Step 3: Run a for [illegible] variable i in [illegible] to find the element in the array

Step 4: [illegible] = [illegible] to [illegible]

Step 5: Use if conditional statement to check if the [illegible] with the element you want to find. Then print the [illegible] corresponding to element. If the condition doesn't satisfy then print element is not found.

```
if x == [illegible]:
    print("Element found at location", [illegible])
else:
    print("Element not found")
```

Enter element in array: 3 2 [illegible]

Element in array are [illegible]

Element to be searched: [illegible]

Element found at location 2

Elements in array are [illegible]

Element not found

Step6: Now use for condition statement to append the element 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 define 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.


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

Theory:

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

It is worst searching algorithm with worst case 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 examing the middle term linear search is a technique to compare each other and every element with the key element to be found is both of them matches found and its position is also found.

Practical - 1

Aim: Linear - Search

Q.1] 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 element in the array.

Step 3 : If the condition in step 2 statistics, return the index no. of the given array. If the condition doesn't statistics then get out of loop.

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

Step 5 : Now initialize a variable as empty array.

```
>>> Enter elements in array : 1 2 3 4 5
>>> Elements in array are : [1,2,3,4,5]
>>> Enter element to search: 2
    The element is found at position 1
>>> Enter element in array : 3 2 5 1 4
>>> Enter 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
imp = input("Enter elements in array = ").split()
array = []
for ind in imp:
    array.append(int(ind))
print("Element in array are:", array)
x1 = int(input("Enter the element to be searched:"))
x2 = linear(array, x1)
```

02] Unsorted array:

Algorithm:

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

Step 2: Now use of conditional statement to check whether the given statement is equal to the elements in array.

Step 3: If the condition in step 2 satisfies 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 is empty.

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

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 the linear search which is time consuming. This can be avoided by using binary fashion search.

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

Step 5: Use for conditional statement to append the element in empty array.

Step 6: Now initialize a variable to find the element.

Step 7: Now initialize a variable to call the defined function.

```
inp= input.
arr=[]
for ind inp:
    arr.append(int(ind))
print("Element of array before sorting",
    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 85

>>> Element of array before sorting:
[2, 3, 6, 1, 5]

>>> Element of array after sorting
[1, 2, 3, 5, 6]

Practical : 3

Aim: Implementation of Bubble sort Program on given list.

Theory: Bubble sort is based on the idea of repeatedly comparing pairs of adjacent element and then swapping their position if the sorted from the sorting available. In this, we sort the given element in ascending or descending order by comparing adjacent element of a time. :) ;)

Algorithm:

Step 1: Bubble sort algorithm starts by comparing first two element of an array and swapping if necessary

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

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

Step 4: Again second and third element 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 above should be mentioned n-1 to get the required output.

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

a

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 and 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 top most position.

Step 8: Assign the element in push method as a print the given value.

Step 9: Attach the input and output of above operation.

Step 10: First condition checks whether the number of elements are zero while the second one whether top is assigned any value then print the stack is empty.

Output:

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

```
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
x = 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 ~~go~~ works in the life principle (last in first out). ~~The~~ 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 both 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 list is greater than range of list then print stack

Output

Quick sort

enter element in the list 21 22 20 30 24 56
elements in list are: [21, 22, 20, 30, 24, 56]
elements after quick sort are [20, 21, 22, 24, 30, 56]

Step 9 - Quick sort helper, begins with same base as the merge 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 click the coding and output of above algorithm.

23/01/2024

Step 4- We begin by increasing leftmask until we locate a value that is greater the p.v. We then decrement rightmask until we find value that is less than the p.v. At this point we have discovered the items that are out of place with respect to eventual split point.

Step 5- At the point where rightmask becomes less than leftmask we stop. The position of rightmask now be split point.

Step 6- The p.v can be exchanged with the content of split point and P.V. is now in place.

Step 7 :- In addition all the items to left of split point all less than P.V. and all the items to left to the right of split point all greater than P.V. The list can now be divide at split point and quick sort can be involved recursively. or on the two values.

```
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 = partion(arr, low, high)
        quicksort(arr, low, pi-1)
        quicksort(arr, pi+1, high)
x1 = input("Enter elements in the list").split()
alist1 = []
for b in x1:
    alist.append(int(b))
print("element in list are:", alist1)
n = len(alist1)
quicksort(alist1, 0, n-1)
print("element after quick sort are", alist1)
```

Practical - 5

m: Implement Quick sort to sort the given list.

heory: The quick sort is a recurrence algorithm then based on the 'divide and conquer technique.

lgorithm:

Step 1: Quick sort first select a value, which is called pivot value first value element serve as our 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 item to the appropriate side of the last either lefts then ~~orgo~~ or greater than pivot value.

Step 3- Partitioning begins by locating two position marker ~~and~~ let's call them leftmark and rightmark at the beginning and end of remaining items in the list. The goal of the partition process is to move items that are on wrong side with repeat to first value while converging on the split point.

Output:

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


Algorithm:

Step 1: Define a class Queue and assign global variable. Then define init() method or initialize the inial value with the help of self argument.

Step 2: Define a empty list and define enque we() method with 2 arguments assign the lngth 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 or display that queued element added successfully & increment by it.

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

Step 5: Now all the Queue() function & give the element that has to be added in the empty list by using that after adding &

Practical-6

Implementing a Queue using Python list.

Theory: Queue is a linear data structure which has 2 references front & rear implementing a queue using python list is the simplest as the python list provides in built functions to perform the specified operations of queue. It is based on the principle that a new element is inserted after rear and 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


```
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()
```

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 on the 'P'.

Step 6: If the token is an operator *, /, +, -, n, it will need two operands. Pop the 'P' twice the

Step 7: If token is an operator push the result back on the 'n'.

Step 8: When the arithmetic operation. Push the on the stack. Pop the 'P' and return the value.

Step 9: Print the result of doing the evaluation of postfix.

Step 10: Attach output and input of about 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

```
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(a)*int(a))
```


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 operations in the program. A given postfix expression can easily be evaluated using stacks reading the expression is always from left to right in Postfix.

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 ranges of string then give condition use if statement.

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

Step 10 : Similarly we can traverse rest of node in the linked list

Step 11 : Now condition now is to find determinating condition for the while loop

Step 12 : The last node in the linked list is referred by the tail of linked list.

Step 13 - So we can refer to last node of linked list

Step 14 :- We have to now see how to start traveling the linked list and how to identify whether that we have reached the last node

Step 15 :- Attach the coding or input and output of above algorithm

>> 20
30
40
50
60
70
80

```
            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.addL()
```


Step4: Now that we know that we can traverse the entire linked list using the head pointer we should only avoid to refer the first node of list only

Step 5: We should not use the head pointer do traverse the entire linked list because the head pointer is our only references to 1st node

Step 6: We may loose the refernance 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, will use temporary node to determine the entire linked ~~test~~ list.

Step 7 :- we will use this temporary node as a copy of 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 that current is refurring to the first node if we want to access 2nd node of list we use of list we need do refer it as next node of 1st node.

Practical- 8

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

Theory: A linked list is a linear data structure which is storing the element in a node in a linear fashion but no neccessary in a linear fashion ~~but no~~ element of the linked list called as a Node compress of 2 parts ① Data ② Next Data stores all the information wrt the element whereas next refers to the next node

Algorithm:

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

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

CODE -

```
class node:
    global data
    global data
    def __init__(self, item):
        self.data = item
        self.next = None
class linkedlist:
    global s
    def __init__(self):
        self.s = None
    def addA(self, item):
        newnode = node(item)
        if self.s = None:
            self.s = newnode
        else:
            head = self.s
            while head.next != None:
                head = head.next
            head.head = newnode
    def addB(self, item):
        newnode = node(item)
        if self.s == None:
```

```
# code:
def short (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
```


Aim: Implementation of mergesort by using python.

Theory: Merge sort is a divide and conquer way into two halves its if for the two halves & the merge the two sorted halves the merge function is used for merging the halves.

Algorithm:

Step 1 - The list is divided into left & right in each recursive call until two adjecent element all obtained

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

Step 3: If the value at i is smaller than the value at j (,) 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 Jh[i+1] are all sorted

Step 5 - At the end of this loop, one of the halves may not have been traversed completely and remaining is stored 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 mergsort (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, 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]

Practical-10

Aim: Implementation of sets using python.

Algorithm:

Step 1 - Define two empty set as set and set2 now use for statement printing the range of above 2 sets.

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

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

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

Step 5 - Display that element in set 3 is not in set 4 using mathematical operation.

CODE:

```
set1 = set()
set2 = set()
for i in range(8,15):
    set1.add(i)
for j in range(1,12):
    set2.add(j)
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 & set2: set4",
      set4)
print("\n")
if set3>set4:
    print("set3 is superset of set4")
else:
    print("set3 is same as set4")
```

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


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

set.clar()
print ("set After applying clar, 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 & set 2 : set 3

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

Intersection of set 1 and set 2 {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}

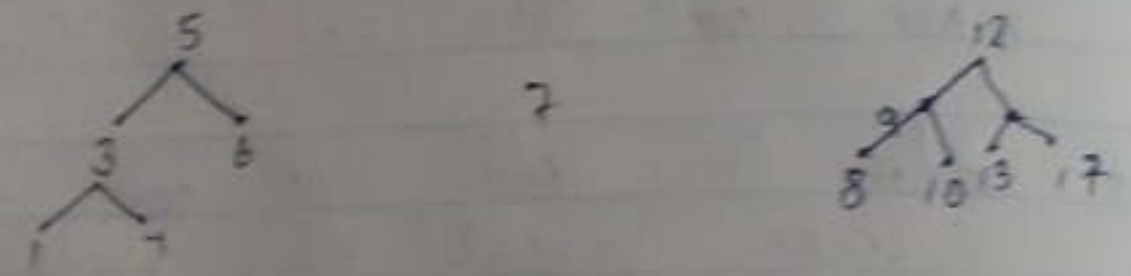
set 4 & set 5 are matually exclusive

after applying clar set 5 is empty set

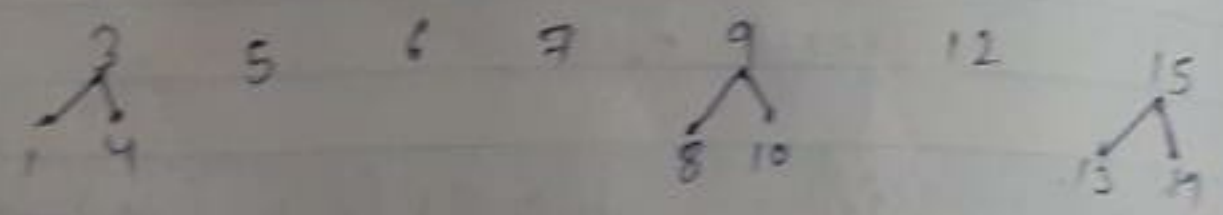
set 5 = set 1)

INORDER: (LVR)

Step 1:



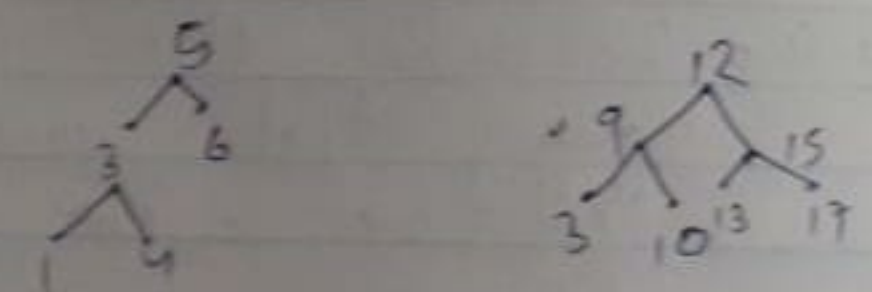
Step 2:



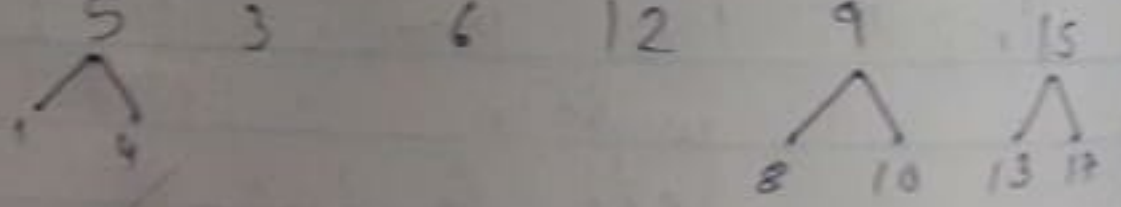
Step 3: 1 3 4 5 6 2 8 9 10 12 13 15 17

• PREORDER: (NLR)

Step 1 : 7



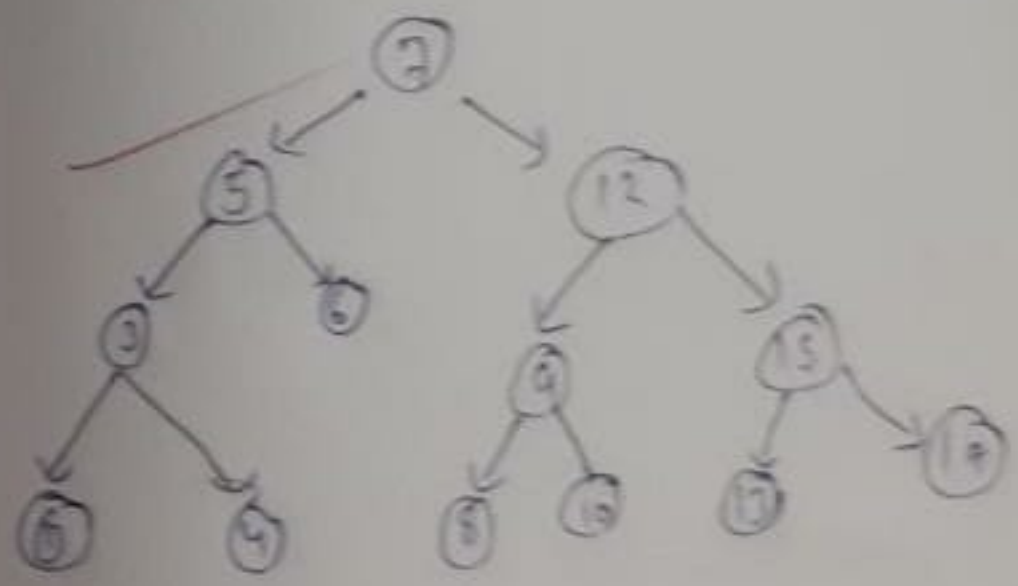
Step 2 : 7 5 3 6 12 9 15



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

```
>>> print("\n Preorder:", Preorder(t.root))
Preorder:
1
2
3
4
5
Preorder : None
>>> Print(" Postorder:", Postorder(t.root))
3
5
4
2
1
Postorder: None
```

* Binary search tree:



Algorithm:

Step 1: Define class node & define init() initialize The with 2 arguments in this method.

Step 2: Again, define a class BST that 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(val

Step 4: Use if statement for checking the conditional that root is none then else statement for if node is then the main node then put arrangement in left side.

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

```
            h.right = p
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.left)
        preorder(root.right)

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

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 sucessfully")
            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

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 can have either = 0 or 1 or 2 children there is another identity of binary tree that if 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 inform might have left and right subtree.

- **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 inform might have left & right subtree:

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

t = BST()
t.insert(1)
root is added successfully
t.insert(3)
3 node is added to right side successfully
t.insert(4)
4 node is added to right side successfully
t.insert(5)
5 node is added to right side successfully
t.insert(2)
2 node is added to leftside successfully
print("In Inorder:", Inorder(t.root))
In Order:
1
2
3
4
5
Inorder done:
```


Step 6 - Use if statement within that else statement for checking that node is greater than main root then put it into rightside

Step 7 - Again this left side tree & right subtree repeat this method to binary search tree

Step 8 - Define Inorder() , Preorder() & postorder() with root as argument & use if statement that root is none & return that all.

Step 9 : In order , else statement will be given that condition if first left root & then right

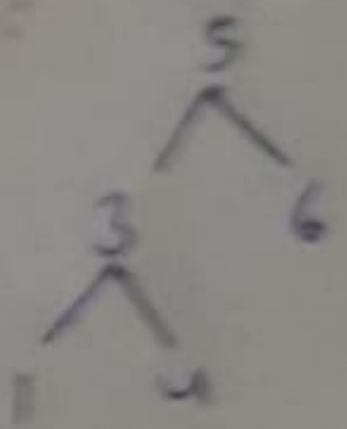
Step 10 : For preorder we have to give condition in else that left root left & then right node.

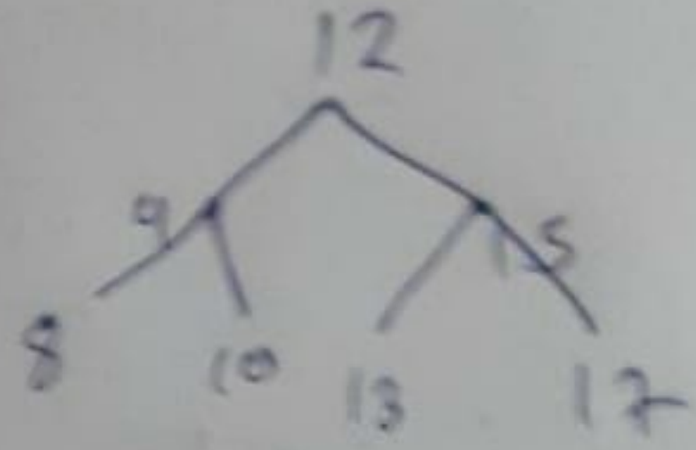
Step 11 : For postorder in else part assign left & right & root

Step 12 - Deploy the output & input

- POSTORDER (L R V)

Step 1:

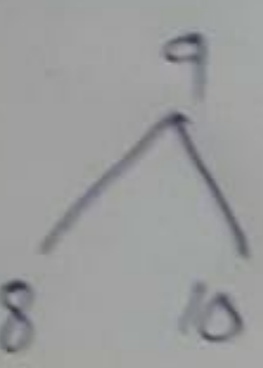


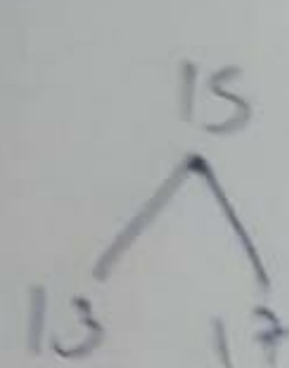


Step 2:

6

5





12 :

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