☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member's Signature
1.	Implement linear search to find an item in list	31	25/12/19	2/12/19
2.	Implement binary search to find an item in list	35	2/12/19	2/12/19
3.	Implement bubble sort to find an item in list	37	9/12/19	9/12/19
4.	Quick sort	39	23/12/19	17/02/20
5.	Implementation of stack	41	6/1/2020	06/01/2020
6.	Implementation of Queue		13/1/2020	
7.	Evaluation of Postfix		20/1/2020	27/02/20
8.	Implementation of linked list by adding the nodes from last position			
9.	Implementation of sort		27/1/2020	
10	Implementation of Sets in python		2/2/2020	
11.	Binary Tree		10/2/2020	10/02/20

Pratical no 1

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

Theory :-

Linear search

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

It is worst searching algorithm with worst case complexity of time. It is a force approach. On the other hand in case of an ordered list, istead of searching the list in sequence a binary 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 matched, the algorithm returns that element found and its position is also found.

① Unsorted :-

* Algorithm :-

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

step 2 : Accept the total number of elements to be inserted into the list from the user.

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 is to be searched in the list.

step 6 : Use for loop in a range from '0' to the total number 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 no element is found along with the element's 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 their in the list

step 10 : Draw the output of the given algorithm.

Code :-

```
print (" linear search")
a = []
n = int (input ("Enter the range = "))
for s in range (0, n):
        s = int (input ("Enter the number = "))
        a. append (s)
        print (a)
c = int (input (" Enter a search number = "))
for i in range (0,n):
        if (a[i] == c):
                print (" found at position = ", i)

else :
        print ("Not found")
```

OUTPUT :-

```
>>> linear search
Enter the range = 3
Enter the number = 2
[2]
Enter the number = 1
[2, 1]
Enter the number = 3
[2,1,3]
Enter a search number = 3
found at position = 2
```

Code :-

```
print ("Linear search")
a = []
n = int (input ("Enter a range = "))
for s in range (0, n):
    s = int (input ("Enter a number = "))
    a.append (s)
    a.sort()
    print (a)
c = int (input ("Enter a search number = "))
for i in range (0, n):
    if (a[i] == c):
        print ("found at position =", i)
        break
else:
    print ("Not found")
```

mr

② Sorted :-

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

* Algorithm :

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

Step 2 : Accept total number of elements to be inserted into the list from the user.

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

Step 4 : Use sort() method to sort the accepted elements and assign in increasing order the list then print the list.

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

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

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 the input statement

Step 8 : Use if loop that the elements in the list is equal to the element accepted from user.

Step 9 :- If the element is found then print the statement that the element is found along with the element position.

Step 10 : Use another if loop to print that the element is not found if the element which is accepted from the user is not there in the list

output :

```
>>>  Linear search
     Enter a range = 3
     Enter a number = 2
     [2]
     Enter a number = 1
     [1, 2]
     Enter a number = 3
     [1, 2, 3]
     Enter a search number = 2
     found in position = 1
```

P8.

Code :

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

mm 21/11/19

Pratical - 2

Aim :- Implement Binary search to find a searched number 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 making use of binary fashion search.

Algorithm :-

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

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

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

Step 4 :- Use sort() method to sort the accepted element & assign it in increasing ordered list. Print the list.

Step 5 :- Use if loop to give the range in which element is found in given range then display "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 & key of element that element has to be searched.

Step 8 :- Initialize first to 0 & last to last element of the list. As array is starting from 0 hence it is initialized 1 less than the total count.

Step 9 :- Use for loop & assign the range.

Step 10 :- If statement in list and still the element to search is not found then find the middle element

Step 11 :- Else if the item to be searched is still less than the middle term then Initialize last (n) = mid (m-1)

Else

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

Step 12 :- Repeat till you found the element - Display the output of the above algorithm

Output :

```
>>> Enter a range = 4
    Enter a number = 2
    [2]
    Enter a number = 1
    [1, 2]
    Enter a number = 4
    [1, 2, 4]
    Enter a number = 8
    [1, 2, 4, 8]
    Enter a search number = 2
    Element found at = 1
```

Code :-

```
print ("Bubble sort")
a = []
b = int (input ("Enter no of elements = "))
for s in range (0, b):
    s = int (input ("Enter element: "))
    a.append (s)
    print (a)
n = len (a)
for i in range (0, b):
    for j in range (n-1):
        if a[i] < a[j]:
            temp = a[j]
            a[j] = a[i]
            a[i] = temp
print ("Elements after sorting: ", a)
```

2/12/19

Pratical no :- 3

Aim :- Implementation of Bubble sort programm of the given list.

Theory :-

Bubble sort

Bubble sort is based on the idea of repeatedly comparing pair 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 element in ascending or descending order by comparing two adjacent elements at a time.

- Algorithm :-

Step 1 :- Bubble sort algorithm start 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 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.

Output :-

Bubble sort

Enter no. of elements = 4

Enter elements : 6

[6]

Enter elements : 5

[6, 5]

Enter elements : 8

[6, 5, 8]

Ebnter elements : 2

[6, 5, 8, 2]

Elements after soarting : [2, 5, 6, 8]

9/12/19

Pratical - 4

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

Theory :- The given sort is a recurrsion algorithm based on the divide and the conquer technique.

Algorithm :-

step 1 :- Quick sort first selects a value, which is called pivot value. First 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 either items to the appropriate side of the list, either less than or greater than value

step 3 :- Partitioning begins by locating two position markers lets call them left marks and right marks at the end and beginig of remaining items in the list. The goal of the partition process is to move items that are wrong with respect to the pivot value while also converging at the split point.

step 4 : We began by incrementing left mark until we locate a value that is greater than p.v. 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.

Step 5 : At the point, where right mark becomes less than left mark, we stop. The position of right mark is now the split point

step 6 : The pivot value will be exchanged with the content of the split point and p.v

step 7 : In addition all the items to the left of split point are less than p.v and all the items to the right.

```
def quick sort (a list) :
    help (a list, 0, len (a list)-1)
def help (a list, first, last):
    if first < last :
        split = part (alist, first, last)
        help (a list, first+1, last)
        help (a list, split+1, first)
def part (a list, first, last):
    pivot = a list [first]
    l = first + 1
    r = last
    done = false
    while not done :
        while l <= r and list [l] <= pivot :
            l = l + 1
        while a list [r] >= pivot and r >= l :
            r = r - 1
        if r < l :
            done = True
        else :
            temp = a list [l]
            a list [l] = a list [r]
            a list [r] = temp
    temp = a list [first]
    a list [first] = a list [r]
    a list [r] = temp
    return r
x = int (input ("Enter a range of list :"))
a list [ ]
for b in range (0, x):
    b = int (input ("Enter element :"))
    a list. append (b)
    0 = len (a list)
```

```
## stack ##
print ("The Man")
class stack :
    global tos
    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 is empty")
        else:
            k = self.l[self.tos]
            print ("data =", k)
            self.tos = self.tos - 1
    def peak(self):
        if self.tos <0:
            print ("stack is empty")
        else:
            p = self.l[self.tos]
            print ("Top element", p
S= stack()
```

06/01/22

Pratical - 5

Aim :- Implementation of stacks 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 or a pile. The elements in the stack are added or removed only from one position i.e the top most position. Thus, the stack works on the LIFO (Last in First out) principle. as the element that was inserted last will be removed. First A stack can be implemented using array as well as linked list. stack has three basic operations : push, pop, peek. The operations of adding and removing the element is known as Push and Pop.

Algorithm :-

step 1:- Create a class stack with instance variable. items.

step 2:- Define the init method with self argument and initialize the initial value and then initialize to an empty list.

Step 3:- Define methods push and pop under the class stack.

LA:

Step 4 :- Use if statement to give the condition that if length of given list is greater than the range of list then print stack is full.

Step 5 :- Use Or else print statement as insert the element into the stack and then initializing 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 top most position.

Step 8 :- First condition checks whether the no. of elements are zero while the second case whether tos is assigned any value. If tos is not assigned any value, then make sure that stack is empty.

Step 9 :- Assign the element value in push method to add and print the given value is poped, not.

Code

```
class Queue :
    global r
    global f
    def __init__(self):
        self.r = 0
        self.f = 0
        self.l = [0, 0, 0, 0, 0]
    def add(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")
    def remove(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()
```

Pratical - 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 operation of the queue. It is based on the princip 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 principle (FIFO

Queue () : Creat a new empty queue

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

Dequeue () : Returns the element which was at the front is moved to the successive element A dequence operation cannot remove element of the queue is empty.

Algorithm :

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

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

Step 3 : Use if statement that length is equal to rear then queue is full or else insert the element in empty list or display 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 & 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() & print the list after adding and same for deleting and display the list after deleting the element from the list.

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.
```

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 = stack.pop()
            b = stack.pop()
            stack.append(int(b)/int(a))
    return stack.pop()
s = "8 6 9 * +"
r = evaluate(s)
```

Pratical - 7

Aim : Program on evaluation of given string by using stack in python environment i.e post fix

Theory :

The post fix expression is free of any paranthesis first hen we took case of the priority of th operation in the program. given postfix expression can easily be evaluated using stick reading the expression is a always from left to right in post fix.

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 for given conclusion using if statement.

step 5 : scan the taken list from left to right taken is an operand, convert if from string of an integer and pust the value into 'p'

Step 6 :- If the token is an operation *, /, +, - it will need to operand. pop the ' ' two The first pop i.e the 2nd operand and stand pop in the first operand.

Step 7 :- Perform the arithmetic operation push the result back on the m

output :

[8]

[8, 6]

[8, 6, 9]

[8, 15]

[120]

The evaluated value is : 120

Pratical 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 is a linear fashion but no necessarily continuous. 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 roll no., name, address, etc, whereas next refers to the next node. In case of larger list, if we add/remove any element from the list, all the elements of list has to adjust itself every time we add, it is very tedious task so linked list is used to solving this type of proble

Algorithm :-

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

Step 2 :- The entire linked list can be accessed with the first node of the linked list. The first node of the linked list in Turn is referred by the pointer of the linked list.

Step 3 :- Thus the entire linked list can be transversed using the node which is referred by the head pointer of the linked list

Step 4 :- Now that we know that we can transverse the entire linked list using the head pointer, we should not use it to refer the first node of list only.

Step 5 :- We should not use the head pointer to transverse the entire 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 revert back

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

Step 7 :- We will use this temporary note as a copy of the node we are currently transversing. since we are making temporary node a copy of current node the data type of the temporary node should also be node.

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 addA(self, item):
        new node = node(item)
        if self.S = NONE :
            self.S = new node
        else :
            head = self.S
            while head.next != NONE :
                head = head.next
                head.next = new node
    def addB(self, item):
        new node = node(item)
        if self.S = NONE :
            self.S = new node
        else :
            new node.next = self.S
            self.S = new node
    def display(self):
        head = self.S
        while head.next != NONE :
            print(head.data)
```

```
            head = head.next
            print (head.data)
    def delete (self):
        if self.s = NONE:
            print ["LIST is Empty")
        else:
            head = self.s
        while True:
            if head.next != NONE:
                d = head
                head = head.next
            else:
                d.next = NONE
                break
```

```
s = linked list()
s.addL (50)
s.addL (80)
s.addL (70)
s.addL (80)
s.addL (40)
s.addL (30)
s.display ()
```

output :-

```
20
30
40
50
60
70
```

Step 9 :- But the 1st node is referred 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 while loop.

Step 12 : The last node in the linked list is referred 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 as self.s = none.

Step 14 : 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

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

Pratical - 9

Aim :- Implementation of merge sort.

Theory :- Like Quick sort, mergesort is a divide & conquer algorithm. It divides input array in two halves, calls itself for the two 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 into one.

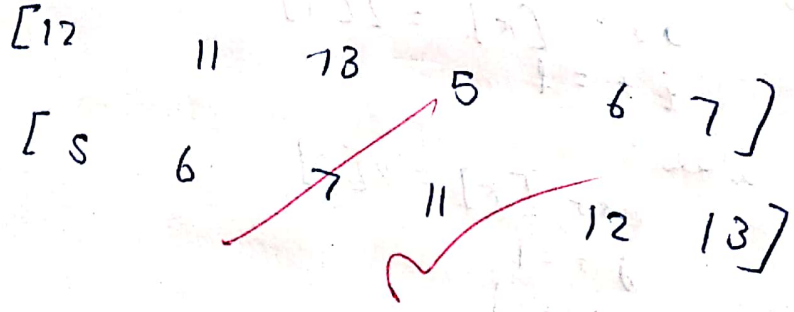
Algorithm.

1. Define the sort [arr, l; m, r].
2. Stores the starting ~~variable~~ starting position of both parts in temporary variables.
3. Check if first part comes to an end or not.
4. Checks if second part comes to an end or not
5. Checks which parts has smaller element.

```
def sort [arr, l, m, r] :
    n1 = m - l + 1
    n2 = r - m
    L = [0] * (n1)
    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 & j < n2
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
            k += 1
        while l < 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 r m, r)
        arr = [ ]
        print [arr]
        n = len (arr)
        mergesort (arr, 0, n-1)
        print (arr)
```

OUTPUT :-



6. Now the real array has element in sorted manner including both parts.

7. Defines the corret array in 2 part.

8. Sort the 1st part of array

9. Sort the 2nd part of array

10. merge the both parts by comparing elements of both the parts

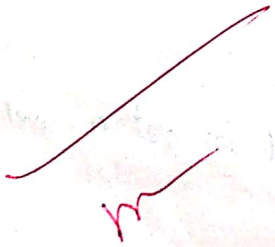
step 3 : Find the union and intersection of above 2
by using & (and), ! (OR) method print the set
of union and info as set 3 and set 4.

step 4 : Use if statement to find out the subset a
superset of set 3 and set 4 display the above

step 5 : Display that element in set 3 and set 4,
set using mathematical

```
# code
print ("Himanshu")
set 1 = set ()
set 2 = set ()
for i in range (8, 15):
    set 1. add (i)
for i in range (1, 12):
    set 2. add (i)
print ("set 1 :", set 1)
print ("set 2 :", set 2)
print ("\n")
set 3 = set 1 / set 2
print ("Union of set 1 & set 2 : set 3", set 3)
set 4 = set 1 & set 2
print ("Intersection of set 1 and set 2 : set 4", set 4)
print ("\n")
if set 3 > set 4 :
    print (" set 3 is superset of set 4")
elif set 3 < set 4 :
    print (" set 3 is subset of set 3")
    print ("\n")
set 5 = set 3 - set 4
print ("Element in set 3 & not in set 4 : set 5", set 5)
print ("\n")
if set 4 isdisjoint (set 5):
    print ("set 4 & set 5 are Mutually Exclusive \n")
set 5. clear()
print (" after applying clear, set 5 is empty set :")
print (" set 5=", set 5)
```

```
# class node :
    global r
    global l
    global data
    def __init__(self, l):
        self.l = None
        self.data = ~~None~~ l
        self.r = NONE
class Tree :
    global root
    def __init__(self):
        self.root = None
    def add(self, val):
        if self.root == None:
            self.root = node(val)
        else :
            newnode = node(val)
            h = self.root
            while ~~root~~ True:
                if newnode.data < h.data:
                    if h.l != None:
                        h = h.l
                    else :
                        h.l = newnode
                        print(newnode.data,
                              "added on the left of",
                              h.data)
                        break
```



Pratical - 11

Aim : Program based on binary search Tree by implementing inorder, preorder & postorder.

Theory: Binary Tree is a tree which support maximum of 2 children for any node within the tree. Thus, any particular node can have either 0 or 1 or 2 children

- Inorder : (i) Traverse the left subtree. The left subtree in turn might have left & right subtrees.
  (ii) Visit the root node
  (iii) Traverse the right subtree and repeat it.
- Preorder : (i) Visit the root node.
  (ii) Traverse the left subtree. The left subtree in turn might have left & right subtree.
  (iii) Traverse the right subtree, repeat it.
- Post order : (i) Traverse the left subtree. The left subtree in turn might have left and right subtrees.
  (ii) Traverse the right subtree
  (iii) Visit the root node.

Algorithm :

Step 1 : Define class node and define init () method with 2 argument. Initialize the value in method.

Step 2 : Again, Define a class BST that is Binary search Tree with init () method with

self arguement 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 left side

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 right side

Step 7: After this, left subtree and right subtree repeat this method to arrange the node according to BST

Step 8: Define Inorder(), Pre order() & Post order() with root argument and use if statment that root is none & return that in all.

Step 9: In inorder, else statement used for giving that condition first left, root & then right node.

```
        else :
            if h.r != None :
                h = h.r
            else :
                h.r = newnode
                print (newnode.data, "added in the right of", h.data)
                break
    def preorder (self, start):
        if start != None:
            print(start.data)
            self.preorder (start.l)
            self.preorder (start.r)
    def inorder (self, start):
        if start != None:
            self.inorder (start.l)
            print (start.data)
            self.inorder (start.r)
    def postorder (self, start)
        if start != None:
            self.inorder (start.l)
            self.inorder (start.r)
            print (start.data)
```

```
T = Tree (1)
T.add (7)
T.add (5)
T.add (12)
```

```
T. add(3)
T. add (6)
T. add(9)
T. add (15)
T. add(1)
T. add(4)
print ("Preorder:")
T. preorder (T.root)
print("Inorder:")
T. inorder (T.root)
print ("Postorder:")
T. postorder (T.root)
```

Output :

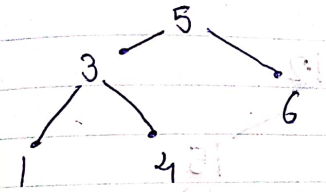
6 added on the left of 7
12 added on left of 7
3 added on right of 5
6 added on right of 5
9 added on left of 12
15 added on right of 12
1 added on left of 3
4 added on right of 3
8 added on left of 9
10 added on right of 9
13 added on left of 15
17 added on right of 15
added on left of
added on right of

Preorder	Inorder	Postorder
7	1	1
5	3	3
3	4	4
1	5	5
4	6	6
6	7	7
12	8	8
9	9	9
8	10	10
10	12	12
15	13	13
13	15	15
17	17	17

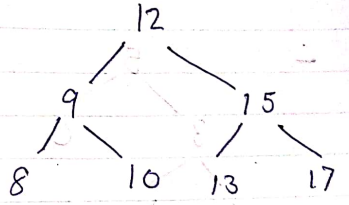
Step 10 : For preorder, we have to give condition in else that first root, left and then right node.

Step 11 : For postorder, we have to give condition in else that first left, right & root.

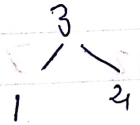
→ Inorder (LVR) :

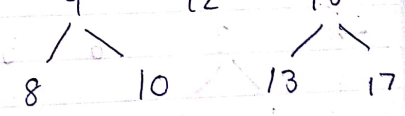


7



→ 3 5 6 7 9 12 15

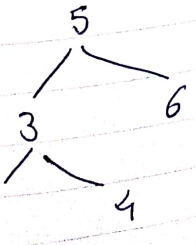


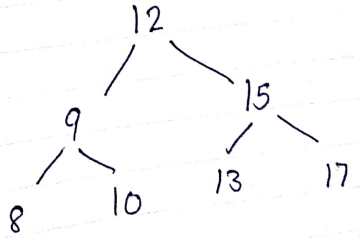


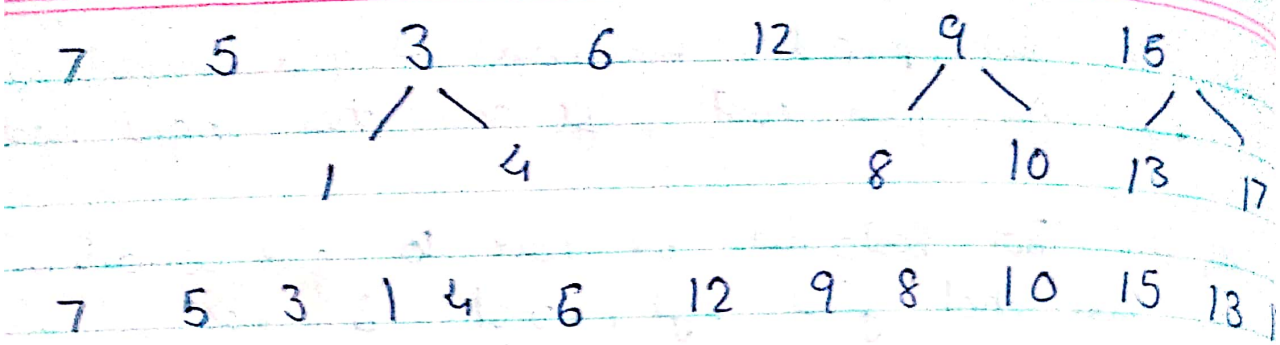
→ 1 3 4 5 6 7 8 9 10 12 13 15 17

→ Preorder (VLR)

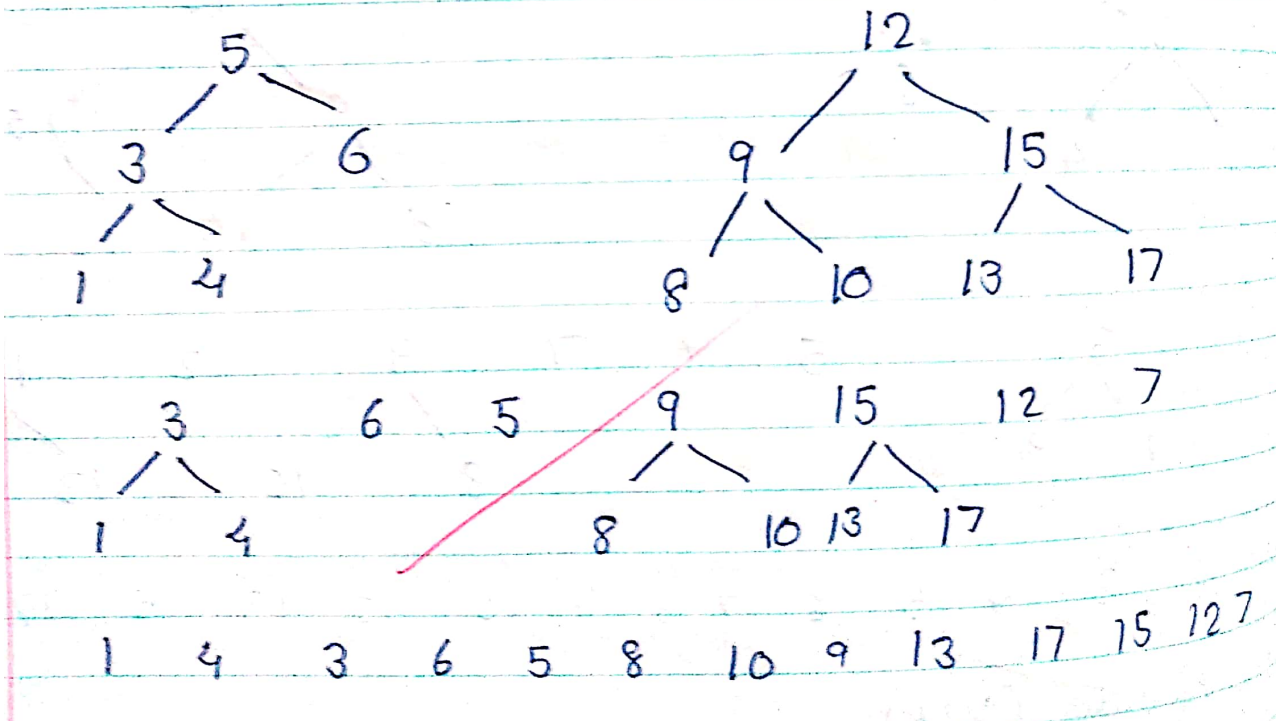
7



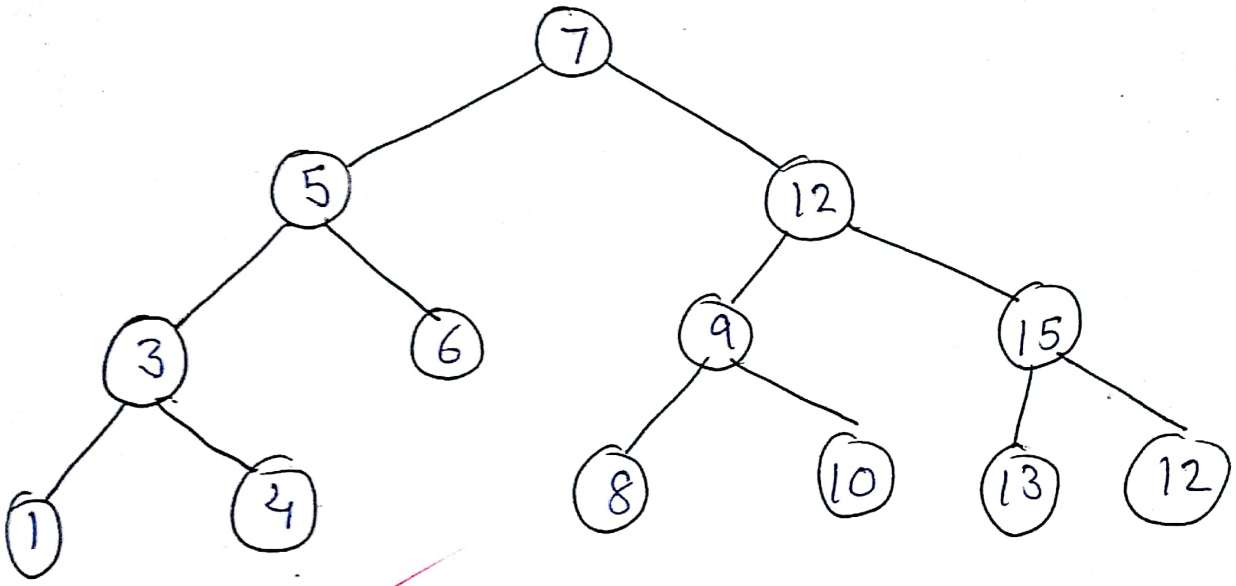




Post order (LRV)



* Binary Search Tree



MS
10/02/2021