PRACTICAL NO: 1

AIM: LINEAR SEARCH.

a] SORTED 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 initialize a variable to enter elements in the arrays from user. Now use split() method and to split the values

Step 5: Now initialize a variable as empty array

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

Step 7: Now again initialize another variable to ask 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.

SORTED ARRAY:

CODE:

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

```
>>> Enter elements in array: 1 2 3 5
>>> Element 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.
```

>>> Enter element in array = 1 2 3 5

```
# unsorted array:
def linear(arr,x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
inp = input("Enter elements in array:").split()
array = []
for ind in inp:
    array.append(int(ind))
print("Element in array are:", array)
x1 = int(input("Enter the element to be searched:"))
x2 = linear(array,x1)
if x2 == x1:
    print("Element found at location", x2)
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
>>> Element in array: 2 4 5 3 1
>>> Element in array are: [2, 4, 5, 3, 1]
>>> Element to be searched: 6
Element not found.

UNSORTED ARRAY:

Algorithm:

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

Step 2: Now use if conditional statement to check whether the given statement is equal to the elements in arrays

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

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

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

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

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

Step 5: Again initialize a variable to call the defined function.

Step 9: Use if conditional statement to check if the variable in step 5 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 in not found

THEORY:

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

It is worst searching algorithm with worst case time complexity. It is force approach, on the other hand in case 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, the algorithm returns that element found and its position is also found.

#CODE:

```
a = list(input("Enter the list of elements:"))
a.sort()
n = len(a)
s = int(input("Enter the number to be searched:"))
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((f+1)/2)
      if s==a[m]:
        print("The 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 the number to be searched: 5
The element is found at: 1

PRACTICAL NO: 2

AIM: Binary Search

Algorithm:

Step 1: Create empty list and assign it to a variable

Step 2: Using input method, accept the range of given list

Step 3: use for loop, add elements in list using append() method.

Step 4: Use sort() method to store the accepted element and assign it in increasing ordered list print the list after sorting.

Step 5: use if loop to give the range in which element is found in given range then display a message "Element not found."

Step 6: Then use else statement, if statement is not found in range then satisfy the below condition

Step 7: Accept an argument and key of the element that element has to be searched.

Step 8: Initialize first to 0 and last to last element of the list as array is starting from 0 hence it is initialized 1 less than the total const

Step 9: use for loop and assign the given range.

Step 10: If statement in list and still the element to be searched is not found then find the middle element (m)

Step 11: Else if the item to be searched is still less than the middle term then initialize (the) = mid(m) - 1 else initialize f.(l) = mid(m) - 1.

Step 12: Repeat till you found the element stick the input and output of above algorithm

Theory:

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

17/12/17

#b.

```
# Code
inp = input("enter a element:").split()
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] = a[i]
            a[i] = tmp
print("element after sort", a)
```

```
>>enter a element; 2 5 8 6
element before sorting [2, 5, 8, 6]
element after sort [2, 5, 6, 8]
```


Aim: Implementation of Bubble sort program on given list

Theory: Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their position if they exist in the wrong order. This is the simplest form of sorting available. In this we sort the given elements in ascending or descending order by comparing two adjacent elements at a time.

Algorithm:

Step 1: Bubble sort algorithm starts by comparing the 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 elements.

Step 3: If the first element is smaller than second 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 and second last element is compared and swapped.

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

Step 6: Stop the output and input of above algorithm of bubble sort steps.

```
#Program:
84
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 element in the list").split()
alist1 = []
for b in x1:
    alist1.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 NO: 5 49

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:

Step 1: 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 less than or greater than pivot value

Step 3: Partitioning begins by locating two position markers lets 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 also converging on the split point

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

Step 5: At the point where rightmark becomes less than leftmark we stop. The position of rightmark is now the split point.

Step 6: The pv 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 are less than p.v and all the items to left to the right of split point are greater than p.v. The list can now be divided at split point and quick sort can be invoked recursively or on the two halves

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

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 stick the coding and output of above algorithm

OUTPUT:

```
Quick Sort
enter element in the list: 23 22 21 28 20
element in list are: [23,22,21,28,20]
element after quick sort are [20,21,22,23,28]
```

AIM: Implementation of stack using python list

THEORY: A stack is a lineal data structure that 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 i.e, the topmost 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 three basic operation: push, pop, peek. The opeeations of adding and removing the elements is known as push and pop.

ALGORITHM:

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

Step 2: define the init method with self argument and initialize the initial value 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

PROGRAM 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] = data
    def pop(self):
        if self.tos<0:
            print("Stack is empty")
        else:
            k = self.l[self.tos]
            print("data =",k)
            self.l[self.tos]=0
            self.tos = self.tos-1
s = stack()
s.push(10)
s.push(20)
s.push(30)
s.push(40)
s.push(50)
s.push(60)
```

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 values

Step 6: push method used to insert the element but pop method used to delete the element from the stack

Step 7: If in pop method value is less than 1 then return the stack is empty or else delete the element from stack at topmost position

Step 8: First condition checks whether the no of element 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 not

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

```
s.push(70)
s.push(80)
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
```

OUTPUT:

```
Stack is full
data = 70
data = 60
data = 50
data = 40
data = 30
data = 20
data = 10
Stack 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()
```

PRACTICAL NO: 6.

AIM: Implementing a Queue using python list.

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

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

Algorithm:

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

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

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

Step 4: Define deQueue() with self-argument under this use if statement that front is equal to length of list then display Queue is empty or else, give that front is at zero and using that delete the element from front side and increment it by 1

Step 5: Now call the Queue() function and give the 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 from the list.

OUTPUT:

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

#code:

```
def evaluate(s):
    k = s.split()
    n = len(k)
    stack = []
    for i in range(n):
        if k[i].isdigit():
            stack.append(int(k[i]))
        elif k[i] == '+':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) + int(a))
        elif k[i] == '-':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) - int(a))
        elif k[i] == '*':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) * int(a))
        else: a=
            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)
```

PRACTICAL NO: 7

Evalution of a postfix Expression

AIM: Program on Evalution of given string by using stack in python Environment i.e postfix.

THEORY: The postfix expression is free of any parentheses. Further we took care of the priorities of the operators 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 and print it

step 4: use for loop to assign the range of string then give condition using if statement

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

step 6: If the token is an operator *, /, +, -, ^. it will need two operands. pop the 'p' twice. The

first pop is second operand and the second pop is the first operand

Step 4: perform the Arithmetic operation push the result back on the stack

Step 5: when the input expression has been completely processed the result is on the stack. pop the 'f' and return the value

step 6: Find the result of string after the evaluation of postfix.

Step 7: Attach output and input of above algorithm

output:

The evaluated value is : 6.2

```
# 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 addinit
    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)
```


PRACTICAL NO: 8

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

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

Algorithm:

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

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

Step3: Thus, the entire linked list can be transversed using the node which is referred by the head pointer of the linked list.

Step4: Now that we know that we can transverse the entire linked list using the head pointer, we should only use it to refer the first node of list only

Step5: we should not use the head pointer to transverse the entired linked list because the head pointer is our only reference to the 1st node in the linked list, modifying the reference of the head pointer can lead to changes which we cannot get back.

Step6: we may lose the reference to the 1st node in our linked first and hence most of our linked list. So in order to avoid making same unwanted changes to the 1st node we will use a temporary node to transverse the entire linked list

Step7: we will use this temporary node as a copy of the node we are currently transversing. Since we are making temporary node a copy of current node the datatype of the temporary node should also be node

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

20
30
40
50
60
70
80
```

Step 9: Now that current is refering to the first node, if we want to access 2nd node of list we can refer it as the next node of the 1st node

Step 11: But the 1st node is released by current. So we can transverse to 2nd nodes as h = h.next

Step 10: Similarly we can transverse rest of nodes in the linked list using same method by while loop

Step 11: Our concern now is to find terminating condition for the while loop.

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

Step 13: So we can refer the last node of linked list a self.s = None.

Step 14: we have to now see how to start transversing the linked list and how to identify whether we have reached the last node of linked list

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

PRACTICAL NO: 9

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 the merges the two sorted halves. The merge() function is used for merging two 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-arrays. The algorithm first moves from top to bottom, divides the list into smaller and smaller parts until only the separate element remain.

Algorithm:

Step 1: The list is divided into left and right in each recursive call until two adjacent elements are obtained

Step 2: Now begins the sorting process. The i and j iterators traverse the two halves in each call. The k iterator ~~from~~ traverses the whole lists and makes changes along the way.

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

```
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+=1
        else:
            arr[k] = R[j]
            j+=1
            k+=1

    while i<n1:
        arr[k] = L[i]
        i+=1
        k+=1

    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]
```


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

Step 5: At the end of each loop, one of the values may not have been traversed completely. The values are simply assigned to the remaining slots in the list.

Step 6: Thus, the merge sort has been implemented

PRACTICAL NO:10

AIM: Implementation of sets Using Python.

Algorithm:

Step 1: Define two empty set as set2 and set2 now. Use for statement providing the range of above 2 sets.

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

Step 3: Find the union and intersection of above 2 sets by using (and) &, !(or) method. Print the sets of union and intersection of set 3.

Step 4: Use if statement to find out the subset and Supriset 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.

Step 6: Use is disjoint() to check that anything is common an element is present or not. If not then display that it is mutually Exclusive Event

Step 7: Use clear() to remove or delete the sets and print the set after clearing the element present in the set.


```
set1 = set()
set2 = set()
for i in range (8,15):
    set1.add(i)
for j in range(1,12):
    set2.add(j)
print("set 1:", set1)
print("set 2:", 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("set3 is superset of set4")
elif set3 < set4:
    print("set3 is subset of set4")
else:
    print("set3 is same as set4")
if set4 < set3:
    print("set4 is subset of set3")
    print("\n")
set5 = set3 - set4
print("Element in set3 and not in set4: set5", set5)
print("\n")
if set4.isdisjoint(set5):
    print("set4 and set5 are mutually exclusive \n")
set5.clear()
print("After applying clear, set5 is empty set:")
print("set5=", set5)
```

58

OUTPUT:

Set1: {8, 9, 10, 11, 12, 13, 14}

Set2: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

union of set 1 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

element 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 mutualy exclusive

after applying clear, set 5 is empty set:

Set 5 = set()

13/02/2020

PRACTICAL NO:11

AIM: Implementation of Binary Search tree using python and Inorder, preorder and postorder Transversal.

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

(1) Inorder: (i) Transverse the left subtree. The left subtree inturn might have left and right subtrees.
(ii) Visit the root node.
(iii) Transverse the right subtree and repeat it.

(2) Pre order: (i) Visit the root node
(ii) Transverse the left subtree. The left subtree inturn might have left and right subtrees.
(iii) Transverse the right subtree. Repeat it.

(3) Post order: (i) Transverse the left subtree. The left subtree inturn might have left and right subtrees
(ii) Transverse the right subtrees
(iii) Visit the root node.


```
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
            while True:
                if p.val < h.val:
                    if h.left == None:
                        h.left = p
                        print(p.val, "Node is added to leftside
                                                successfully")
                        break
                    else:
                        h = h.left
                else:
                    if h.right == None:
                        h.right = p
                        print(p.val, "Node is added to rightside successfully
                        break
                    else:
                        h = h.right
def Inorder(root):
    if root == None:
        return
    else:
        Inorder(root.left)
        print(root.val)
        Inorder(root.right)
```

Algorithm:

STEP 1: Define class node and define init() method with 2 argument. Initialise the value in this method.

Step 2: Again, Define a class BST that is Binary Search Tree with init() method with self argument and assign the root is none

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

Step 4: 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 leftside

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

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

Step 7: After this, left subtree and right subtree repeat this method to arrange the node according to Binary Search Tree.


```
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 1
>>> 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 rightsid successfully
>>> t.add(3)
3 node is added to leftside successfully
>>> 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
>>> postorder form
>>> print("Postorder form of tree", Postorder(t.root))
3
5
4
2
1
Postorder form of tree None.
```


Step 9: Define Inorder(), preorder() and postorder() with root as argument and use If statement that root is none and return that in all.

Step 10: In Inorder, else statement used for giving that condition first left, root and then right node.

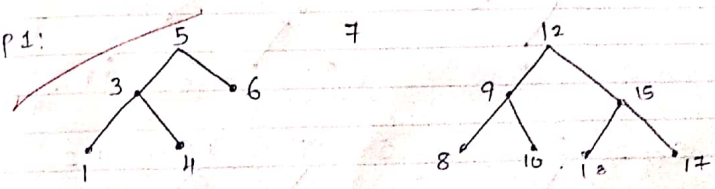
Step 10: For preorder, we have to give condition in else that first root, left and then right node.

Step 11: For postorder, In else part, assign left then right and then go for root node.

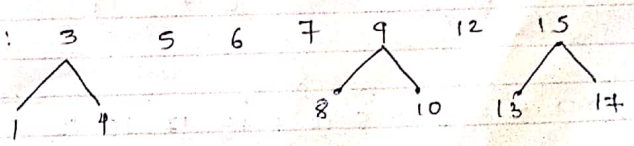
Step 12: Display the output and input

Inorder: (LVR)

Step 1:

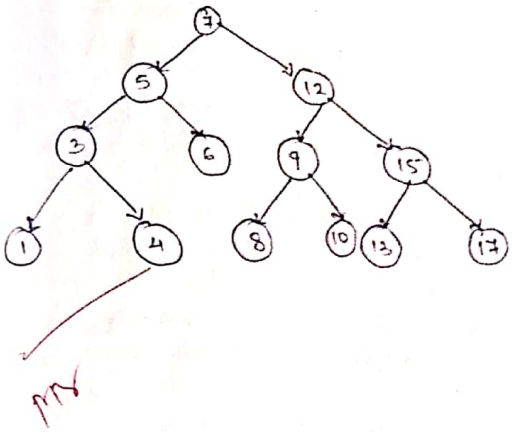


Step 2:



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

Binary Search Tree

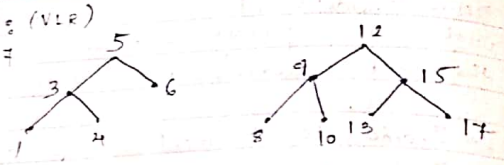


PRY

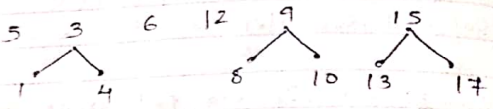
50

- Pre Order : (VLR)

Step 1: 7



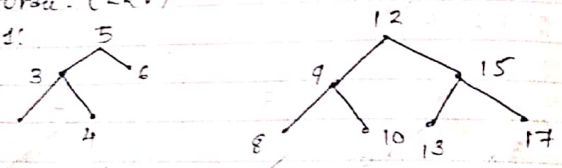
Step 2: 7 5 3 6 12 9 15



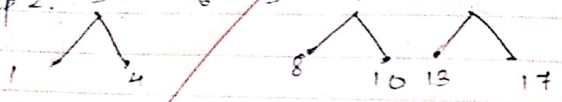
Step 3: 7 5 3 1 4 6 12 9 10 15 13 17

- Post Order : (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 12 7