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 each item in the list. It is worst searching algorithm with worst case time complexity. It is a force approach. On the other hand incase of an ordered list, instead, of searching the list in sequence. A binary search is used which will start by examining the middle term.
Linear search is a technique to compare each & every element with the key element to be found, if both of them matches, the algorithm returns that element found & the position is also found.

Sorted array

Algorithm:-

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

Step 2:- Accept the total no. of elements to be inserted into the list from the user say 'n'.

Step 3:- Use for loop for adding the elements into the list.

Step 4:- Print the new list.

Step 5:- Accept an element from the user that to be searched in the list.

Step 6:- Use the for loop in a range from '0' to the total no. of elements to search the elements from the list.

Step 7:- Use if loop that the elements in the list is equal to the element accepted from user.

Step 8:- If the element is found then print the statement that the element is found along with the elements position.

Step 9:- Use another if loop to print that the element is not found if the element which is accepted from user is not there in the list.

Step 10:- Terminate the process of given operation.

Program code:-

```
def linear(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

inp = input("Enter elements in array: ")
array = []
for ind in inp:
    array.append(int(ind))
print("elements in array are:", array)
array.sort()
x = int(input("enter element to be searched: "))
x1 = linear(array, x)
if x1 != -1:
    print("Element found at position:", x1)
else:
    print("Element not found")
```

Enter elements in array: 1 2 3 4 5
Elements in array are: [1, 2, 3, 4, 5]
Enter element to be searched: 2
The element is found at position 1.

// Unsorted array :-

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

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

>>> Enter elements in array 3 2 4 5 1.
>>> Elements in array are: [3, 2 4 5 1]
>>> Elements to be searched: 4
Elements found at location 2.

2) Sorted Linear search :-

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

- Algorithm :-

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

Step 2 :- Accept total no. of elements to be inserted into the list from user, say n.

Step 3 :- Use for loop for using append() method to add the elements in the list.

Step 4 :- Use sort() method to sort the accepted element & assign in increasing order the list. Then print the list.

Step 5 :- Use if statement to given the range in which element is found in given range then display "Element not found".

Step 6 :- Then use else statement, if element is not found in range then satisfy the given cond^n.

Step 7 :- Use for loop in range from 0 to the total no. of elements to be searched before doing this accept an search no from user using input statement.

Step8 - Use if loop that the elements in the list is equal to the elements 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 there in the list

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

19/12/19

```
#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
arr = input("Enter the sorted list of numbers :").split()
arr = []
for ind in array:
    arr.append(int(ind))
key = int(input("Enter element to search:"))
index = binary(arr, key)
if index < 0:
    print("Element not found")
else:
    print("Element found at index", index).
```

Practical 2 :- Binary Search

41

Aim :- Implement Binary Search to find the searched no in the list.

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 sort 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 cond^.

step 7 :- Accept an argument of key of the element that element has to be searched.

Step1- Initialize first to 0 and last to last element of the list as array is starting from 0 hence if is initialized if less than the total count.

Step2- use for loop & assign the given range.

Step3- If statement in list and still the element to be searched is not found then find the middle element (m).

Step4- Else if the item to be searched is still less than the middle term then
Initialize last (h) = mid (m) - 1
else
Initialize first (f) = mid (m) - 1.

Step5- Repeat till you found the element stick the input & output of above algorithm.

Step6- If the first element is smaller than second then we do not swap the element.

Step7- 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.

Step8- If there are n elements to be sorted then the process mentioned above should be repeated n-1 time

>>> Enter the elements in array:
3 5 10 12 15 20
>>> element to be search: 12
12 element was found at index 3
>>> Enter the elements in array:
3 0 1 5 6 78
>>> Elements to be search: 2
element was not found.

```
# Code :-
n = int(input("Enter elements: "))
arr = []
for i in range :
    arr.append(int(input()))
print("Elements of array before sorting are :", arr)
n = len(arr)
for i in range(0, n):
    for j in range(n-i):
        if arr[j] > arr[j+1]:
            temp = arr[j]
            arr[j] = arr[j+1]
            arr[j+1] = temp
print("Element of array after bubble sort :", arr)
```

>>> Enter elements : 2 3 6 1 5
>>> Elements of array before sorting : [2, 3, 6, 1, 5]
>>> Elements of array after sorting : [1, 2, 3, 5, 6].

Practical 3 - Bubble sort

- **Aim :-** To implement bubble sort program on given data.
- **Theory :-** Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements & then swapping their positions 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 :-**

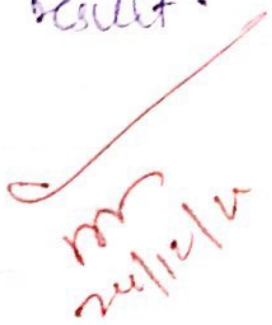
Step 1 :- Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary.

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 element.

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

Step 4 :- Again second & third elements are compared and swapped if it necessary and this process go on until last and second last element is compared and swapped.

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



24/12/19

code:-

```
class stack:
    def __init__(self):
        self.l = [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:
            x = self.l[self.tos]
            print("data=", x)
            self.l[self.tos] = 0
            self.tos = self.tos - 1
s = stack()
```

15. Peeks

Practical-4 – Stack :-

45

- 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 topmost position. Thus it works in the LIFO principle (Last in first out). It has 3 basic operations namely: push, pop, peek.
- Algorithm:-

Step1:- Create a class stack with instance variable item
Step2:- Define the init method with self argument & initialize the initial value & then initialize the empty list.
Step3:- Defined methods push & pop under the class stack
Step4:- use if statement to give the condn that if length of given list is greater than the range of list then print stack is full.
Step5:- Use else statement to print a statement as input the element into the stack & initialize the value
Step6:- Push method is used to insert the element but pop method is used to delete the element from the stack.
Step7:- If in pop method, value is less than 1 then return is empty, or else delete the element from stack at topmost position.
Step8:- Assign the element values, in push method

Step 9:- Attach the input & output of above algorithm.
Step 10:- First cond^n check whether the no. of elements are zero while the second checks whether top is assigned any value. If top is not assigned any value then print that the stack is empty.

Output:-

```
>>> x.push(10)
>>> x.push(20)
>>> x.push(30)
>>> x.push(40)
>>> x.push(50)
>>> x.push(60)
>>> x.push(70)
The stack is full
>>> x.
[10, 20, 30, 40, 50, 60, 70]
>>> x.pop()
70
>>> x.pop()
60
>>> x.pop()
50
>>> x.pop()
40
>>> x.pop()
30
>>> x.pop()
20
>>> x.pop()
10
>>> x.pop
The stack is empty
```

09/01/21

```
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)
x = input("enter elements in the list").split()
alist = []
for b in x:
    alist.append(int(b))
print("elements in list are:", alist)
n = len(alist)
quicksort(alist, 0, n-1)
print("elements after quick sort are:", alist).
```


Practical 5 - Quick Sort

Aim: Implement Quick sort to sort the given list.

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

ALGORITHM:-

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

Step2:- The partition process will happen next. It will find the split point & at the same time move other items to the appropriate side of the list either less than or greater than pivot value.

Step3:- Positioning begins by locating two position markers, lets call them leftmark & right mark at the beginning & end of remaining items in the list. The goal of the partition process is to move items that are on the wrong side with repeat to first value while also converging on the split point.

Step4:- We begin by increasing leftmark until we locate a value that is greater than the pivot value, we then decrement right mark until we find value that is less than the pivot value. At this point, we have discovered two items that are out of place with respect to

eventual split point.
Step5:- At the point where rightmark becomes less than leftmark we stop. The position of rightmark is now the split point.
Step6:- The pivot value can be exchanged with the content of split point and pivot value is now in place.
Step7:- In addn all the items to left of split point are less than pivot value & all the items to left to the 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 halves.
Step8:- The quicksort funcn invokes a recursive fn quick sort helps.
Step9:- Quick sort helps, begins with same base as the merge sort.
Step10:- If length of the list is less than 0 or equal to one, it is already sorted.
Step11:- If it is greater than it can be partitioned and recursive funcn.
Step12:- The partition fn implement the process described earlier.
Step13:- Display & stick the coding and output of above algorithm.

Output:-

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

23/01/2020

Output:-

>>> Q.add(7)
>>> Q.add(40)
>>> Q.add(60)
>>> Q.add(70)
>>> Q.add(90)
>>> Q.add(95)
Queue is full
>>> Q.

Practical-6:- Queue :-

Title :- Implementing a Queue using python list.

Theory :- Queue is a linear data structure with two references front and rear. Implementing a Queue using python list is the simplest as the python list provides inbuilt functions to perform the spritied operations of the queue.

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.

Algorithm:

Step1: Define a class Queue and assign global variables then define __init__() method with self argument in init() assign or initialize the initial value with the help of self argument.

Step2: Define a empty list and define enqueue() method with 2 arguments assign the length of empty list.

Step3: Use if statement that length is equal to rear then queue is full or else insert the element in empty list or display that queue element added successfully & increment by 1.

Step4: Define Queue() with self argument. Use if statement that front is equal to length of list then display Queue is empty or else give that front is at 0 & using that delete the element from front side & increment it by 1.

Step5: Now all the Queue() fn & give the element that has to be added in the empty list by using enqueue() & print list after adding & same for deleting.

#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().
```

```
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 = stack.pop()
            b = stack.pop()
            stack.append(int(b) / int(a))
    return stack.pop()
s = "8 9 * +"
r = evaluate(s)
print("The evaluated value is:", r)
```

Practical - 7

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 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:-

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

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

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

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

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

Step6:- If the token is an operator *, /, +, -, ^, it will need two operands. Pop the 's' twice. The first pop is second operand & the second pop is the first operand.

Input - Perform the arithmetic operation. Push the result back on the stack.

Step - When the input expression has been completely processed, the result is on the value.

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

Step - Attach output & input of above algorithm.

Output :-

The evaluated value is : 62

```
# code
class node:
    global data
    global next
    def __init__(self, item):
        self.data = item
        self.next = None

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

    def addB(self, item):
        newnode = node(item)
        if self.s == None:
            self.s = newnode
        else:
            newnode.next = self.s
            self.s = newnode.
```


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 elements in a node in a linear fashion but not necessarily continuous. The individual element of the linked list called as a node comprises of 2 parts ① Data ② Node Next. Data stores all the information wrt the element whereas next refers to the next node.

A Algorithm:-

Step1: Traversing of a linked list means positioning all the nodes in the linked list in order to perform some operation on them.

Step2:- The entire linked list means can be accessed as the 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 linked list.

Step4:- Now that we know that we can traverse 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 traverse the entire linked list because the head pointer is our only reference to 1st node.

Step6 - We may lose the reference to 1st node in our linked list & hence most of our linked list. So, in order to avoid making some unwanted changes to the 1st node we will use temporary node to terminate the entire linked list.

Step7 - We will use this temporary node as a copy of the node we are currently traversing. Since, we are making temporary node a copy of current node the datatype of temporary node should also be a node.

Step8 - Now that current is referring to the first node, if we want to access 2nd node of list we need to refer it as next node of 1st node.

Step9 - But the 1st node is referred by current so we can traverse to 2nd node as x = x.next

Step10 - Similarly, we can traverse rest of nodes in the linked list.

```
def display (self):
    head = self.
    while head.next != None:
        print (head.data)
        head = head.next
    print (head.data)
start = Linkedlist()
```

Output -

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

Step 11 - Our concern now is to find terminating 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 traversing the linked list & how to identify whether we have reached the last node.

22

Practical 7:-

Aim:- Implementation of merge sort using Python.

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

Algorithm:-

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

Step2:- Now begin the sorting process. The i & j iterators traverse the two halves in each call. Then k iterator traverse the whole list & makes the changes in position accordingly.

Step3:- If the value at i is smaller than the value at j is assigned to the arr[it] slot and is incremented. If not then R[j] is chosen.

Step4:- In this way, the values being assigned through h[it] are all sorted.

Step5:- At the end of this loop, one of the halves is not being traversed completely, keeping remaining slots in the list unchecked.

Step6:- After the sorting, the elements are placed in the particular order & then the merge sort has been implemented.

```
# code
def sort(arr, l, m, r):
    n1 = m-l+1
    n2 = r-m
    L = [0]*(n1)
    R = [0]*(n2)
    for i in range(0, n1):
        L[i] = arr[l+i]
    for j in range(0, n2):
        R[j] = arr[m+1+j]
    i=0
    j=0
    k=l
    while i<n1 and j<n2:
        if L[i]<=R[j]:
            arr[k] = L[i]
            i+=1
        else:
            arr[k] = L[i]
            j+=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)
merge(arr, l, m, r)

arr = [12, 23, 34, 56, -78, 45, 56, 98, 42]
print(arr)
n = len(arr)
mergesort(arr, 0, n-1)
print(arr)
```

Output:-

[12, 23, 34, 56, 48, 45 86, 98, 42]

[12, 23, 34, 56, 42, 45, 48, 86, 98]

Practical 10.-

Aim:- Implementation of sets using Python..

Algorithm:-

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

Step2:- Now use add() method for addition of the elements according to given range then print the sets for addition.

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

Step4:- Use if statement to find out the subset and superset of set 3 and set 4. Display the sets.

Step5:- Display that element in set 3 is not present in set 4 using mathematical operations.

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 & 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 are not in set4: set5", set5)
print("\n")
if set1.isdisjoint(set5):
    print("Set1 and set5 are mutually exclusive\n")
set5.clear()
print("After applying clear, set5 is empty set:")
```

Output:

Set 1 = {8, 9, 10, 11, 2, 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, 4, 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 are not in set 4.
Set 5 = {1, 2, 3, 4, 5, 6, 7, 12, 13, 14}.
Set 4 and set 5 are mutually exclusive.
After applying clear, set 5 is empty set.
Set 5 = set( ).

Practical - 11

59

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 it is ordered such that one child is identified as left child and other as right child.

Inorder :- Transverse the left subtree. The left subtree information might have left and right substance.

Preorder :- Visit the root node traverse the left subtree and right subtree.

Postorder :- Traverse the left subtree, the left subtree in return might have left & right substance.

Algorithm :-

1) Define the class node and init() which has 2 arguments initialization the value in this method.

2) Again define a class BST that in binary search tree with init() with self argument & assign the root is None.

3) Define add() for adding the node, define a variable p that p-node(value).

4) Use if statement for checking the condition that root is None then use else statement for if node is less than main node then put an argument that is left side.

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.

6) Use if statement within that else statement for declaring that node is greater than main not then put it into right side.

7) After this for leftside tree and right tree accessed this method to arrange the node according to binary search tree.

```
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, "Node is added
                        successfully").
                    break
                else:
                    h = h.left.
            else:
                if h.right = None:
                    h.right = p
                    print(p.val, "Node is added
                    successfully to the right node").
                    break.
```

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

t = BST()
```


c) Define Inorder(), Preorder() & Postorder() with root as argument & use if statement that root is None & return that all.

a) Inorder, use else statement for giving that condition first left node & then right node root

b) for preorder, you have to give condition in else that first left root & then right node.

d) for postorder, in else part assign left and right root, so display input and output.

Output :-

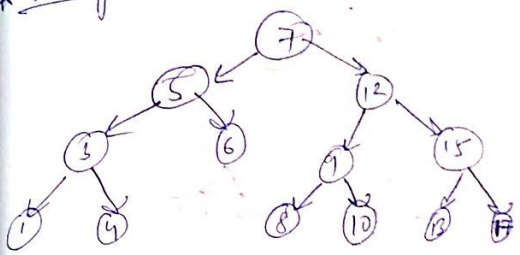
>> t.add(1)
root is added successfully.
>> t.add(2)
node is added successfully to left side.
>> t.add(4)
node is added to right side.
>> t.add(5)
node is added to left side.
>> t.add(3).
node is added to right side.

>> print("\n Inorder:", Inorder(t.root))
Inorder:
1
2
3
4
5
Inorder: None.

>> print("\n Preorder:", Preorder(t.root)).
Preorder
1
2
4
3
5

>> print("Postorder:", Postorder(t.root))
Postorder:
5
5
4
3
2
1
Postorder: None.

* Binary tree:

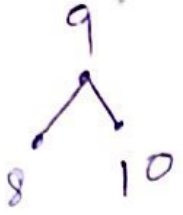


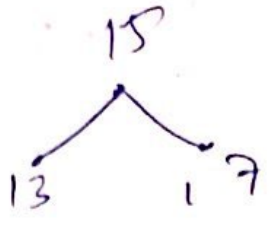
Postorder (LRV) :-

1) 

2) 3 6 5 9 15 12 7

e)

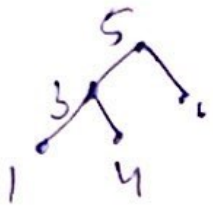


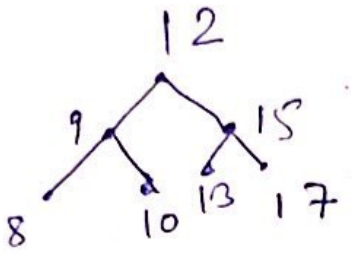


⇒ 1) 1 4 3 6 5 8 10 9 13 17

⇒ # Inorder (LVR).

1)



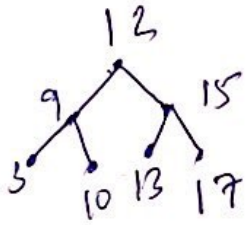


⇒ 2) 3 5 6 7 9 12 15

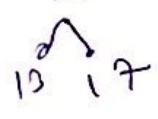
3) 1 3 4 5 6 7 8 9 10 12 13 15 17

Preorder ~~(LVR)~~ (VLR)

⇒ 1)



2) 7 5 3 6 12 9 15



3) 7 5 3 1 4 6 12 9 8 10 15 13