Practical :- 1.

Aim :- Implement linear search to find an item in the list.

Theory :-

Linear search

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 a 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.

1) Unsorted:

Algorithm:-

Step1: Create an empty set and assign it to a variable.

Step2: Accept the total no. of elements to be inserted into the list from the user. Say 'n'

Step3: Use for loop for adding the elements into the list

Step4: Print the new list.

Step5: Accept an element from the user that to be searched in the list.

Step6: Use for loop in a range from '0' to the total no. of elements to search the element from the list.

Step7: Use if loop that the element in the list is equal to the element accepted from user.

Step8: If the element is found then print the statement that the element is found along with the element's position.

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("Enter Element in array are:", array)
x1 = int(input("Enter the statements to be searched:"))
x2 = linear(array, x1)
if x2 == x1:
    print("Element found at location", x2)
else:
    print("Element not found")
```

output:-

Enter elements 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

Step9:- use another if loop to print that the element is not found if the element which is accepted from user is not their in the list.

Step10:- Draw the output of given algorithm.

2) Sorted linear search:-

Sorting means to arrange the element in increasing or decreasing order.

Algorithm:-

Step1:- Create Empty list and assign it to a variable.

Step2:- Accept total no. of elements to be inserted into the list from user, say 'n'.

Step3:- use for loop for using append() method to add the element in the list.

Step4:- use sort() method to sort the accepted element and assign in increasing order the list then print the list.

Step5:- use if statement to give the range in which element is found in given range then display "Element not found".

Step6:- Then use else statement, if element is not found in range then satisfy the given condition.

Sorted array:-

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

Output:-

Enter elements in array: 1 2 3 4 5

Enter elements in array are : [1, 2, 3, 4, 5]

Enter element to be serached : 2

The Element is found at position 1.

Step7:- use for loop in range from 0 to the total no. of elements to be searched before at doing this accept an search no from user using Input statement.

Step8:- use if loop that the element in the list is equal to the element accepted from user.

Step9:- If the element is found then print the statement that the element is found along with the element position.

Step10:- use another if loop to print that the element is not found if the element which is accepted from user is not their in the list.

Step11:- Attach the input and output of above algorithm.

Practical :- 2

Aim:- Implement binary search to find an search no in the list

Theory :- Binary search

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.

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:- We for loop, add elements in list using append() method

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

Coding :-


```
a=[]
n=int(input("enter element:"))
for b in range(0,n):
    b=int(input("enter a element:"))
    a.append(b)
    a.sort()
    print("Element list is",a)
s=int(input("Enter search number"))
if(s<a[0] or s>a[n-1]):
    Print("Element not found")
else:
    f=0
    i=n-1
    for i in range(0,n):
        m=int(f+1)/2
        print(m)
        if(s==a[m]):
            Print("found element at",m)
            break
        else:
            if(s<a[m]):
                i=m+1
            else:
                f=m+1
```

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

step6:- Then use else statement, if statement is not found in range then satisfy the below condition.

step7:- Accept an argument & key of the element that element has to be searched.

step8:- Initialize first to 0 and last to last element of the list. of array is starting from 0. hence it is initialized 1 less than the total count.

step9:- use for loop & assign the given range.

step10:- If statement in list and still the element to be searched is not found then find the middle element (m).

step11:- Else if the item to be searched is still less than the middle term then
Initialize list(hi) = mid(m)-1
Else
initialize. first(l) = mid(m)-1

step12:- Repeat till you found the element stick the input & output of above algorithm.

Practical :-3

Aim:- Implementation of Bubble Sort program on given list.

Theory:- Bubblesort is based on the idea of repeatedly comparing pairs of adjacent ~~if they exist~~ in element 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 element 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~~ first element is greater than second then; we need to swap the element.

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

Coding:-

```
inp = input("Enter the element:").split()
a=[]
for ind in inp:
    a.append(int(ind))

Print("Enter before sort ",a)
n=len(a)
for i in range(0,n):
    for j in range(n-1):
        if a[i] < a[j]:
            temp=a[j]
            a[j] = a[i]
            a[i] = temp

Print("Enter after sorting",a)
```

output:-

```
>>> Enter the Element: 5 10 11 2 1
Enter before sort[5 10 11 2 1]
Enter after sorting[1 2 5 10 11]
```

Step 4:- Again second and third is smaller ~~than~~ element 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:- Stick the output and input of above algorithm of bubble sort stepwise.

Practical :- 4

Aim :- Implementation of Stack using python list.

Theory :- A Stack is a linear data structure that can be represented in the real world in the form of a physical stack of a the element 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.

Algorithm :-

Step1 :- Create a class stack with instance variable items.

Step2 :- Define the init method with self argument & initilize the initial value & initialize to an empty list.

Step3 :- Define methods push & pop under the class stack.

Step4 :- use if statement to give the condition that if length of given list then print stack is full.


```
print("Rahul yadav")
class stack:
    global tos
    def __init__(self):
        self.l = [0,0,0,0,0,0,0,0,0,0]
        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 empty")
        else:
            k = self.[self.tos]
            print("data =", k)
            self.tos = self.tos-1
x = stack()
```

Output :-

Rahul yadav

```
>>> S.push(70)
>>> S.push(20)
>>> S.push(30)
>>> S.push(40)
>>> S.push(50)
>>> S.push(60)
>>> S.push(70)
>>> S.push(80)
>>> S.push(90)
```

Stak is full

```
>>> S.pop()
    data=80
>>> S.pop()
    data=70
>>> S.pop()
    data=60
>>> S.pop()
    data=50
>>> S.pop()
    data=40
>>> S.pop()
    data=30
>>> S.pop()
    data=20
>>> S.pop()
    data=10
Stack empty.
```


Step 5 :- Or else print statement as insert the element into the stack and initialize the value.

Step 6 :- Push method used to insert the element but POP method used to delete the element from the stack.

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

Step 8 :- Assign the element values in push method & print the given values in poped not.

Step 9 :- First condition check whether the no. of element are zero while the second case whether tos is designed any value. If tos is not assign any value, then they can be store that stack is empty.

Step 10 :- Attach the input & output of above algorithm.

23/01/2020

Practical :- 5

Aim :- Implement Quick sort to sort the given list.

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

Algorithm :-

Step1 :- Quick sort first selects a value, which is called pivot value. first value element. serve as our first pivot value since we knows that first will eventually end up as last in that list.

Step2 :- 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 val.

Step3 :- partitioning begins by locating two position, markers. let calls them leftmark and rightmark at the beginning and end of remaining items in the list.

Step4 :- we begun by increasing leftmark until we locate a value that is greater than the p.v, we the decrement right mark until we find value that is less than the p.v.

Step5 :- At the point where rightmark becomes less than leftmark we stop. The position of rightmark is now the split point.


```
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+j]
    return i+1
def quicksort(arr, low, high):
    if low < high:
        Pi = Partition(arr, low, high)
        quicksort(arr, low, Pi-1)
        quicksort(arr, Pi+1, high)
x1 = input("Enter Elements in the list").split()
alist1 = []
for b in x1:
    alist1.append(int(b))
Print("Elements in the list are:" alist1)
n = len(alist1)
quicksort(alist1, 0, n-1)
Print("Elements after quicksort are:", list)
```

Output:-

Quick sort

Enter elements in the list 21 22 20 30 24 56

elements in list are: [21, 22, 20, 30, 24, 56]

elements after quick sort are [20, 21, 22, 24, 30, 56]

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 ~~and~~ all less than p.v and all the items to left to the right split point are greater than p.v. The list can now be divided at split point and quick sort can be invoked recursively on the two values.

step 8:- The quicksort function invokes a recursive functi.. quick sort helps.

step 9:- quick sort helps, 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 recursived function.

step 12:- The partition function implement the process described earlier.

step 13:- Display and stick the coding and output of above algorithm.

Practical:-6

Aim:- 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 in the simplest as the python list provide inbuilt function.

Queue():- creates a new Empty Queue.

Enqueue():- Insert an element at the rear of the queue & similar to that of insertion of linked using tail.

Dequeue():- Return the element which was at the front, The front is moved to the successive element. A dequeue operation cannot remove element of the queue is empty.

Algorithm:-

Step I:- Defines a class Queue & assign global variables then defines init() method with self argument in init() assign or initialize the initial value with the help of self argument.

Step II:- Define a empty list & define enqueue() with 2 argument assign the length of empty list.


```
class queue:
    global r
    global f
    def __init__(self):
        self.r = 0
        self.f = 0
        self.l = [0,0,0]
    def enqueue(self,data):
        n = len(self.l)
        if self.r<n:
            self.l[self.r] = data
            self.r = self.r+1
            print("Element inserted....", data)
        else:
            print("Queue is full")
            self.r = 0
    def dequeue(self):
        n = len(self.l)
        if self.f<n
            print(self.l[self.f])
            self.l[self.f] = 0
            print("Element Deleted....")
            self.f = self.f+1
        else:
            print("Queue is empty")

q = queue()
```

Output :-

```
>>> Q.add(10)
element insert 10
>>> Q.add(20)
element insert 20
>>> Q.add(3)
element insert 3
>>> Q.add(4)
Queue is full
>>> Q.remove()
10
element deleted.
```


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

Step IV :- Define deQueue() with self argument, under this, use if statement that front is empty or else, give that front is at zero & using that delete the element from front side & increment it by 1.

Step V :- Now call the Queue() functions & give the element that has to be added in the list after adding & some for deleting & display the list after deleting the element from the list.

Practical no:-7

Aim:- Evaluation of given string using stack i.e Postfix.

Theory:- The postfix expression is free of any paranthesis further we took care of the priorities of operation in program. A given postfix expression is always from left to right.

Algorithm:-

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

Step2:- convert the string to a list by using the string method 'split'.

Step3:- calculate the length of string & print it.

Step4:- We for loop to assign the range of string then given condition using if statement.

Step5:- Scan the token list from left to right. If token is an operand convert it from a string to integer & push the value onto the p:

coding:-

```
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))
    return stack.pop()
s = '8 6 9 * +'
r = evaluate(s)
print("The evaluated value is:", r)
```

Output :-
The evaluated value is : 62

Step 6 :- If the token is an operator it will need two operands pop the 'p' twice. The first pop is second operand & the second pop is the first operand.

Step 7 :- Perform the Arithmatic operation push the result back on the 'n'.

Step 8 :- When the input expression has been completed processed the result is on the stack 'pop the 'p' & return the value.

Step 9 :- Print the result of string after the evaluation of postfix.

Step 10 :- Write output & input of above algorithm.

Practical no:-8

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

Theory:- A linked list is a linear data structure which stores the element in a node. The individual element of 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, names, address etc whereas next refers to the next nodes.

Algorithm:-

Step1:- Traversal storing of a linked list means visiting all nodes in linked list in order to perform some operation on them.

Step2:- The entire linked list can be accessed with first node of the linked list.

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

Step4:- Now that we know that we can traverse the entire linked list using.

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

start = linkedlist()
```

Output :-

```
>>> start.addL(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 5 :- we should not use the head pointer to transverse the entired linked list because the head pointer is our only reference to the 1st node in the linked list, modifying reference of head pointer can lead to changes which we cannot convert back.

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

Step 7 :- 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 temporary node should also be node.

Step 8 :- Now that current is referring to the first nodes, if we want to access 2nd node of list we can refer it as the next node of 1st node.

Step 9 :- But 1st node is refered by current. So we can transverse to 2nd nodes as h=h.next.

Step10:- Similarily, we can transverse rest of nodes in the linked list using some method by while loop.

Step11:- Our concept now is to find Terminating condition for the while loop.

Step12:- The last node in the linked list is refered by tail of linked list. Since, the last node of linked list does not have any next node, The value in the next field of last node is None.

Step13:- So we can refer the last node of linked list self.s:-None.

Step14:- we have to now see how to start transversing the linked list & how to identify whether we have reached the last node of linked list or not.

Step15:- write the coding & input, output of above algorithm.

code :-

```
def sort(arr, l, m, r):
    n1 = m-l+1
    n2 = r-m
    L = [0]*(n1)
    R = [0]*(n2)
    for i in range(0,n1):
        L[i] = arr[i+l]
    for j in range(0,n2):
        R[j] = arr[m+1+j]

    i = 0
    j = 0
    k = l
    while i<n1 and j<n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i = i+1
        else:
            arr[k] = R[j]
            j = j+1
            k = k+1

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

    while j<n2:
        arr[k] = R[j]
        j+=1
        k = k+1

def 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)
```


Practical no :- 9

Aim :- Implementation of mergesort by using python.

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

Algorithm :-

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

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

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

Step4 :- This way, the values being assigned through h[i+1] are all sorted.

Step 5:- At the end of this loop, one of the halves may not have been traversed completely remaining slots in the lists.

Step 6:- Thus, the merge sort has been implemented.


```
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 & 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("Elements in set3 and not in set4:
                                set5", set5)
print("\n")
```


Practical No :- 10

Aim :- Implementation of sets using python

Algorithm :-

Step1 :- Define two empty set as set1 and set2 now use for statement providing the range of above 2 sets.

Step2 :- Now add() method is used for addition the element according to given range then print the sets for addition.

Step3 :- Find the union and intersection of above 2 sets by using &, | method. print the sets of union & intersection of set 3.

Step4 :- use if statement to find out the subset and superset of set3 and set4. Display the above set.

Step5 :- Display that element in set3 is not in set4 using mathematical operation.

Step6 :- use clear() to remove or delete the sets and print the set after clearing the element present in the set.

```
if set4.isdisjoint(set5):
        print("set4 and set5 are mutually exclusive\n")
set.clear()
print("After applying clear, set5 is empty set:")
print("set5:", set5)
```

Output:-

set1: {8,9,10,11,12,13,14}
set2: {1,2,3,4,5,6,7,8,9,10,11}
union of set1 & set2 : set3
{1,2,3,4,5,6,7,8,9,10,11,12,13,14}
Intersection of set1 & set2 {8,9,10,11}
set3 is superset of set4
Elements in set3 and not in set4 : set5
{1,2,3,4,5,6,7,12,13,14}
set4 & set5 are mutually exclusive
After applying clear, set5 is empty set
set5 = set()

#Code:

```
class node:
    def __init(self,value):
        self.left = None
        self.val = value
        self.right = None

class BST:
    def __init__(self):
        self.root = None
    def add(self, value):
        P = node(value)
        if self.root == None:
            self.root = P
            Print("Root is added successfully",
                                        P.val)
        else:
            h = self.root
            if P.val < h.val:
                if h.val = None:
                    h.left = P
                    Print(P.val," None is
                    added successfully")
                    break;
                else:
                    h = h.left
            else:
                if h.right = None:
                    h.right = P
                    Print(P.val,"None is
                        added to right side
                        Successfully")
                    break;
                else:
                    h = h.right
```


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

Theory:- Binary tree is a tree which supports maximum of 2 children for any node within the tree. Thus any particular node can have either 0 or 1 or 2 children. There is another identity of binary tree that 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 inturn might have left and right subtrees.
- Preorder: 1) visit the root node traverse the left sub tree and right subtree. traverse the right subtree.
- Postorder: 1) Traverse the left subtree. The left subtree inturn might have left & right subtrees.

Algorithm:-

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

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

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

Step 4:- use if statement for checking the conditional the root is none then use else statement for if node is less than the main node then put or arrange them in left side.

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

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


```
def Inorder(root):
    if root = None:
        return
    else:
        inorder(root.left)
        Print(root.val)
        Inorder(root.right)
def preorder(root):
    if root == None:
        return
    else:
        Print(root.val)
        Preorder(root.left)
        Preorder(root.right)
def postorder(root):
    if root == None:
        return
    else:
        postorder(root.left)
        postorder(root.right)
        Print(root.val)
t=BST()
```

#Output:-

```
>>> t.add()
root is added successfully
>>>t.add(2)
2node is added to rightside successfully
```

```
>>> t.add(4)
4 node is added to right side successfully
>>> t.add(5)
5 node is added to right side successfully
>>> t.add(3)
3 node is added to ~~right~~ left side successfully
>>> print("\n Inorder:", Inorder(t.root))
 Inorder:
1
2
3
4
5
Inorder: None
>>> print("\n Preorder:", Preorder(t.root))
 Preorder:
1
2
3
4
5
Preorder: None
>>> Print("\n Postorder:", Postorder(t.root))
 Postorder:
3
5
4
2
1
Postorder: None
```

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

Step 8:- Define inorder() preorder() & Post order() with root as argument & use if statement that root is none & return that all.

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

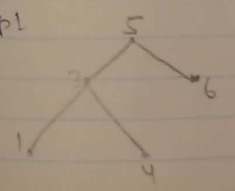
Step 10:- For preorder, we have to give condition in else that first root, left & the right node.

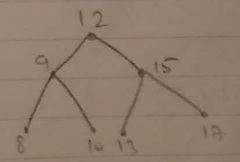
Step 11:- for postorder in else part assign left & right & root.

Step 12:- Display the output & input.

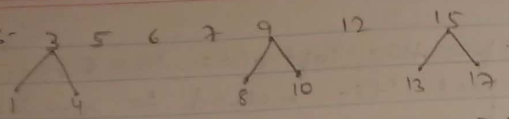
- INORDER (LVR)

Step 1





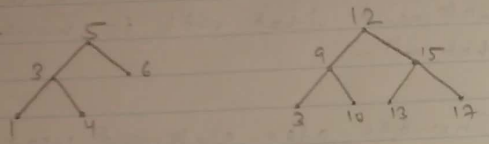
Step 2:- 3 5 6 7 9 12 15



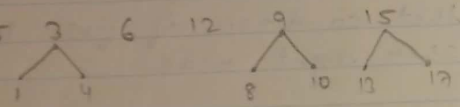
Step 3: 1 3 4 5 6 7 8 9 10 12 13 15 17

• PREORDER (VLR)

Step 1: 7



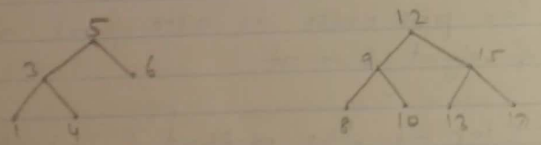
Step 2: 7 5 3 6 12 9 15



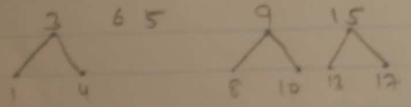
Step 3:- 7 5 3 1 4 6 12 9 8 10 15 13 17

• POSTORDER (LRV)

Step 1:



Step 2:-



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

62

* Binary search tree: