Theory:

Linear search is one of the simplest searching algo... in which each item is subsequentially matched. ... each item in the list

It is worst searching algorithm with worst case time ... complexity. It is a force approach. On the other hand ... incase of an ordered list, instead of searching the ... in sequence. A binary search is used which will start ... by examining the middle term linear search is a tech... to compare each other and every element with the key ... element to be found is both of them matches found ... its position is also found.

PRACTICAL 1 :

Aim : Linear Search

Sorted array

. Algorithm :

Step 1 : Define a function with two parameters. Use for conditional statement with range i.e. 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 elements in the array.

Step 3 : If the condition in step 2 satisfies, 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 arrays from user. Now use split() method 4 to split the values.

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

Step 8: Again initialize a variable to call the def function.

Step 9: Use if condition statement to check if the variable in step 8 matches with the element you want to find then print. The index corresponds to the element. If the condition doesn't satisfy then print that element is not found.


```
# 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("elements 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 elements 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 1 4
>>> Elements in array are : [1,2,3,4,5]
>>> Enter element to be searched : 6
    Element not found.
```

```
# unsorted array :
    def linear(arr,n):
        for i in range(len(arr)):
            if arr[i] == n :
                return i

    inp = input("Enter elements in array :").split()
    array = []
    for ind in inp:
        array.append(int(ind))
    print("Elements in array are :",array)
    n1 = int(input("Enter the elements to be searched :"))
    n2 = linear(array,n1)
```


Q.2] Unsorted array :
Algorithm :

Step 1 : Define a function with two parameters use for conditional statement with range i.e. length of array to find . index

Step 2 : Now use if conditional statement to check whether the given statement is equal to the elements 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 then get out of loop.

Step 4 : Now initialize a variable to enter elements in the arrays from user now use split() method & to split the values.

Step 5 : Now initialize a variable as array i.e empty.

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

Step 7 : Now again initialize another variable to ask user to find the element in the array.

Step 8 : Again initialize a variable to call the index function.

Step 9 : Use of conditional statement to check if the variable in step 8 matches with the element you want to find then print the index corresponding to element. If the condition doesn't satisfies then print that element is not found.


```
of n2 == n1 :
    print ("Element found at location", n2)
else :
    print ("Element not found")
```

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

```
# binary search:
# Code :
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 2 : 041

Aim : Binary Search

Algorithm :

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

Step 2 : Use if conditional statement to determine at which position the mid value should point.

Step 3 : If the condition doesn't 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 function.

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

No.
7
8
9

Theory:

Binary search is also known as half interval search, logarithmic search or binary is a search algorithm that finds the position of a 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. This can be avoided by using binary fashion search.

>>> Enter the elements in array:
3 5 10 12 15 20
>>> Element to be searched: 12
12 elements was found at index: 3
>>> Enter the elements in array:
30 1 5 6 7 8
>>> Elements to be search: 2
Element was not found.

```
# 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-1):
        if arr[i] < arr[j]:
            temp = arr[j]
            arr[j] = arr[i]
            arr[i] = temp
print("Elements of array after bubble sort:",arr)
```

```
>>> Enter elements : 2 3 6 1 8 5
>>> Elements of array before sorting :
        [2,3,6,1,5]
>>> Elements of array after sorting :
        [1,2,3,5,6]
```

PRACTICAL : 3

043

Aim : Implementation of Bubble Sort Program on given list

Theory : Bubblesort is based on the idea of repeatedly comparing pairs of adjacent elements an then swapping their position if the simplest form the sorting available. In this, we sort the given element in ascending or descending order by comparing two adjacent element at a time.

Algorithm :

Step 1 : Bubble sort algorithm starts by comparing first two element of an array and swapping of 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 than we do not swap the element.

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

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

Step 6: Stick the output and input of above algorithm of bubble sort stepwise.

Coding:

```
# 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()
```


PRACTICAL : 04 : STACK

Aim : Implementation of stack using Python list

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

Algorithm :

Step 1 : Create a class stack with instance variable items.

Step 2 : Define the init method with self argument and initialize the initial value and then initiate to an empty list.

Step 3 : Define methods push and pop under the class stack.

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

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

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 values, in push method and print the given value.

Step 9: Attach the input and output of above algorithm

Step 10: First condition checks whether the number of elements are zero while the second case whether top is assigned any value. If top is not assigned any value ~~If to~~ then print that the stack is empty.

Output :

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

S-peek()

09/01/24

```
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()
alist1 =[]
for i in I1:
    alist1.append(int(i))
print("Elements input list are:",alist1)
n = len(alist1)
quicksort(alist1,0,n-1)
print("Elements after quick sort are:",alist)
```


PRACTICAL : 05

Aim: Implement Quick sort to sort the given list.

Theory: The quick sort is a recursive algorithm that based on the divide and conquer technique

Algorithms:

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

Step 2: The ~~position~~ partition process will happen next. It will find the 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 & rightmark at the beginning & end of remaining items in the list. The goal of wrong side with repeat to first value while also converting on the split point.

Step 4: we begin by increasing leftmark use locate a value that is greater than the pivot value, we then decrement rightmark until we find value that is less than the pivot value.

Step 5: At the point where right mark becomes less than leftmark we 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 all less than pivot value & all the items to left to the right of split point are greater than pv. The list can now be divided at split point are greater than pv. The list can now be divided at split point & quick sort can be invoked recursively on the two values.

Step 8: The quick sort function invokes a recursive function quick sort helper.

Step 9: Quick sort helper, begins with same base as the message sort.

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 and recursive function.

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

Step 13: Display and stick the coding and output at above algorithm.

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]

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-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()
```


PRACTICAL : 6

Title : Implementing a Queue using Python list.

Theory : Queue is a linear data structure which has 2 references front and rear. Implementing a queue using python list is the simplest as the python list provides in-built functions to perform the specified operation of the queue. It is based on the principal that a new element is inserted after rear and element of queue is deleted which is at front. In simple terms, a queue can be described as a data structure based on first in first out 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 the linked using tail.

Dequeue() : Returns the element which was at the front. The front is moved to the successive element. A dequeue operation cannot remove element if the queue is empty.

Algorithm :

Step 1 : Define a class queue and assign global variables the define init() method with self argument in init(), assign or initialize the initialize value with the help of self argument.

Step 2 : Define a empty list and define enque() with with 2 arguments assign the length of empty list

Step 3 : Use if statement that length is equal to rear then queue is full or else insert the element in empty list or display that queue element added successfully & increment by it.

Step 4 : Define queue() with self argument use if statement that front is equal to length of list then display queue is empty or else give that front is at 0 & using that queue delete the element from front side & increment it by 1.

Step 5 : Now all the queue() function & give the element that has to be added in the empty list by using enqueue() & print list after adding same & for deleting.

Output :

```
>> q.add(30)
>> q.add(40)
>> q.add(50)
>> q.add(60)
>> q.add(70)
>> q.add(80)
>> q.add(90)
   Queue is full
>> q.remove()
   30
>> q.remove()
   40
>> q.remove()
   50
>> q.remove()
   60
>> q.remove()
   70
>> q.remove()
   80
>> q.remove()
   Queue is empty
```

```
# Code:
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))
```


PRACTICAL : 7

Aim : Program on Evaluation of given string by using stack in python environment i.e Postfix.

Theory : The postfix expression is free of any parenthesis. Further we took care of the priorities of the operators in the program. A given postfix expression can easily be evaluated using stacks.

Algorithm :

Step 1 : Define evaluate as function then create a empty stack in Python.

Step 2 : Convert the string to a list by using the string method 'split'.

Step 3 : Calculate the length of string & print it.

Step 4 : Use for loop to assign the range of string then give condition using if statement.

Step 5 : Scan the token list from left to right. If token is an operand convert it from a string to an integer & push the value onto the 's'.

Step 6: If the token is an operator *, /, +, -, ^, it will need two operands. Pop is second operand & the second pop is first operand.

Step 7: Perform the arithmetic operation. Push the result back on the 'm'.

Step 8: When the input expression has been completely processed the result is on the stk

Step 9: Print the result of string after the evalu of postfix.

Step 10: Attach output & input of above algorithm

```
        else:
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b)/int(a))
    return stack.pop()
s = "89*+"
r = evaluate(s)
print("The evaluated value is:",r)
```

Output:

The evaluated value is: 62

```
class node:
    global data
    global nest
    def __init__(self, items):
        self.data = items
        self.nest = None

class linked list:
    global s
    def __init__(self):
        self.s = None
    def add(self, item):
        newnode = node(item)
        if self.s = None:
            self.s = newnode
        else:
            head = self.s
            while head.nest != 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 = new node
```


PRACTICAL : 8

Aim: Implementation of single linked list by adding the nodes from position.

Theory: A linked list is a linear data structure which stores the elements in a node in a linear fashion but no necessarily contigous. The individual elements of the linked list called a Node. Node comprises of 2 parts ① Data ② Nest. Data stores all the information w.r.t the element for example roll.no, address, etc. whereas nxt refers to the next node. In case of larger list, if we add/remove any element from the list, all the elements of list all the elements of list has to adjust itself every time we add it is very tedious task linked list is used to solving this type of problems.

Algorithm

Step 1: Traversing of a linked list means visiting all the nodes in the linked list in order to perform some operation on them.

Step 2: The entire linked list has been accessed used the first node of the linked list. The first node of the linked list in turn is referred by the head pointer of the linked list.

Step 3: Thus the entire linked list can be transversed using the node which is referred by the head pointer of the linked list.

Step 4: Now that we know that we can transverse the entire linked list using the head position we should only use it to refer the first node of list only.

Step 5: We should not use the head pointer to transverse the entired linked list because the head pointer is our only reference to the 1st node in the linked list modifying the reference of the head pointer can head to changes which we cannot revert back.

```
def display(self):
    head = self.s
    while head.next ≠ None:
        print(head.data)
        head = head.next
    print(head.data)
start = linked.list()
```

Output:

```
>>> start.addL(80)
>>> start.addL(70)
>>> start.addL(10)
>>> start.addL(50)
>>> start.addL(40)
>>> start.addL(30)
>>> start.addL(20)
>>> start.display()
```


Step 6: We may lose the reference to the 1st node in the linked list and hence most of our linked list. so in order to avoid making some unwanted changes to the 1st node, we will use temporary node to transverse the entire linked list.

Step 7: We will use this temporary node as a copy of the node we are currently traversing. Since we are making temporary node a copy of current node the datatype of temporary node should also be node.

Step 8: Now the current node is referring to the first node if we work to access 2nd node of test we need to refer it as next node of 1st node.

PRACTICAL : 09

Aim : Implementation of mergesort by using Python.

Theory : Merge sort is a divide & conquere algorithm. It divides input into two halves with it recursively for the two halves & the merge the two sorted halves. The merge function is used for merging two halves.

Algorithm :

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

Step 2 : Now begin the sorting process. The i, j iterators traverse the two halves in each call. The k iterator traverse the whole lists & makes changes along the way.

Step 3 : If the value at i is smaller than the value at L(i) is assigned to the arr[i+1] slot & i is incremented. If not then R[j] is chosen.

Step 4 : This way, the values being assigned through L[i+1] are all sorted.

Code :

```
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[i + l]
    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 = i + 1
        else:
            arr[k] = R[j]
            j = j + 1
            k = k + 1
    while i < n1:
        arr[k] = L[i]
        i = i + 1
        k = k + 1
```

```
while j<n2:
    arr[k] = R[j]
    j+=1
    k = k+1
def merge sort (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, 22, 34, 56, 78, 45, 86, 98, 42)
print (arr)
n = ln(arr)
mergesort (arr, 0, n-1)
print(arr)
```

Output :

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

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

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.

T2(1)

PRACTICAL : 10

Aim : Implementation of sets using Python.

Algorithm :

Step 1: Define two empty set as set 1 & set 2. Now use set for statement providing the range of above 2 sets.

Step 2: Now add() is used for addition of the element according to given range then print the sets for addition.

Step 3: Find the union & intersection of above 2 sets by using to method. print the sets of union & intersection of set 3.

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

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

Step 6: Use linear() to remove or delete the sets & print the set after clearing the element present in the set.

#Code :

```
set 1 = set()
set 2 = set()
for i in range(8,15):
    set 1.add(i)
for i in range(1,12):
    set 2.add(i)
print("set 1:", set 1)
print("set 2:", set 2)
print("\n")
set 3 = set 1 / set 2
print("union of set 1 and set 2 : set 3", set 3)
set 4 = set 1 & set 2
print("Intersection of set 1 & set 2 ; set 3", set 4)
print("\n")
if set 3 > set 4:
    print("set 3 is superset of set 4")
elif : set 3 < set 4:
    print("set 3 is subset of set 4")
else:
    print("set 3 is same as set 4")
if set 4 < set 3:
    print("set 4 is subset of set 2")
set 5 = set 3 - set 4
print("Elements in set 3 & not in set 4 ; set 5", set 5)
print("\n")
```

```
if set4.isdisjoint(set5):
    print("set 4 & sets are mutuald. F exclusive In")
set.clear()
print("After applying clear, set 5 is empty set:")
print("set 5:", set5)
```

Output:

set 1 = {8, 9, 10, 11, 12, 13, 14}

set 2 = {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 set 1 & set 2 {8, 9, 10, 11}

set 3 is superset of set 4

Elements in set 3 & not in set 4; set5 {1, 2, 3, 4, 5, 6, 7, 12, 13, 14}

set 4 & set 5 are mutually exclusive after applying clear, set 5 is empty

set 5 = set()

PRACTICAL : 11

Aim : Program based on binary search tree by implementing inorder, preorder & postorder transversal.

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 identified as left child & other as right child.

Inorder : Transverse the left subtree. The left subtree inturn might have left & right subtrees.

Preorder : visit the root node traverse the left subtree traverse the right subtree.

Postorder : Traverse the left subtree. The left subtree inturn might have left.

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 successfully",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")
                else:
                    h = h.left
                if h.right = None:
                    h.right = p
                    print(p.val,"None is added to right side 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()
```


Algorithm :

Step 1 : Define class node & define init() with 2 arguments. Initialize the value in this method.

Step 2 : Again, define a class BST that is binary search tree, with init() with self argument & assign the root is none.

Step 3 : Define add() for adding the node. Define a variable p that p = node(value)

Step 4 : Use if statement for checking the conditional that root is none then use else statement for if node is less than the main node then put on arrange them in leftside.

Step 5 : Use while loop for checking the node is less than or greater than the main node & break the loop if it is not satisfying.

Step 6 : Use if statement within that else statement for checking that node is greater than main root then put it into right side.

Step 7 : After this, left side tree & right subtree repeat this method to arrange the node according to binary search tree.

Step 8: Define Inorder(), preorder() & postorder() with root as argument, use of statement that root is none & return that all.

Step 9: In-order, else statement used for getting that condition if first left, root & then right node.

Step 10: For pre-order we have to give condition in else that first root, left & the right node.

Step 11: For post-order in else part assign left & right & root

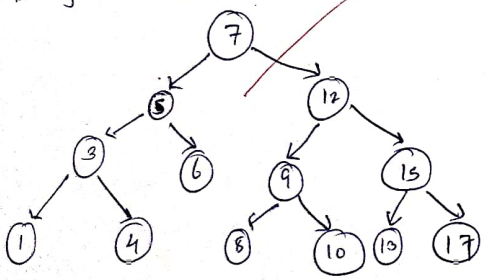
Step 12: Display the output & input.

Output:

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

```
>>> print("In Preorder :", Preorder(t.root))
    preorder :
    1
    2
    4
    3
    5
    Preorder : None
>>> print("Post order :", postorder(t.root))
    postorder :
    3
    5
    4
    2
    1
    post order : None
```

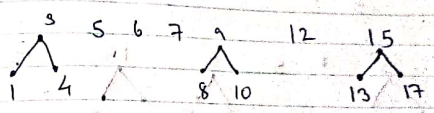
* Binary search Tree :



Inorder : (LVR)

Step 1 :

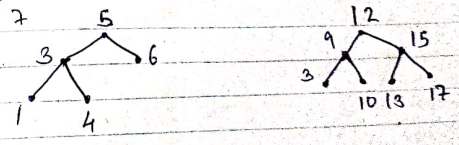
Step 2 :



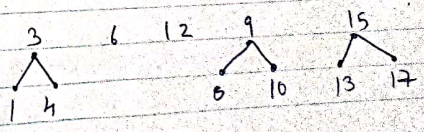
Step 3 : 1 3 4 5 6 7 8 9 10 12 13 15 17

Preorder (VLR)

Step 1 :



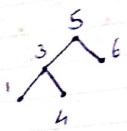
Step 2 : 7 5 3 6 12 9 15

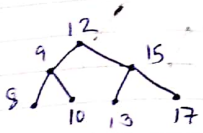


Step 3 : 7 5 3 1 4 6 12 9 8 10 15 13 17

- Postorder: (L & V)

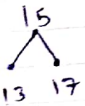
Step 1:





Step 2:

6 5



12 7

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