PRACTICAL-01

Aim: Implement Linear search to Find an Item in the list.

Theory:

LINEAR SEARCH

Linear search is one of simplest item searching algorithm in which targeted item is sequentially matched with each items in the list.

It is worst ~~situation~~ 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 ordered list. 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 if them matches the algorithm returns that element found and its position is also found.

(1) Unsorted:

Algorithm:

S1: Create an empty list and assign it to a variable.

S2: Accept the total no. of elements to be inserted into the list from the user say 'n'.

S3: Use for loop for adding the elements into the list.

S4: Print the new list.

S5: Accept an element from the user that are to be searched in the list.

S6: Use for loop in a range from '0' to the total no. of elements to search elements from the list.

S7: Use if loop that the elements in the list is equal to the element accepted from user.

S8: If element is found then print statement that element is found along with the element's position.

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

inp = input ("Enter element in array: ")
array = []
for ind in inp:
    array.append (int (ind)):
print ("Element in array are:", array)
    n1 = int (input ("Enter element to be searched
                                        :"))
    n2 = linear (array, n1)
if n2 == n1:
    print ("Element found at location", n2)
else:
    print ("Element not found")
```

```
>>> enter element in array: 3 2 4 5 1
>>> Element in array are: [3 2 4 5 1]
>>> Element to be searched: 4
    Element found at location: 2
```

```
def linear (arr, x):
    for i in range (len(arr)):
        if arr[i] == x:
            return i
inp = (input ("Enter elements in arrange: ")
split ()
array = []
for ind in inp:
    array.append (int (ind))
print ("Elements in array are : ", array)
array.sort ()
x1 = int (input ("element element to be searche
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 5
>>> elements in array : [1 2 3 5]
>>> enter element to be searched : 7
>>> Element not found.

(2) Sorted linear search:

Sorting means to arrange the elements in increasing or decreasing order

. Algorithm:

S1: create empty list and assign it to a variable.

S2: Use Accept the total no. of elements to be inserted into the list from elements in the list.

S3: Use for loop for using append() method to add the elements in the list.

S4: Use sort() method to sort the accepted element and assign in increasing order the list then print the list.

S5: Use if statement to give range in which elem is not found in given range then display "Element not found".

S6: Then use else statement if element is not found in range then satisfy the given cond^{n}.

S7: Use for loop in range from 0 to the table no. of elements to be searched, before doing this accept an search no. from the using Input statements.

S8: Use if loop that the elements in the list is equal to the element accepted from user

S9: If the element is found then print the stmt. that the element is found along with the element position.

S10: Use another if loop to print that the element is not found if the element which is accepted from user is not there in the list.

S11: Attach the input and output of above algorithm.

Thus algorithm

```
a = list (input ("Enter the list of elements: "))
    a = sort()
n = len(a)
s = int (input ("Enter the number to be searc
if (s > a[n-1] or s < a[0]):
    print ("Element not found")
else:
    f = 0
    l = n-1
    for i in range (0, n):
    m = int (lf + l/2)
    if s == a[m]:
        print ("Element is found at :", m)
        break
    else:
        if s < a[m]:
        l = m-1
        else:
            f = m+1
```

>>> Enter the list of elements : 1, 5, 2, 4

>>> Enter number to be searched : 5

>>> The element is found at : 1

PRACTICAL-02

Aim: Binary Search.

Algorithm:

S1: create empty list & assign it to a variable.

S2: Using input method, accept the range of given list.

S3: Use for loop, add elements in list using append -() method.

S4: Use sort() method to sort the accepted elements & assign it in increasing ordered list, print the list after sorting.

S5: Use if statement to give range in which element is found then display a message "element not found".

S6: Then use else statement, if statement is not found in range then satisfy the below condition.

S7: Accept an argument & key of element that element has to be searched.

S9: Use for loop & assign the given range.

S10: If start in list & still the element to be searched is not found then find middle element (m).

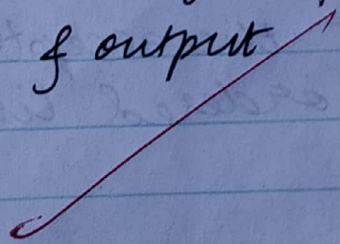
S11: Else if the step to be searched is still less than the middle term then,

Initialize least (h) = mid (m) - 1

Else

Initialize first (l) = mid (m) - 1.

S12: Repeat till you found the element, stick the input & output of above algorithm.



```
inp = input ("Enter a statement:")
a = [ ]
for ind in inp:
    a = append (int (ind))
print ("Element before sorting", a)
n = len (a)
for i in range (0,n):
    for j in range (n-1):
        if a[i] < a [j]:
        tmp = a[j]
        a[j] = tmp
print ("Element after sorting", a)
```

>>> Enter a ~~state~~ element ; 2 5 8 6
elements before sorting [2, 5, 8, 6]
elements after sort [2, 5, 6, 8]

PRACTICAL-03

039

BUBBLE SORT

- **Aim**: Implementation of Bubble sort program on a given list.

- **Theory**: Bubble sort is based on the idea of repeatedly comparing parts of adjacent elements and then swapping their swapping their position if they exist in the wrong order. This is 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**:

S1: Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary.

S2: If we want to sort the elements of array in ascending order then second then, we need the swap element.

S3: If the first element is smaller than second then we do not swap the element.

S4: Again second and those elements are compared and swapped if it is necessary and this process goes on until last & second last element is compared and swapped.

S5: If there are n elements to be sorted, then the process mentioned above should be repeated n-1 times to get the required result.

S6: Stick the output & input of above algorithm of bubble sort stepwise.

041

PRACTICAL-04

STACK

. Aim: Implementation of stack using python list.

. Theory: A stack is a binary data structure that can be represented in the real world in the form of a python physical stack or a pile. The elements in the stack are added or removed only from one position i.e, the top most position. Thus, the stack works on the LIFO (Last In First Out) 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 basic 3 operations: push, pop, peek. The operations of adding and removing the elements the elements is known as Push & Pop.

- **ALGORITHM:**

S1: Create a class stack with instance variable items.

S2: Define the init method with self argument & initialize the initial value and then initialize to an empty list.

S3: Define methods ~~and~~ of push & pop under the class stack.

S4: Use if stmt. to give the condition that length of given list is greater than the range of list then print stack is full.

S5: Or Else print stmt. as insert the element into the stack and initialize the variable.

S6: Push method used to insert the element but pop method used to check & to delete the element from the stack.

S7: If in pop method, value is less than 1 then, to add and print the given value is poped out.

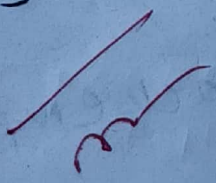
```
print ("sakshi kulkarni")
class stack:
    global tos
    def __init__(self):
        self.l=[0,0,0,0]
        self.tos = -1
    def push(self,data):
        n=len(self.l)
        if self.tos==n-1:
            print ("stack is full")
        else:
            self.tos = self.tos+1
            self.l[self.tos]=data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

s=stack()
```

SPO

output :

sakshikulkarni

```
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.l
[10,20,30,40]
>>> s.pop()
data =40
>>> s.l
[10,20,30]
```



~~S9: Attach the input~~

S9: Assign the element values in push method to add and print the given value is poped out.

S10: Attach the input and output of above algorithm.

S8: First condition checks whether the number of elements are zero while the second case whether tos is designed any value. If tos is not assigned any value, then we can be sure that stack is empty.

02-11-2020

EAD

PRACTICAL-05

Aim: Implement Quick sort to sort the given list.

THEORY: The quick sort is a recurrence algorithm based on the divide & conquer technique.

ALGORITHM:

S1: Quick sort first selects a value, which is called pivot value. Since we know that first will eventually end up as last in that list.

S 2: The position process will happen next. It will find the split point and at same time move other items to appropriate side of list either less than or greater than pivot values.

: Positioning begins by locating two position marks let's call them leftmark and rightmark at the beggining and end of remaining items in the list. The goal at the position process is to move items that are on wrong side with pivot value.

```
def quicksort(alist):
    quicksortHelper(alist, 0, len(alist)-1)
def quicksortHelper(alist, first, last):
  if first < last:
    splitpoint = partition(alist(alist, first, last))
    quicksortHelper(alist, splitpoint+1, last)
def partition(alist, first, last):
    pivotvalue = alist[first]
    leftmark = first+1
    rightmark = last
    done = False
    while not done:
      while leftmark <= rightmark and alist[left
                                mark] <= pivotvalue
      while alist[rightmark] >= pivotvalue
                  and rightmark >= leftmark:
      if rightmark < leftmark
          done = True.
      else:
          temp = alist[leftmark]
          alist[leftmark] = alist[rightmark]
          alist[first] = alist[rightmark].
          return rightmark.
alist = [42,54,45,67,89,66,55,80,100]
quicksort(alist)
print(alist)
```

100

OUTPUT:

>>> [12, 45, 54, 55, 65, 89, 67, 80, 100].

S4: We begin by increasing leftmark until we locate is value that is greater than the p.v., we then decrement right mark until we find value that is less than the p.v.

S5: At the point where right mark becomes less than leftmark we stop. Thus, position of rightmark is now the split point.

S6: The pv. can be exchanged with content of split point and p.v. is now in place.

S7: In addition, all the items to left of split pt. are less than p.v. and all the items to the left to right- at split pt. greater than p.v. The list can now be divided at split pt. and quick sort can be recursively on on 2 values.

S8: Quickest function involves recursive function, quick sort helper.

S9: Quick sort helper, begins with same base as merge sort.

S10: If length of the list is less than 0 or equal one list is already sorted.

S11: It is the greater than it can be position and recursive function.

S12: The ~~partiti~~ position function implement the process described earlier.

S13: Display and stick the cooling and output of above algorithm.

Scanned with CamScanner

TAO

PRACTICAL-06

- **Aim**: Implementing a queue using Python List.

- **Theory**: Queue is a linear data structure which has two references - front and rear.
  Implementing a queue using python list is the simplest as the python list provides built-in functions to perform the specified operations of queue. It is based on principle that a new element is inserted in rear and 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-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.r<n-1:
            print(self.l[self.f])
            self.f=self.f+1
        else:
            print("Queue is empty")
q=Queue()
q.add(30)
q.add(40)
q.add(50)
```

880

Q. add (60)
Q. add (70)
Q. add (80
Q. remove()
Q. remove()
Q. remove()
Q. remove()
Q. remove()
Q. remove()

• ALGORITHM:

S1: Define a class queue and assign global variables then define init() method with self argument in init(), assign or ~~initi~~ initialize the init value with help of self argument.

S2: Define empty list and define enqueue() method with 2 arguments, assign length of empty list.

S3: Use if statement that length is equal to rear than queue is full or else insert element in empty list or display that queue element added succesfully & increment by 1.

S4: Define deQueue() with self argument under this. If Use if statement that front is equal to length of list then display queue is empty or else give that front is at 0 and using that, delete element from front side and increment by 1.

S5: Now, call queue() function & give element that has to be added in the empty list by using enqueue() and print the list after adding and same for deleting and display the list after deleting the element in list.

PRACTICAL-07

Evaluation of a Postfix Expression

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

Theory: The postfix expression is free of any parameters. Further we took care of the priorities of the expression in the program. Reading the expression is always from left to right in Postfix.

Algorithm:

S1: Define evaluate at function then create an empty stack in Python.

S2: Convert the string to a list by using the string method split.

S3: Calculate length of string and print it.

S4: 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.append(int(b)-int(a))
        elif k[i] == '*':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b)* int(a))
        else:
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b)/int(a))
    return stack.pop()
s = "8 6 9 * +"
r = evaluate(s)
print("The evaluated value is:", r)
print("Mahesh Gurnani")
```

Output:

```
>>>
The enamated value is :6 2
Mahesh Gurnani
```

S5: Scan the token list from left to right. If token is an operator, convert it from a string to an integer and push the value onto the 'p' stack.

S6: 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.

S7: Perform the Arithmetic operation. Push the result back on the 'm'.

S8: When the input expression has been completely processed, the result is on the stack. Pop the 'p' and return the value.

S9: Print the result of string after evaluation of Postfix.

S10: Attach output and input of above algorithm.

PRACTICAL-08

- **Aim:** Implementation of single Linked list by adding nodes from pos last position.

- **Theory:** A linked list is a linear data structure which stores the elements in a node in a linear fashion but not necessarily contigious. The individual elements of the linked list called a Node. Node comprises of 2 parts ① Data ② Nent.
  Data stores information of element whereas nent refers to the nent node by pointing towards it. In case of largen list if we add/remove any element from the list, all the elements of list has to adjust itself every time we add, it is very tedious task so linked list is used to solving this type of task.

- **Algorithm:**

S1: Transversing of linked list means visiting all the nodes in linked list in order to perform some operation on them.

S2: The entire linked list means can be accessed using first node of list. The first node of list in turn is refered 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 addB(self, item):
        newnode = node(item)
        if self.s == None:
            self.s = newnode
        else:
            newnode.next = self.s
            self.s = newnode
```

```
def display(self):
    head = self.s
    while head.next != None:
        print(head.data)
        head = head.next
    print(head.data)
start = linkedlist()
start.addL(50)
start.addL(60)
start.addL(70)
start.addL(80)
start.addB(40)
start.addB(30)
start.addB(20)
start.display()
print("Mahesh Gurnani")
```

output :

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

S3: Thus, entire list can be transversed using node which is referred by the head pointer ~~by~~ of the linked list.

S4: Now that we know that we can transverse the entire linked list using head pointer, we should only use it to refer the first node of list only.

S5: We should not use head pointer to transverse entired 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.

S6: We may lose the reference to the 1st node in our 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 a temporary node to transverse the entire linked list.

S7: We will use this temporary node as a copy of the node we are currently traversing. Since we are making currently making a copy of temporary node the data type of the temporary node should also be node.

S8: Now that current is referring to the first node, if we want to access 2nd node of list refer it as next node of the 1st node.

S9: But the 1st node is referred by current. So, we can transverse to 2nd nodes as h = h.next.

S10: Similarly, we can transverse rest of nodes in the linked list using same method by while loop.

S11: our concern now is to find terminating condition for the while loop.

S12: The last node in linked list is referred by Tail of list. Since the last node of linked list does not have any next node, the value in the next field of the last node is None.

S13: So we can refer the last node of list as
self.s = None.

13/02/2020

S14: We have to now see how to start transversing linked list & how to identify whether we have reached the last node of list or not.

S15: Attach coding a i/p & o/p of above algorithm.

```
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] = 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-09

055

MERGE SORT

Aim: Implementation of merge sort by using python.

Theory: Merge sort is a divide and 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) is key process that assumes that arr[l..m] and arr[m+1, r] are sorted and merges the two sorted sub-array.

Algorithm:

S1: The list is divided into left and right in each recursive call until two adjacent elements are obtained.

S2: Now begins the sorting process. The i and j iterators traverse the two halves in each call. The k iterator traverses the whole lists and makes changes along the way.

S3: If the value at j, L[i] = is average to the arr[i+i] slot and is incremented. If not then R[j] is chosen.

S4: This way, the values being assigned through [l+i] are all sorted.

S5: At the end of this loop, one of the halves may not have been traversed completely. If values are simply assigned to the remaining slots in the list.

S6: Thus, the merge sort has been implemented.

```
while j<n2:
    arr[k] = R[j]
    j+=1
    k+=1
def mergesort(arr, l, r):
    if l<r:
        m = int((l+(r-1))/2)
        mergesort(arr, l, m)
        mergesort(arr, m+1, r)
        sort(arr, l, m, r)
arr = [12,23,34,56,78,45,86,98,42]
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)
set4 = set1 & set2
print("Intersection of set1 and set2: set4", set4)
print("\n")
if set3 > set4:
    print("set 3 is superset of set4")
elif set3 < set4:
    print("set 3 is subset of set4")
else:
    print("set3 is same of set4")
if set4 < set3:
    print("set4 is subset of set3")
    print("\n")
```

PRACTICAL-10

057

SETS.

Aim: Implementation of sets using python.

Algorithm:

~~S1: Now add() method used for~~

S1: Define two empty sets as set2 and set1. Now, use if statement providing the range of the above 2 sets.

S2: Now, add() method is used for adding elements according to the given range then print the sets after addition.

S3: Find the union and intersection of above two sets by using (and) & .! (or) method. Print the sets of union and intersection sets.

S4: Use if statement to find out the subset and superset of set 3 and set 4. Display the above set.

S5: Display that element in set 3 is not in set 4 using mathematical operation

S6: Use its disjoint() to check that anything is common or element is present or not. If not, then display that it is mutually exclusive event.

S7: Use clear() to remove or delete the set and print the set after clearing the element present in the set.

```
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 .isdisjoint(set 5):
    print ("set 4 and set 5 are mutually
            exclusive \n")
set 5. clear ()
print ("After applying clear, set 5 is
        empty set :")
print ("set 5 =", set 5)
```

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 and set 2: set 3 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                 13, 14}
Intersection of set 1 and set 2: set 4 {8, 9, 10, 11}
set 3 is superset of set 4
elements in set 3 and not in set 4: set 5 {1, 2, 3,
4, 5, 6, 7, 12, 13, 14}
set 4 and set 5 are mutually exclusive
after applying clear, set 5 is empty set:
set 5 = set ()
>>>
```

CODE:

~~def Inorder~~

```
class node :
    def __init__(self, value):
        self.left=None
        self.value = 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
            while True :
                if h.left==None :
                    h.left =p
                    print(p.val, "Node is added to left side
                        successfully")
                    break
                else:
                    h =h.left
            else :
                if h.right==None :
                    h.right=p
                    print("p.val, "Node is added to right side
                        successfully")
```

Algorithm:

S1: Define class node and define init() method with 2 arguments. Initialize the value in this method.

S2: Again, define a class BST that is Binary Search Tree with init() method with self arguments and assign the root is none.

S3: Define add() method for adding the node. Define a variable p that p=node(value).

S4: Use if statement for checking the condition that root is none then use else statement for if node is less than the main node then put or arrange that in the leftside.

S5: Use while loop for checking node is less than or greater than the main node and break the loop if it is not satisfying.

S6: Use if statement within that else statement for checking that node is greater than main root then put it into rightside.

S7: After this, left subtree and right subtree repeat

```
            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()
t.add(1)
t.add(2)
t.add(3)
t.add(4)
t.add(5)
```

Output: 030

```
>>> print ("\n Inorder form of tree ", Inorder(t.root))
1
2
3
4
5
Inorder form of tree None
>>> print ("Preorder form of tree ", Preorder(t.root))
1
2
4
3
5
preorder form of tree None
>>> print ("postorder form of tree ", Post order(t.root))
3
5
4
2
1
Postorder form of tree None
```

this method to arrange the node according to Binary Search Tree.

S8: Define Inorder(), preorder() and postorder() with root argument and Use If statement that root is none and return that in all.

S9: In Inorder, else statement used for giving that condition first left, root and then right node.

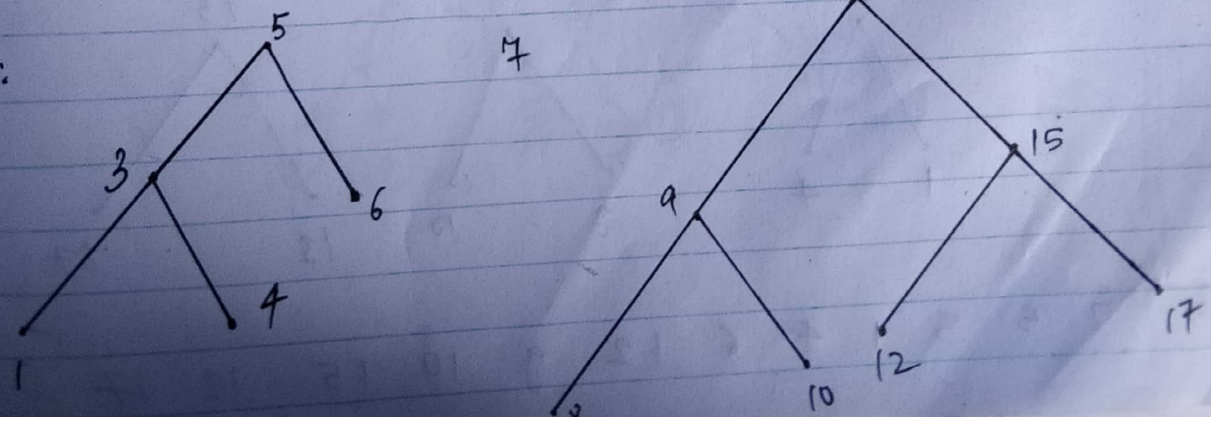
S10: For postorder, In else part, assign left then right and then go for root node.

S10: For preorder, we have to give condition in else that first root, left and then right node.

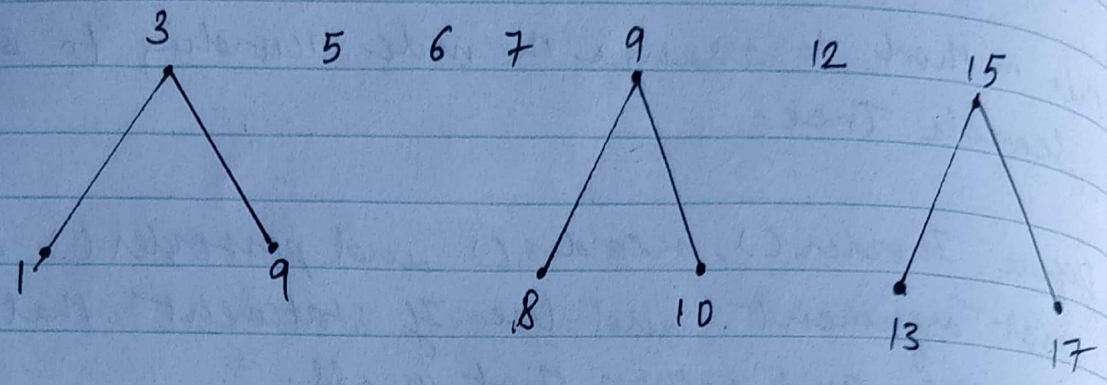
S12: Display the input & output.

- Inorder: (LVR)

S1:



S2:

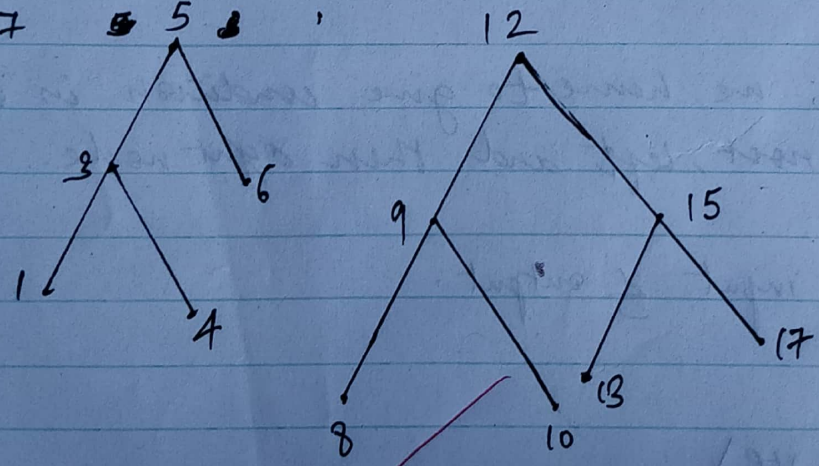


5 6 7 9 12 15

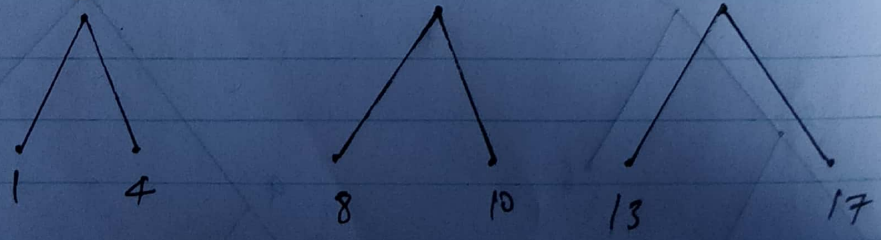
S3: 1 3 4 5 6 7 8 9 10 12 13 15 17

- **Pre-order** : (V L R)

S1: 7 5



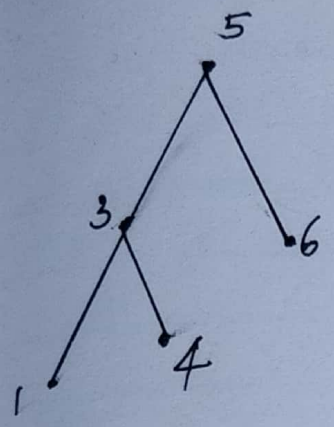
S2: 7 5 3 6 12 9 15

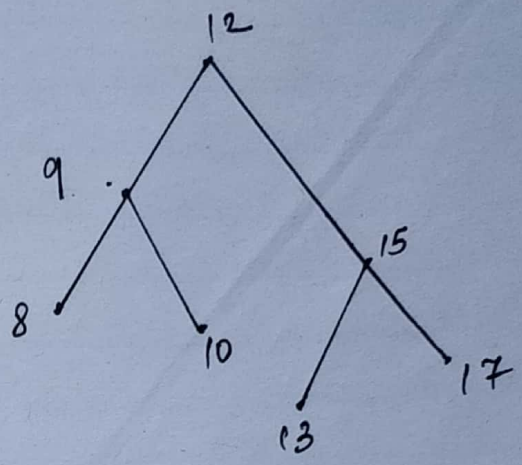


S3: 7 5 3 1 4 6 12 9 10 15 13 17

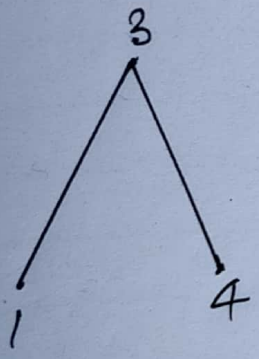
. Post-order : (LRV)

S1:

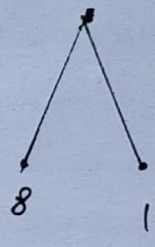


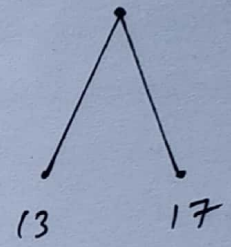


S2:



6 5

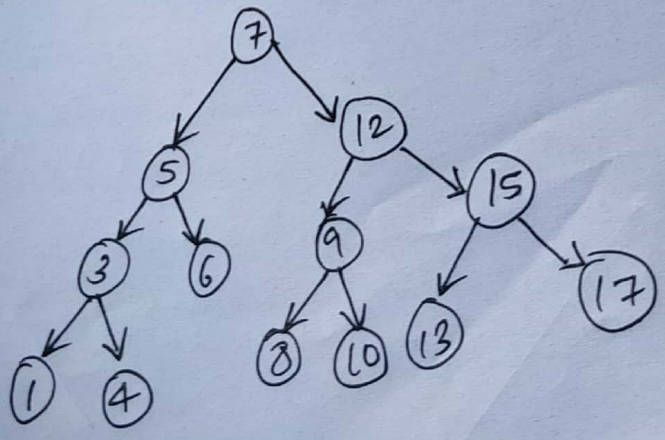




12 7

S3: 1 4 3 6 5 8 10 9 13 17 15 12 7

* BINARY SEARCH TREE:



20/02/2020 -