are:", array)
array.sort()
x1 = int(input("Enter Element to be searched:"))
x2 = linear(array, x1)
if x2 == x1:
    print("Element found at position", x2)
else:
    print("Element not found")
```

```
>>> Enter Element in array: 1 2 3 4 5
>>> Elements in array are: [1,2,3,4,5]
>>> Enter Element to searched: 2
    The Element is found at position 1
>>> Enter Elements in array: 3 2 5 14
>>> Elements in array are: [1,2,3,4,5]
>>> Enter Element to be searched: 6
    Element not found
```

```
# Unsorted array:
def linear(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
inp = input("Enter Elements in array:").split()
array = []
for ind in inp:
    array.append(int(ind))
print("Element s in array are:", array)
x1 = int(input("Enter the Element to be searched:"))
x2 = linear(array, x1)
```

Q2] Unsorted array:

Algorithm

STEP 1: Define a function with two parameters we for conditional statement with range is length of array to find index.

STEP 2: Now we if conditional statement to check whether the given statement is equal to Element in array.

STEP 3: If the condition in step 2 satisfies return the index no. of the given array. If the condition doesn't satisfies the get out of loop.

STEP 4: Now initialize a variable to Enter Element in the array from user now we split() method to split the values.

STEP 5: Now again initialize another variable to ask user to find the element in array.

STEP 6: Again initialize a variable to call the define function.

```
if pos != -1:
    print ("Element found at location", pos)
Else:
    print ("Element not found")
```

>>> Enter elements in array : 3 2 4 5 1
>>> Element in array are : [3 2 4 5 1]
>>> Element to be searched : 4
Element is found at location 2
>>> Element in array are : [2, 4, 5, 3, 1]
>>> Element to be searched : 6
Element not found

```
# binary search:
# create
    def binary(arr, key):
        start=0
        end=len(arr)
        while start<end:
            mid=(start+end)//2
            if arr[mid]>key:
                end=mid
            elif arr[mid]<key:
                start=mid+1
            else:
                return mid
        return -1
    array=input("Enter the sorted list of numbers:")
    arr=[]
    for ind in array:
        arr.append(int(ind))
    key=int(input("Enter Element to search:"))
    index=binary(arr, key)
    if index<0
        print("Element not found")
    Else:
        print("Element not found at index", index)
```

PRACTICAL NO: 2

AIM: Binary Search

ALGORITHM:

STEP 1: Define a function with two parameters. Now initialise variable with 0. Use while conditional statements to find mid value.

STEP 2: Use of conditional statement to determine at which position the mid value should point

STEP 3: If the condition doesnt satisfies then return -1

STEP 4: Now initialize a variable to enter the elements in the array

STEP 5: Use for conditional statement to append the elements in empty array.

STEP 6: Now initialize a variable to find the elements in array.

STEP 7: Now initialize a variable to call the defined functi

STEP 8: Now use if conditional statements to determine the index value and print the index value

binary Search

CODE :

```
def binarey (arr, key) :
start = 0
end = len (arr)
while start < end:
    mid = (start + end) // 2
    if arr[mid] >= key:
        end = mid
    elif arr[mid] < key:
        start = mid + 1
    else:
        return mid
return -1
array = input ("Enter the sorted list of numbers:")
arr = []
for ind in array:
    arr.append (int (ind))
key = int (input ("Enter element to search:"))
index = binary (arr, key)
if index < 0:
    print ("Element not found")
else:
    print ("Element not found at index", index)
```

OR:

>>> Enter the element in array:
3 5 10 12 15 20

>>> Element to be searched = 12
12 Element was found at index = 3

>>> Enter the Element in array:
30 15 6 78

>>> Element to be search = 2
Element was not found

THEORY:

Binary search is also known as half interval search logarithmic search or binary is a search algorithm that finds the position of target value within a sorted array. If you are looking for number which is at the end of list then you need to search entire list in linear search which is time consuming it can be avoided by using binary fusion search

PRACTICAL NO: 3

AIM: Implementation of Bubble sort program on given list.

THEORY: Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if the simplest from the sorting available. In this, we sort the given elements in ascending or descending order by comparing two adjusent Element at a time

ALGORITHM:

STEP 1: Bubble sort algorithm starts by comparing first two Elements of an array and swapping if necessary.

STEP 2: If we want to sort the elements of array in ascending order then first element is greater than second then we need to swap the element.

STEP 3: If the given element is smaller than second element then we do not swap the element.

CODE

```
inp = input ("Enter elements") split()
arr = []
for ind in inp:
        arr append (int (ind))
Print ('Elements of array before sorting then = ", arr)
n = len (arr)
for i in range (0,n):
    for j in range (n, i):
        if arr[j] < arr [i]:
            temp = arr [i]
            arr [j] = arr[i]
            arr [i] = temp
    Print ("Element of array after bubble sort:", arr)
```

>>> Enter Element: 2 3 6 1 5
>>> Elements of array before sorting: [2, 3, 6, 1, 5]
>>> Element of array after sorting:
[1, 2, 3, 5, 6].

STEP 4: Again second & third elements are compared & swapped if it is necessary & this process go on until last & second last element is compared & swapped.

STEP 5: If there are n elements to be sorted then the process mentioned n-1 above should be mentioned to get required output.

STEP 6: Stick the output & input of above algorithm of bubble sort stepwise

PRACTICAL NO: 4

AIM: Implementation of stack using python list

THEORY: A stack is linear data structure that can be represented in a real-world from by physical stack or pile. The element in the stack are the top most position. Thus, it works in the life principle (last in first out). It has 3 basic operation namely: push, pop, peek.

ALGORITHM:

STEP 1: Create a class stack with instance variable item.

STEP 2: Define the init() method with self argument & initialize the initial value & then initialite to an Empty list.

STEP 3: Define methods push & pop under the class stack.

STEP 4: Use if statement to give the condition that if length of given list is great than the range of list the print stack is full.

STEP 5: Use the 'else' statement to print a statement as input the element into the stack & initialize the value.

CODE:

```
# stack
class stack :
    def __init__(self):
        self.tos = -1
    def push(self, data):
        n = len(self.l)
        if self.tos == n-1:
            print ("The stack is full:")
        else :
            self.tos = 1
            self.l[self.tos] = data
    def pop(self):
        if self.tos<0:
            print ("The stack is empty")
            self.l[self.tos] = 0
            self.tos -= 1
n = stack()
```

STEP 6: Push method used to insert the element but pop method used to delete the element from the stack.

STEP 7: If in pop method, value is less than 1 the element from stack at topmost position

STEP 8: Assign the element value, in push method & print the given value.

STEP 9: Attach the input & output of above algo.

STEP 10: First condition checks whether the no. of Elements are zero while second case whether top ie. assigned any value If top is not assigned any value then print that the stack is empty.

II OUTPUT :

```
>>> x.push(10)
>>> x.push(7)
>>> x.push(8)
>>> x.push(79)
>>> x.push(72)
>>> x.push(69)
    The stack is full.
>>> x.pop()
72
>>> x.l
[10,7,8,79,72,69]
```

PRACTICAL NO: 5

AIM: Implement quick sort to sort the given list.

THEORY: The quick sort is a recursive algorithm that based on the divide & conquer technique

ALGORITHM:

STEP 1: Quick sort first selects a value, which is called pivot value first value element since as our first pivot value since we know that first will eventually end up as last in that list.

STEP 2: The partition process will happen next. It will find a split point & at the same move other items to the appropriate side of the list either less than or greater than pivot value.

STEP 3: Positioning begins by locating two position markers, lets call them leftmark & right mark at the beginning & End of remaining item in the list. The goal of wrong side will repeat do first value while also converting on the split point

STEP 4: We begin by increasing leftmark we locate a value that is greater than the pivot value, we then decrement right mark until we find value that is less than the pivot value.

```
Print ("Quick Sort")
def partition (arr, low, high):
    i = low-1
    pivot = arr[high]
    for j in range (low, high):
        if arr[j] <= pivot
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1
def quicksort (arr, low, high):
    if low < high:
        pi = partition (arr, low, high)
        quicksort (arr, low, pi-1)
        quicksort (arr, pi+1, high)
I1 = input ("Enter Elements in the list :").split()
alist 1 = []
for b in I1 =
        alist 1. append (int (b))
Print ("Element input list are: ", alist)
n = len(alist 1)
quicksort (alist 1, 0, n-1)
Print ("Element after quicksort are :", alist)
```

OUTPUT:

Quick sort

Enter Elements in the list: 21, 22, 20, 30, 24, 56

Elements in list are: [21, 22, 20, 30, 24, 56]

Elements after quicksort are [20, 21, 22, 24, 30, 56]

STEP 5: At the point where right mark meets the left mark stop the position of rightmark is now the split point

STEP 6: The pivot value can be exchanged with the content of split point and pivot value is now in place.

STEP 7: In addition all the items to left of split point are less than pivot value & all the items left to the right of split point are greater than it. The list can now be divided at split point and greater than pv. The list can now be divided at split point & quick sort can be invoked on the two value.

STEP 8: The quick sort function invoked a recursive function quick sort helps

STEP 9: Quick sort helper, begins with some base as the simple case

STEP 10: If length of the list is less than 0 or equal one it is already sorted

STEP 11: If it is greater than it can be partitioned & recursive function

STEP 12: The partition function implement the process described earlier.

PRACTICAL NO: 6.

* Aim: Implementing a queue using python list.

THEORY: Queue is a linear data structure which has two references front & rear.
Implementing a queue using python list is the simplest as the python list provides built in function to perform the specified operation of queue. It is based on principle that a new Element is inserted in rear of Element of queue is deleted which is at front. It is called FIFO principle

- Queue (): Creates a new Empty queue
- Enqueue (): Insert an Element at the rear of the queue and similar to that of insertion of linked using tail
- Dequeue (): Returns Element which was at the front the front is moved to the successive Element An dequeue operation cannot remove Element if the queue is Empty.

CODE:


```
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:
            self.l[self.r]=data
            self.r=self.r+1
        else:
            print("queue is full")
    def remove(self):
        n=len(self.l)
        if self.r<n-1:
            print(self.l[self.f])
            self.f=self.f+1
        else:
            print("Que is Empty")
q=queue()
q.add(20)
q.add(40)
q.add(50)
```

82.

Q. add (10)
Q. add (70)
Q. add (80)
Q. remove ()
Q. remove ()
Q. remove ()
Q. remove ()
Q. remove ()
Q. remove

ALGORITHM:

STEP1: Define a class queue & assign global variable then define init() method with self argument in init() , assign or initialize the init value with help of self argument

STEP2: Define Empty list and define Enqueue() method with 2 arguments , assign length of Empty list

STEP3: Use if statements that length is equal to rear then queue is full or else insert Element in Empty list or display that queue Element added successfully & increment.

STEP4: Define dequeue() with self argument which then Use if statement that front is equal to length of list then display queue is Empty or else given that front is at 0 & using that, delete element from front side & decrement by 1.

STEP5: Now, call Queue() funct. & give Element that then to be added in Empty list by using Enqueue() and print the list after adding & same for deleting & display the list after deleting the Element in list.

PRACTICAL NO: 7

Evaluation of positive Expression

AIM: Program on Evaluation of given string by using stack in python Environment ie python.

THEORY: The position expression is free of any parameter. Further we took care of the priorities of the operation in program. Reading the expression is always from left to right in postfix.

ALGORITHM:

STEP1: Define evaluate fd function then create an Empty Stack in python

STEP2: Convert the string to a list by using the string method split.

STEP3: Calculate length of string & print it

STEP4: Use for loop to assign the range of string then give condition using if statement

```
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.pppend(int(b) - int(a))
        Elif k[1] == "*":
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) * int(a))
        Elec:
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) / int(a))
        return stack.pop()
s = "869 * +"
r = evaluate(s)
print("The Evaluated value is:", r)
Print("mahesh Gusnani")
```

8A..

OUTPUT:

>>>
The evaluated value is:62
Mahesh Gurunani

STEP 5: Scan the token list from left to right. If token is an operation, convert it from a string to an integer & push the value onto the 'p'.

STEP 6: If the token is an operator *, /, +, -, ^, it will need 2 operands. Pop the 'p' twice the first pop is 2nd operand and 2nd pop is the first operand

STEP 7: Perform the arithmetic operation push the result back on the 'p'

STEP 8: When the input expression has been completely processed the result is on the stack pop the 'p' and return the value

STEP 9: Print the result of string after evaluation of postfix

STEP 10: Attach output and input of above algorithm

PRACTICAL NO: 8

Aim: Implementation of single linked list by adding node from last position

THEORY: A linked list is a linear data structure stores the element in a node in a [illegible] fashion but not necessarily continuous. The individual element of the linked list is called a node. Node comprises of 2 parts ① Data ② [illegible]. Data stores information of element & next refers to the next node by pointing to it. In case of large list if we add/remove any element from the list, all elements of list has to adjust itself every time one add it. very tedious task so linked list is used to solving this type of [illegible].

ALGORITHM:

STEP 1: Traversing of linked list means visiting all the nodes in linked list in order to program some operations on them.

STEP 2: The entire linked list means can be accessed using first node of list. The first node of list in turn is referred by head pointer of the list.

CODE:

```
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 add B (self, item)
               new node = 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()
print ("Mahesh Gurunani")
```

OUTPUT:

```
>>>
20
30
40
50
60
70
80
Mahesh Gurunani
>>>
```


STEP3: Thus, Entire list can be transversed using node which is referred by head pointer of the linked list

STEP4: Now that we know that we can transverse the entire linked not using head pointer, we should only use it to refer the first node of list only.

STEP5: We should not use head pointers to transverse entire linked list because the head pointer is our only reference to 1st node in the linked list, modifying reference of the head pointer can lead to changes which we cannot revert back.

STEP6: We may lose the reference to the ~~last~~ 1st node in our linked list & hence most of our linked list so in order to avoid making some unwanted changes to the 1st node, we will use a temporary node to transverse the entire linked list

STEP7: We will use this temporary node as a copy of the node we are currently transversing. Since we are making currently making a copy of temporary node the data type of the temporary node should also be node

STEP8: Now that current is regressing to the first node, if we want to access 2nd node of list before it can next node of the 1st node

```
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 & j < n2:
        if L[i] <= R[j]
            arr [k] = R[i]
            i += 1
        Else:
            arr [k] = R[j]
            j += 1
            k += 1
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
```

PRACTICAL NO: 9

53

MERGE SORT:

AIM: implementation of merge sort by using python.

THEORY: Merge sort is a divide & conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge (arr, l, m, r) arr [m+1, n] are sorted and merges the two sorted sub-array.

ALGORITHM:

STEP 1: The list is divided into left & right in each recursive call until two adjacent elements are obtained.

STEP 2: Now begins the sorting process. The i and j iterator traverses the two halves in each divided. The K iterator traverses the whole list & makes changes as per the rule.

STEP 3: If the value at j, L[i] is merged to the arr [i+j] sort & is incremented if not the R[j] is closed.

STEP 4: This way, the values being assigned through L[i] are all sorted

STEP 5: At the end of this loop, one of the halves may not have been traversed completely remaining slots in the list.

STEP 6: Thus the merge sort has been implemented


```
while i < n2:
    arr[k] = R[j]
    j += 1
    k = 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, 38, 86, 98, 42]
print(arr)
n = len(arr)
mergesort(arr, 0, n-1)
print(arr)
```

OUTPUT:

[12, 23, 34, 56, 78, 45, 86, 98, 42]

[12, 23, 86, 56, 42, 45, 78, 86, 98]

```
set1 = set()
set2 = set()
for i in range (8,15):
        set1.add(i)
for i in range (1,12):
        set2.add(i)
Print ("set 1: " set1)
Print ("set2: ", set2)
Print ("\n")
set3 = set1 / set2
Print ("union of set1 & set2 : set3"
set4 = set1 /&set2
Print ("Intersection of set1and set2 : set4 ", set4)
Print ("\n")
if set3 >=set4:
  Print (" Set is superset of set 4")
  elif set3 * set4:
     Print ("set 3 is subset of set 4")
  else:
     Print ("set 3 is sub set of")
  if set4 <set3:
     Print ("set 4 is subset of set")
     Print ("\n")
```

PRACTICAL NO: 10.

AIM: Implementation of sets using python

ALGORITHM:

STEP 1: Define two empty set as set1 and set2 now use for statement provision the range of above 2 sets

STEP 2: Now add() method is used for addition the element according to given range then print the sets for addition

STEP 3: Find the union & intersection of above 2 sets by using &, | method print the sets of union & intersection of set3.

STEP 4: Use if statement to find out the subset & superset of set 3 and set4. Display the above set

STEP 5: Display the element in set 3 is not in set 4 using mathematical operation

STEP 6: Use the disjoint() to check that anything is common or element is present or not. If not then display. Thus it is mutually exclusive and then display.

STEP 6: Use clear() to remove or delete the set print the set after clearing the elements passed to the set


```
if set 4 < set 3:
    print ("set 4 is subset of set 3")
    print ("\n")
set 5 = set 3 - set 4
print ("Elements in set 3 and not in set 4: set 5", set 5)
print ("\n")
if set 4. is disjoint (set 5):
    print ("set 4 and set 5 are mutually exclusive \n")
set. clear()
print ("set After applying clear, set 5 is empty set:")
print ("set 5 =", set 5)
```

OUTPUT:

set1 : {8, 9, 10, 11, 12, 13, 14}

set2 : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

union of set 1 & set 2: set 3

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}

Intersection of set1 and set2 {8, 9, 10, 11}

set 3 is superset of set4

Elements in set3 and not in set4 : set5

{1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 14}

set4 & set5 are mutually exclusive

after applying clear set 5 is empty set

set 5 = set()

#CODE:

```
Class node :
    def __init__ (self, value):
        self.left = None
        self.val = value
        self.right = None

Class BST:
    def __init__ (self):
        self.root = None
    def add (self, value):
        p = node(value)
        if self.root == None:
            self.root = p
            Print ("Root is added succesfully", p.val)
        else:
            h = self.root
            if p.val < h.val:
                if h.val = None:
                    h.left = p
                    Print (p.val "None is added succesfully)
                    break;
                else:
                    h = h.left
            else:
                if h.right = None:
```


AIM: Program based on binary search tree by implementing inorder preorder & postorder traversal

THEORY: Binary tree is a tree which supports maximum of 2 children for any node within the tree. Thus any particular node can have either 0 or 1 or 2 children there is another identify of binary tree that it is ordered such that one child is identified as left child and other as right child.

- INORDER: 1) Traverse the left subtree. The left subtree inturn might have left and right subtree.
- PREORDER: 1) Visit the root node traversed the left subtree and right subtree traverse the right subtree
- POSTORDER: Transverse the left subtree the left subtree inturn might have left & right subtree

ALGORITHM:

STEP1: Define class node & define init() with 2 arguments Initialize the value in this method

STEP2: Again, define a class BST that is binary search tree with init() with self argument & assign the root is none

STEP3: Define add() for adding the node. Define a variable p that p=node (value)

STEP4: Use if statement for checking the conditional that root is none then use the else statement for if node is less than the main node then put or arrange them in leftside

STEP5: Use while loop for checking the node is less than or greater than the main node & break the loop if it is not satisfying.

STEP6: Use if statement within that else statement for checking that node is greater than main root then put it into rightside


```
            h.right=p
            print (p.val; "node is added to rightside successfully")
            break
        else:
            h=h.right
def Inorder (root):
    if root==None:
        return
    else:
        Inorder(root.left)
        Print (root, val)
        Inorder (root, right)

def Preorder (root):
    if root ==None:
        return
    else:
        print (root.val)
        Preorder (root.left)
        Preorder (root.right)
def postorder (root):
    if root ==None:
        return
```

else: postorder(root.left)
postorder(root.right)
print(root.val)

t = BST()

OUTPUT:

```
>>> t.add(1)
root is added succesfully
>>> t.add(2)
2 node is added to rightside succesfully
>>> t.add(4)
4 node is added to rightside succesfully
>>> t.add(5)
5 node is added to rightside succesfully
>>> t.add(3)
3 node is added to leftside succesfully
>>> print("In Inorder:", Inorder(t.root))
Inorder:
1
2
3
4
5
Inorder: None
```


STEP7: After this, left side tree & right subtree repeat this method to arrang the node accordingly to binary search tree.

STEP8: Define Inorder(), preorder() & postorder() with root as argument & use if statement that root is none & return that all.

STEP9: In order, else statement used for giving that condition it first left, root & then right node.

STEP10: For preorder we have to give condition in else that first root, left & the right node.

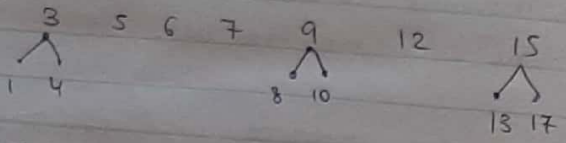
STEP11: For preorder in else post assign left & right & root

STEP12: Display the output & input

INORDER : (LVR)

STEP1 :

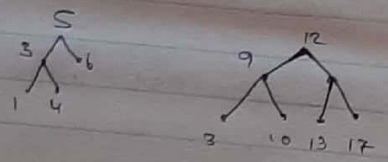
STEP 2:



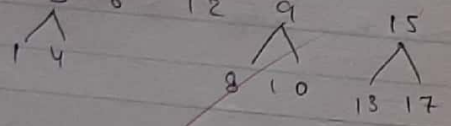
STEP 3: 1 3 4 5 6 7 8 9 10 12 13 15 17

• PREORDER : (VLR)

STEP 1 : 7



STEP 2 : 7 5 3 6 12 9 15



STEP 3 : 7 5 3 1 4 6 12 9 8 10 15 13 17

>>> Print ("In Preorder:", Preorder(root)

Preorder:

1
2
3
4
5

Preorder : None

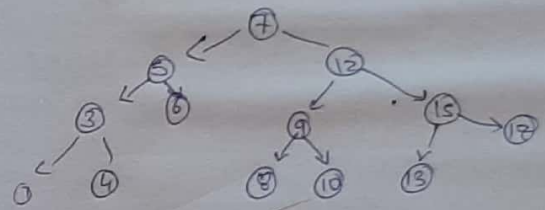
>>> Print ("Postorder:", postorder(T.root)

3
5
4
2
1

postorder : None

* Binary tree :

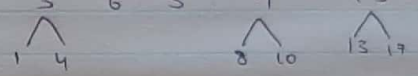
L> search



POSTORDER : (LRV)

STEP 1 :

STEP 2 : 3 6 5 9 15 12 7



STEP 3 : 1 4 3 6 5 8 10 9 13 17 15