Theory:

Linear search is one of the simplest searching algorithm in which each item is sequentially matched with 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 list in sequence. A binary search is used which will start by examining the middle term. Linear search is a technique to compare each and every element with the key element to be found if both of them matches found and its position is also found.

Practical 1:

Aim: Linear search

a) Unsorted array:

. Algorithm:

Step 1: Define a function with two parameters. Use for conditional statement with range ie 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 initialise a variable to enter elements in the array from user. Now use split() method & to split the values

Step 5: Now use for 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 user to find the element in the array.

Step 8: Again initialize a variable to call the defined 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 corresponding to the element. If the condition doesn't satisfies then print that element is not found.

Unsorted array:
CODE:

```
def linear(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i

inp = int(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")
```

01

>>> Enter elements in array: 1 2 3 5 4

>>> 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, 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.x1 = int(input("Enter
  the elements to be searched:")
x2 = linear(array, x1)
```

23. unsorted array:

Algorithm:

step1: Define a function with two parameters use for conditional statement with range ie length of array to find index

step2: Now use if conditional statement to check whether the given statement is equal to the elements in array.

step3: If the condition in step2 satisfies return the index no of the given array. If the condition doesn't satisfies then get out of loop.

step4: Now initialise 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 ie empty.

Step 6: Now use for conditional elem 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 defined function

Step 9: use if 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

```
if x2==x1:
        print("element found at
            location", x2)
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 found at location 2
>>> Element in array: 2 4 5 3 1
>>> Elements in array are : [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:").split()
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 found at index", index)
```

Practical 2:

Aim: Binary search

Algorithm:

Step 1: Define a function with two parameters. Now initialize variable with 0. use while conditional statement 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 initialise a variable to find the element in array.

Step 7: Now initialise a variable to call the defined function

Step 8: If the NOW use if condition to determine the index value and print the index value.

- 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 search: 12
12 Element was found at index 3

>>> Enter the elements in array:
-3 0 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 them:", 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("Element 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:

Aim: Implementation of Bubble sort program on given list

Theory: BubbleSort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their position if the simplest form of sorting available in this, we sort the given element in ascending or descending order by comparing two adjacent elements 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 first element is smaller than second element then 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 mentioned to get the required result.

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

#CODE:

```
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]=dat
    def pop(self):
        if self.tos<0:
            print("stack is
                empty")
        else:
            k=self.l[self.tos]
            print("data=",k)
            self.tos[self.tos]=0
            self.tos=self.tos-1
s=stack()
```

Practical 4:

Aim: Implementation of stack using python list

Theory: A stack is a linear data structure can be represented in the real world in the form of a physical stack or a pile. The elements in the stack are added or removed only from one position ie the topmost position. Thus the stack works on the LIFO principle as the element that was inserted last will be removed first. A stack can be implemented using array as well as linked list. Stack has three basic operation: push, pop, peek. The operations of adding and removing the elements is known as Push & pop.

Algorithm:

Step1: Create a class stack with instance variable items.

Step 2: Define the init method with self argument and initialize the initial variable and then initialize 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: Or else print statement as insert 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() value is less then 1 then return the stack is empty or delete the element from stack

output :

```
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.push(50)
>>> s.push(60)
>>> s.push(70)
>>> s.push(80)
stack is full
>>> s.pop()
 data = 70
>>> s.pop()
 data = 60
>>> s.pop()
 data = 50
>>> s.pop()
 data = 40
>>> s.pop()
 data = 30
```

```
>>> s.pop()
data = 20
>>> s.pop()
data = 10
>>> s.pop()
Stack is empty
```

at topmost position

Step 8: First condition checks whether the no of elements are zero while the second case whether tos is assigned any value. If tos is not assigned any value then we can be sure that stack is empty.

Step 9: Assign the element values in push method to add and print the given value is poped.

Step 10: Attach the input and output of above algorithm.

11/01/2021

Practical No 5

Aim: Implement Quick sort to sort the given list

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

Algorithm:

Step1: Quick sort first selects a value, which is called pivot value first value element. serve 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 the split point and at the same time move other items to the appropriate side of the list either lets then organise that pivot value

Step 3: Partitioning begins by locating two position marked lets call them eliminate and right mark at the beginning and end of

CODE

```
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)
x1 = input("enter elements in the list.
                                split()
alist1 = []
print("elements in list are: ", alist1)
n = len(alist1)
quicksort(alist, 0, n-1)
print("elements after quick sort is ",
                                alist1)
```

02

Output:

Quick sort

enter elements in list 21 20 22 30

elements after quick sort are

20 21 22 30

remaining items in the list. The goal of the partition process is to move items in the list

step 4: We begin by increasing leftmark until we locate a value that is greater than the pv we then decrement right mark until we find value that is less than the pv

step 5: At the point where right mark becomes less than leftmark we stop.

step 6: The pv can be exchanged with the content of split point and pv is now in place.

step 7: In addition all the items to left of split point are less than pv

step 8: The quicksort function invokes a recursive function quick sort helper

step 9: Quick sort helper begins with some base as merge sort

Step 10: Display & stick the coding & output of above algorithm

52 Practical No 6

Title: Implementing a queue using python list

Theory: Queue is a linear data structure which has 2 references front & rear. While implementing a queue using python list is the simplest as the python list provides inbuilt functions to perform the specified operations of the queue. It is based on the principle that a new element is inserted after rear and element of queue is deleted which is at front. In simple term, a queue can be described as a data structure based on first in first out.

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 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.

#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()
```

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
```


Algorithm:

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

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

Step 3: Use if statement that length is equal to size then queue is full or else insert the element in empty list or display that queue element added successfully & increment by 1.

Step 4: Define dqueue() 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 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

Practical No 8

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

Theory: A single list is a linear data structure which is storing the elements in a node in a linear fashion, but no necessarily continues. The individual element of the linked list called as a node consists of 2 parts ① data ② next. Data stores all the information with the element whereas next refers to the next node.

Algorithm:

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

Step 2: The entire linked list means can be accessed as the first node of the linked list.

Step 3: Thus the entire linked list can be traversed using the node which is referred by the head pointer of linked 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 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()
```

output:

```
>>> start.addL(80)
>>> start.addL(70)
>>> start.addL(60)
>>> start.addL(50)
>>> start.addL(40)
>>> start.addB(30)
>>> start.addB(20)
>>> start.display()
```


step4: Now that we know that we can traverse the entire linked list using the head pointer, we should only use it to reflect the first node of list only.

step5: We should not use the head pointer to traverse the entire linked list because the head pointer is our only reference to 1st node

step6: We may lose the reference to 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 temporary node to terminate the entire linked list

step7: 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.

step8: Now that current is referring to the first node if we want to access 2nd node of list we need to refer it as next node of 1st node

Step 9: But the 1st node is referred by current so we can traverse to 2nd nodes as h = h.next

Step10: Similarly we can traverse rest of nodes in the linked list

Step11: Our concern now is to find dominating condition for the while loop

Step12: The last node in the linked list is referred by the tail of linked list

Step13: So we can refer to last node of linked list

Step14: We have to now see how to start traversing the linked list & how to indentify whether the we have reached the last node

Step15: Attach the coding of input & output of above algorithm

```
>>> 20
30
40
50
60
70
80
90
```

#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 No 7

Aim: Program on Evaluation of given string by using stack in Python environment ie 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 stack. Reading the expression is always from left to right in Postfix

Algorithm:

Step1: Define evaluate as function then create a empty stack in Python.

Step2: Convert the string to a list by using the string method 'split'

Step3: Calculate the length of string & print it

Step4: Use for loop to assign the range of string then give condition using if statement

step5: 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 'p'

step6: If the token is an operator *, /, +, -, it will need two operands. Pop the 'p' twice. The first pop is second operand & the second pop is the first operand

step7: Perform the arithmetic operation. Push the result back on the 'm'

step8: When the input expression has been completely processed the result is on the value.

step9: Print the result of string after the evaluation of postfix

step10: Attach output & input of above algorithm


```
else:
    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)
```

output:

The evaluated value is: 62

#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[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=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
```

Practical 9:

Aim: Implementation of mergesort by using python

Theory: Merge sort is a divide and conquer algorithm. It divides input array into two halves, calls itself for the two halves & the merge the two sorted halves. The merge() function is used for merging two halves.

Algorithm:

Step1: The list is divided into left & right in each recursive call until two adjacent elements are obtained

Step2: Now begins the sorting process. The i & j iterators traverse the two halves in each call. The k iterators traverse the whole list & makes changes along the way.

Step3: If the value at i is smaller than the value at j L(i) is assigned to the arr[k+1] slot & i is incremented. If not then R[j] is chosen

Step 4: This way, the values being assigned through h[j+1] 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 j<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, 78, 45, 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, 56, 56, 42, 45, 78, 86, 98]
```

```
# CODE:
set1 = set()
set2 = set()
for i in range(8,15):
    set1.add(i)
for i in range(1,12):
    set2.add(i)
print("set1:", set1)
print("set2:", set2)
print("\n")
set3 = set1 | set2
print("union of set1 and set2 : set3",
                                set3)

setH = set1 & set2
print("intersection of set1 & set2 :
                    setH", setH)

print("\n")
if set3>setH:
    print("set3 is superset of setH)
elif: set3<setH:
    print("set3 is subset of setH")
else:
    print("set3 is same as setH")
```

Practical 10:

Aim: Implementation of sets using python

Algorithm:

step1: Define two empty set as set1 and set2 now use for statement providing the range of above 2 sets

step2: Now add() method is used for addition the element according to given range then print the sets for addition

step3: Find the union and intersection of above 2 sets by using & | method Print the sets of union & intersection of set3.

step4: Use if statement to find out the subset and superset of set3 and setH. Display the above set

step5: Display that element in set3, is not in setH using mathematical operation.

Step 6: Use clear() to remove or delete the sets and print the set after clearing the elements present in the set.

```
if set4 < set3:
    print (" set4 is subset of set3")
    print ("\n")
set5 = set3 - set4
print (" Elements in set3 and not in
        set4: set5", set5)
print ("\n")
if set4.isdisjoint(set5):
    print ("set4 and set5 are mutuall
        y exclusive \n")
set.clear()
print ("set After applying clear, set5
        is empty set:")
print ("set5=", set5)
```


output:

set1 : {8,9,10,11,12,13,14}
set2 : {1,2,3,4,5,6,7,8,9,10,11}

union of set1 & set2 : set3
{1,2,3,4,5,6,7,8,9,10,11,12,13,14}

Intersection of set1 and set2 {8,9,10,11}
set3 is superset of set4
elements in set3 and not in set4 : set5
{1,2,3,4,5,6,7,12,13,14}
set4 & set5 are mutually exclusive
after applying clear, set5 is empty set
set5 = 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 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 successfully")
                    break
                else:
                    h = h.left
            else:
                if h.right = None:
```

Practical 11

63

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 binary tree can have either 0, 1 or 2 children. There is another identity 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 in turn might have left and right subtrees.
- Preorder: 1] Visit the root node traverse the left subtree and right subtree. Traverse the right subtree.
- Postorder: Traverse the left subtree the left subtree in turn might have left & right subtrees.

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 else statement for if node is less than the main node then put on arrange them in left side

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.

```
                h.right = p
                print(p.val, "None is added to right side successfully")
            else:
                break
        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 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("\nInorder:", Inorder(t.root))
Inorder:
1
2
3
4
5
Inorder: None
```


step6: Use if statement within that else statement for checking that node is greater than main root then put it into rightside

step7: After this, left side tree & right subtree, repeat this method to arrange the node according 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 use statement used for giving that condition if root, left, root & then right node

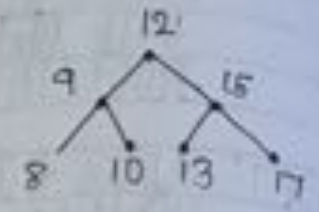
step10: For postorder, we have to give condition in else that like root, left & the right node

step11: For preorder, in else part as left & right & root

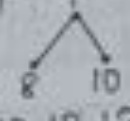
step12: Display the output & input

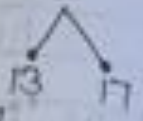
• INORDER: (LVR)

step1: 7



step2: 3 5 6 7 9 12 15

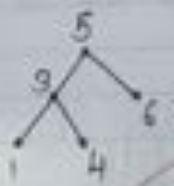


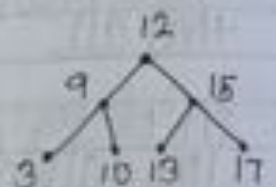


step3: 1 3 4 5 6 7 8 9 10 12 13 15 17

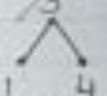
• PREORDER: (VLR)

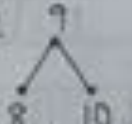
step1: 7

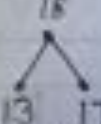




step2: 7 5 3 6 12 9 15



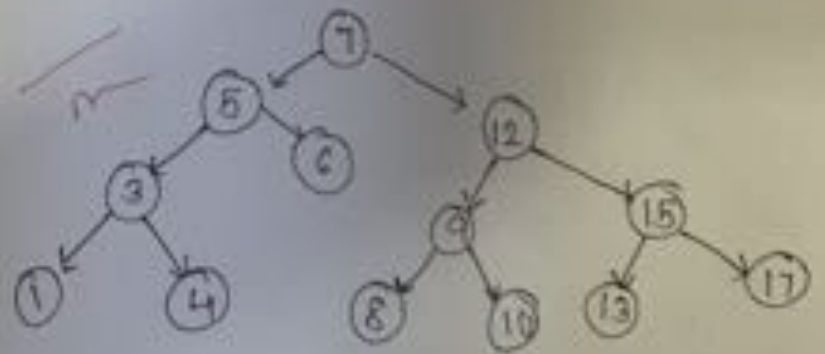




step3: 7 5 3 1 4 6 12 9 8 10 15 13 17

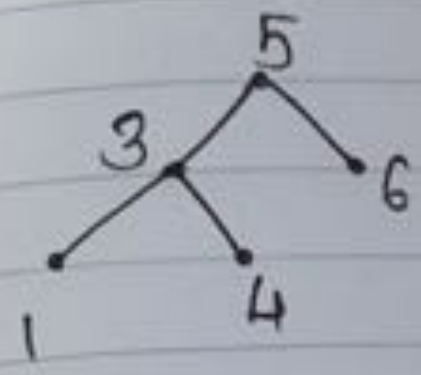
```
>>> print ("In Preorder:", Preorder(t.root))
preorder:
1 2 4 3 5
Preorder : None
>>> print ("Postorder:", postorder(t.root))
3 5 4 2 1
postorder: None
```

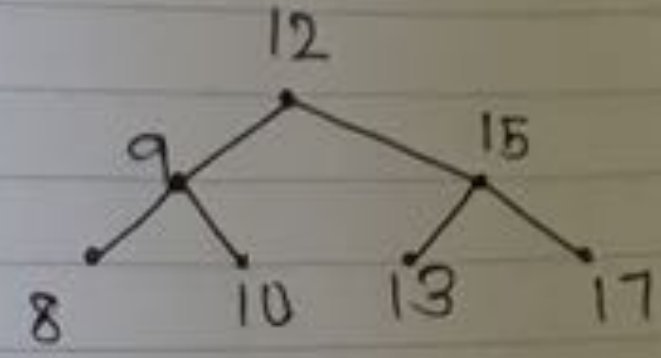
* Binary tree: search



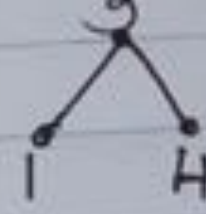
. POSTORDER: (LRV)

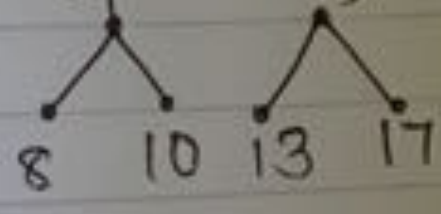
step 1:





step 2: 3 6 5 9 15 12 7





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

20/02/20