Practical no : 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 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 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:

step 1: Create an empty list and assign 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 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 element's position.

Coding

```
print("linear search")
a=[]
n=int(input("enter a range:"))
for s in range(0,n):
    s=int(input("enter a no:"))
    a.append(s)
    print(a)
e=int(input("enter search no:"))
for i in range(0,n):
    if(a[i]==e):
        print("found at position",i)
        break
    else:
        print("not found")
```

Output

```
linear search
enter a range: 5
enter a no: 4
[4]
enter a no: 5
[4, 5]
enter a no: 7
[4, 5, 7]
enter a no: 8
[4, 5, 7, 8]
enter a no: 9
[4, 5, 7, 8, 9]
enter a Search no: 8
found at position: 3
```

Sorted

Coding :-

```
print("linear search")
a=[]
n=int(input("enter a range:"))
for s in range(0,n):
    s=int(input("enter a no:"))
    a.append(s)
    a.sort()
    print(a)
c=int(input("enter search no:"))
for i in range(0,n):
    if(a[i]==c):
        print("found at position",i)
        break
else:
    print("not found")
```

Output:-

```
linear search
enter a range: 5
enter a no: 4
[4]
enter a no: 8
[4, 8]
enter a no: 9
[4, 8, 9]
enter a no: 8
[4, 8, 8, 9]
enter a no: 6
[4, 6, 8, 8, 9]
enter search no: 9
found a position 4
```


Sorted:

Algorithm:

step 1: Create an empty list and assign to a variable

step 2: Accept the total no. of element to be insert into the list from the user say 'n'.

step 3: Use for loop for adding the element into the list

step 4: sort the list, to sort the accepted element and assign in increasing order in the list.

step 5: Print the new list. Use if statement to give the range in which the 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 conditions

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

step 8: Use another if loop to point that the element is not found if the element which is acceptor from user is not their in the list

Aim:- Implement Binary Search to find an searched 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:-

step1:- Create Empty list and assign it to variable.

step2: Using Input method, accept the range of given list.

step3: Use for loop add elements in list using append() method.


```
print ("Binary Search")
a=[]
n=int(input("enter the range:"))
for b in range(0,n):
    b=input("enter no:")
    a=append(b)
    a.sort()
    print(a)
s=input("enter number to searched:")
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+1)/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
```

Output

Binary Search

enter a range: 5

enter no: 4

['4']

enter no: 5

['4', '5']

enter no: 6

['4', '5', '6', '6']

enter no: 8

['4', '5', '6', '6', '8']

enter number to searched 4

0

element found at: 0

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.

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

step 8: Intilizee first to 0 and last to last element of the list. as array is starting from 0 hence it is initialized 1 less than the total count.

step 9: Use for loop & assign the given range.

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

step 11: Else if the item to be searched is still less than middle then

Initialize last(h) = mid(m)-1

Else

Initilize first(l) = mid(m)-1

step 12: Repeat till you found the element stick the input & output of above algorithm.

Practical no: 3

Bubble sort.

Aim: Implementation of Bubble sort program on given list.

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

Algorithm:

Step 1: Bubble sort algorithm shorts by comparing the first two elements of any 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.

Coding:

```
print("Bubble sort")
a=[]
b= int (input ("Enter the range:"))
for s in range (0, b):
    s = input ("enter no:")
    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)
```

Output

Bubble Sort
enter the range: 5
enter no: 4
[4]
enter no: 8
[4, 8]
enter no: 2
[4, 8, 2]
enter no: 3
[4, 8, 2, 3]
enter no 1
[4, 8, 2, 3, 1]
Sorted array [1, 2, 3, 4, 5]

step 3: If the first element is then smaller than second element then we do not swap the elements

step 4: Again second & third elements are compared and swapped if it is necessary and this process goes on until the 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: Display the output of the above algorithm of bubble sort stepwise.

9/12/19

16/12/19 CE

Practical no: 4

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

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

Algorithm:-

step1: 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.

ep2. The partition process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list either less than or greater than pivot value.

03. Partitioning begins by locating two position markers let's call them leftmark & rightmark. at the begining and end of remaining items in the list. The goal of the partition process is to move items that are on wrong side with respect to pivot value while also converging on the split point.

Coding:

```
def quicksort(alist):
    help(alist, 0, len(alist)-1)
def help(alist, first, last):
    if first < last:
        split = part(alist, first, last)
        help(alist, first, split-1)
        help(alist, split+1, last)
def part(alist, first, last):
    pivot = alist[first]
    l = first+1
    r = last
    done = False
    while not done:
        while l<=r and alist[l]<=pivot:
            l = l+1
        while alist[r]>=pivot and r>=1:
            r = r-1
        if r<1:
            done = True
        else:
            temp = alist[l]
            alist[l] = alist[r]
            alist[r] = temp
    temp = alist[first]
    alist[first] = alist[r]
    alist[r] = temp
    return r
```

```
x = int(input("enter a range of list: "))
alist = []
for b in range(0, x):
    b = int(input("enter elements: "))
    alist.append(b)
    n = len(alist)
quicksort(alist)
print(alist)
```

Output

```
enter a range of list : 5
enter element : 8
enter element : 4
enter element : 2
enter element : 9
enter element : 7
[2, 4, 7, 8, 9]
```

16/12/19

step4: We begin by incrementing leftmark will we locate a value that is greater than the p.v. we then decrement rightmark until we find value that is less than the pivot value. At the 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 spilt point.

step6: The pivot value can be exchanged with the content of split point and p.v is now in place.

step7: In addition all the items to left of split point are less than p.v & all the items to the right of split point are greater than pv. The line can be invoked recursively on the two halves.

step8: The quicksort function invokes a Recursive function quicksorthelper

step9: quicksorthelper begins with same based 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, then it can be partitioned and recursively sort

step12: The partition function implement the process described earlier

step13: Display and stick the coding and output of above Algorithm.

Practical :- 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 topmost position. Thus the stack works on the LIFO (Last In First Out) principle as the element that was inserted last will be removed first. A stack can be implemented using array as well as linked list. stack has three basic operation: push, pop, peek. The operations of adding and removing the elements is known as Push & Pop

Algorithm:

Step1: Create a class stack with instance variable tos

Step2: Define the init method with self argument and initialize the initial value and then initialize to an empty list

Code

```
print("Akash Sharma")
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 peek(self):
      if self.tos < 0
          print("stack empty")
      else:
          a = self.l[self.tos]
          print("data =", a)
s.stack()
```

```
>>> Akash Sharma
>>> s.l
[0, 0, 0, 0, 0]
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.l
[10, 20, 30, 40, 0]
>>> s.peek()
data = 40
>>> s.pop()
<bound method stack.pop of <__main__.stacks object at
0x0225 3E80>>
data = 40
>>> s.l
[10, 20, 30, 0, 0]
```

My
06/01/now

step3: Define methods push and pop under the class stack

step4: 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

step5: Or Else print statement as insert the element into the stack and initialize the values

step6: Push method used to insert the element but pop method used to delete the element from the stack.

step7: 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

step8: Assign the element values in push method to and print the given value is poped.

step9: Attach the input & output of above algorithm

step10: first condition checks wheather the no. of element an zero while the second case whether top is assigned any value. If top is not assigned any value then can be store that stack is empty

My

Practical:- 6

Aim:- Implementing a Queue using Python List

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

Queue(): create a new empty queue

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

Dequeue(): Returns the element which was at the front, the front is moved to the successive element. A dequence operation cannot be performed if the queue is empty

Code:-

```
Class Queue:
    global r
    global f
    def __init__(self):
        self.r = 0
        self.f = 0
        self.l = [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("elements deleted:")
            self.f = self.f+1
        else:
            print("queue is empty")

Q = Queue()
```

14

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.
```


Algorithm:-

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

Step2: Define a empty list and define enQueue() method within 2 argument, 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 and increment by 1.

Step4: Define deQuence() with self-argument under this use if statement that front is equal to length of list then display Queue is empty or else give that front is at zero and using that delete the element from front side and increment it by 1.

Step5: Now call the Queue() function and give the element that has to be added in the empty list after adding and some for deleting and display the list after deleting the element from the list.

Practical: no 7

Aim:- Program on Evaluation of given string by using stack in python environment ie postfix

Theory:

The postfix expression is free of any parenthesis further we took care of the priority of the operations in the program. given postfix expression can easily be evaluated using stack 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 and print it

step4: Use for loop to assign the range of string to give conclusion using if statement

step5: Scan the token list from left to right. If token is an operand, convert it from string to an integer and push the value into the stack

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)
```

Output:

[8]

[8, 6]

[8, 6, 9]

[8, 15]

[120]

The evaluated value is: 120.

step 6: If the token is an operation *, /, +, -, ^, it will need to operand pop the 'p' twice. The first pop ie the 2nd operand and stand pop in the first operand.

step 7: Perform the arithemetic operation, push the result back on the m

step 8: When the output expression has been completly processed, the result is on the stack pop the p and return the value

step 9: print the result of string after the evaluation of postfix

step 10: Attach output and input of above algorithm.

20/01/2021

Practical no: 8

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

Theory:- A linked list is a linear data structure which stores the elements in a node in a linear fashion but no necessarily contiguous. The individual element of the linked list called a Node. Node comprises of 2 parts

① Data ② Next. Data stores all the information w.r.t the element of the linked example rollno, 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 problems.

Algorithm:-

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

Step2: The entire linked list can be accessed, the first node of the linked list. The first node of the linked list in turn is referred by the Head pointer of the linked list.

```
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

    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 = linkedlis()
s.addL(50)
s.addL(60)
s.addL(70)
s.addL(80)
s.addB(40)
s.addB(30)
s.addB(20)
s.display()
```

Output

20
30
40
50
60
70
80

step3: Thus, the entire linked list can be transversed using the node which is reffered by the head pointer of the linked list.

step4: Now that we know that we can transverse the entire linked list using the head pointer we should only use it to refer the first node of list only.

step5: We should not use the head pointer to transverse the entired linked list because the head pointer is our only reference to the 1st node in the linked list, modifying the referance of the head pointer can lead to changes which we cannot revert back.

step6: 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

step7: We will use this temporary node as a copy of the node we are currently transversing. Since we are making temporary node a copy of current node the datatype of the tempory node should also be Node

step8: Now that current is referring to the first node, if we want to access 2nd node of list we can refer it to the next node of the 1st node.

step 9: But the 1st node is referred by current. So we transverse to 2nd node as h = h.next.

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

step 11: Our concern now is to find termination cond. for the while loop.

step 12: The last node in the linked list is referred the tail of linked list since the last node of linked does not have any next node. the value the next field of the last node is None

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

step 14: We have to now see how to start transversing the linked list & how to identify whether we reached the last node of linked list or not.

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

Code

Merge sort

```
def sort(arr, l, m, r):
    n1 = m-l+1
    n2 = r-m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0,n1):
        L(i) = arr[l+i]
    for j in range (0,n2):
        R[j] = arr[m+1+j]
    i = 0
    j = 0
    k = l
    while i<n1 and j<n2
        if L[i] <= R[j]:
            arr[k] = L[i]
            i+=1
        else:
            arr[k] = R[j]
            j+=1
            k+=1
        while i<n1:
            arr[k] = L[i]
            i+=1
            k+=1
        while j<n2:
            arr[k] = R[j]
            j+=1
            k+=1
def mergesort(arr, l, r):
    if l<r:
        m = int((l+(r-1))/2)
        merge sort(arr, l, m)
        mergeSort(arr, m+1, r)
        Sort(arr, l, m, r)
```


Practical 9:

Aim:- Implementation of Merge short.

Theory:- Like Quicksort Mergesort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Algorithm:-

Step1: Define the sort (arr, l, m, r):

2) Stores the starting position of both parts in temporary variables

3) Checks iff first part comes to or end or not

4) checks if second part comes to or end or not

5) checks which parts has smaller element

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

7) defines the current array in 2 parts

8) sort the 1st part of array

9) sort the 2nd part of array

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


```
arr: [                    ]
print (arr)
n= len(arr)
mergesort(arr, 0, n-1)
print (arr)
```

Output

[12, 23, 34, 56, 78, ~~45, 86, 48, 42~~

12, 23, 56, 56, 42

[12 11 13 5 6 7]

[5 6 7 11 12 13]

pm

17/02/19

Code:-

```
print("Sharma Akash")
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 and set2: set4", set4)
print("\n")
if set3 > set4:
    print("set3 is superset of set4")
elif set3 < set4:
    print("set3 is subset of set4")
else:
    print("set3 is same as set4")
if set4 < set3:
    print("set4 is subset of set3")
set5 = set3 - set4
print("elements in set3 and not in set4: set5", set5)
print("\n")
if set4.isdisjoint(set5):
    print("set4 and set5 are mutually exclusive")
```


Practical 10

Topic: Implementation of sets by 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 used for adding the element according to given range. Then print the sets after adding

step3: Find the union and intersection of above 2 sets by using & (and), | (or) method. print the sets of union and intersection as set3 & set4

step4: Use If statement to find out the subset and superset of set 3 and set 4 display the above set.

step5: Display that element in set3 is not in set 4 using mathematical operation

step6: Use isdisjoint to check that anything is on element is present or not. If not then display that is is Mutually Exclusive event.

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

```
set5.clear()
print("after applying clear, set5 is empty set:")
print("set5 =", set5)
```

Output:

Sharma Akash

set1: {8, 9, 10, 11, 12, 13, 14}
set2: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

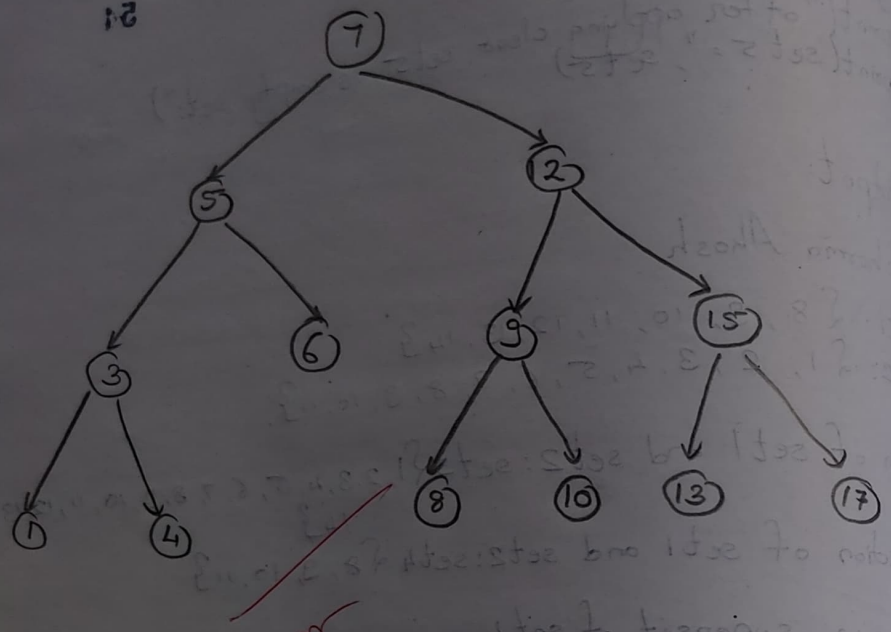
Union of set1 and set2: set3 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}

Intersection of set1 and set2: set4 {8, 9, 10, 11}

set3 is superset of set4
set4 is subset of set3
elements in set3 and not in set4: set5 {1, 2, 3, 4, 5, 6, 7, 12, 13, 14}

set4 and set5 are mutually exclusive
after applying clear, set5 is empty set:
set5 = set()

mr 03/01/2021



Practical : 11

Aim: Program based on Binary Search tree by implementing 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:** i.) Transverse the left subtree. The left subtree intern might have left & right subtrees.
i.) Visit the root node.
ii.) Transverse the right subtree and repeat it.

→ **Preorder:** i.) Visit the root node.
ii.) Transverse the left subtree. the left subtree in turn might have left and right subtree.
iii.) Transverse the right subtree repeat it.

→ **Postorder:** i.) Transverse the left subtree. The left subtree intern might have left & right subtrees.
ii.) Transverse the right subtree repeat it
iii.) visit the root node.

Algorithm:-

step1: Define class node and define init() method with 2 arguement. Initialize the value ito this method

step2: Again, Define a class BST that is Binary search Tree within init() method with self arguement and assign the root is None

step3: Define add() method for adding the node Define a variable p that p=node(value)

step4: Use If statement for checking wheater the root that root is none then use else statement for if no is less than the main node then put or arrange That is leftside.

step5: Use while loop for checking the node is less than or greater than the main node and break the if it is not satisfying.

step6: Use If statement within that else statement for checking that node is greater than main root then put it into rightside

step7: After this leftsubtree & right subtree, repeat this method to arrange the node according to Binary Search Tree

Binary Tree

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

```
        print(newnode.data, "added on right of", t.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)
```

Output

```
5 added on left of 7
12 added on right of 7
3 added on left 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
```

preorder	Inorder	postorder
7	1	1
5	3	3
3	4	4
1	5	5
4	6	6
6	7	8
12	8	9
9	9	10
8	10	12
10	12	13
15	13	15
13	15	17
17	17	7

step8: Define Inorder(), Preorder() & Postorder() with root is none and return that in all

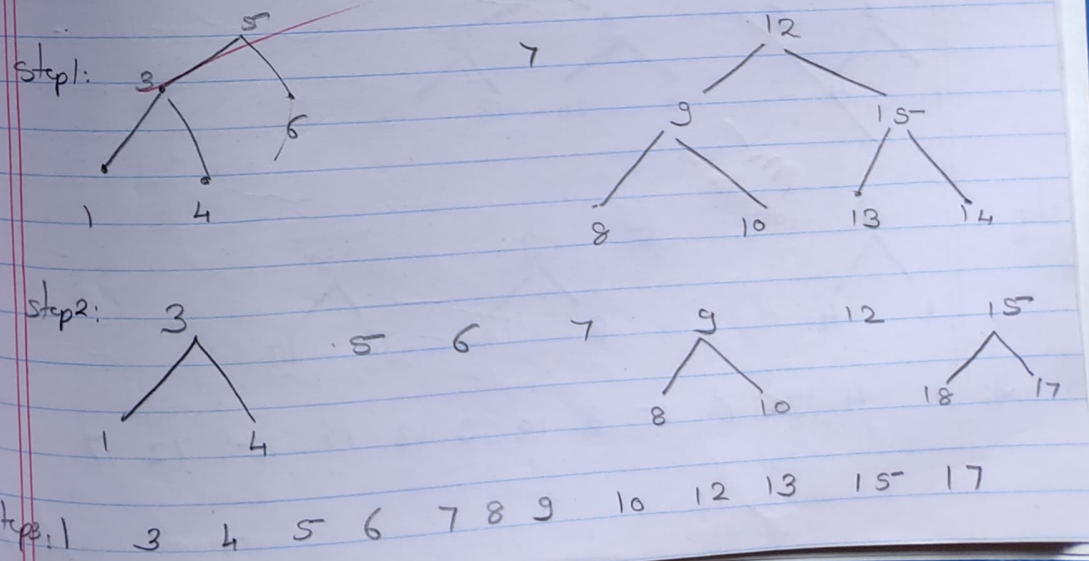
step9: In Inorder, else statement used for giving that condition first left, root and then right node

step10: for Preorder, we have to give condition in else that first root, left and then right node.

step11: for Postorder, In else part, assign left then right and then go for root node

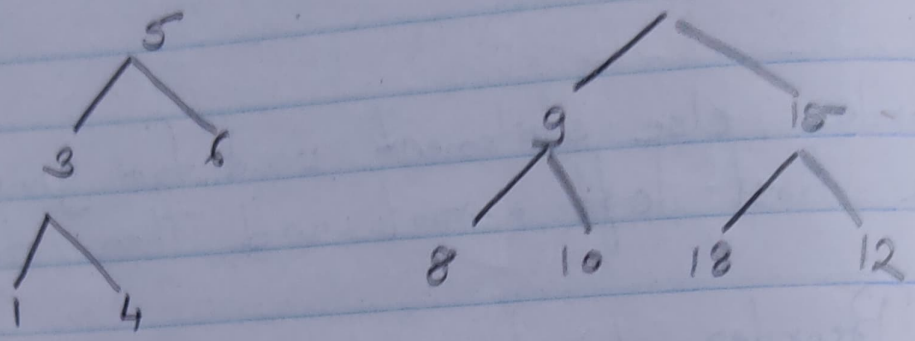
step12: Display the output and input of above Algorithm.

Inorder: (LVR)

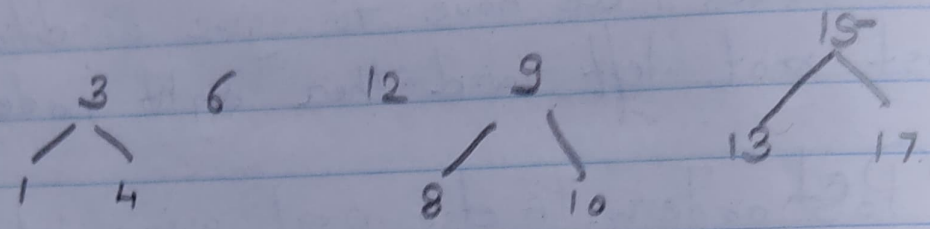


Preorder: (VLR)

step 1: 7



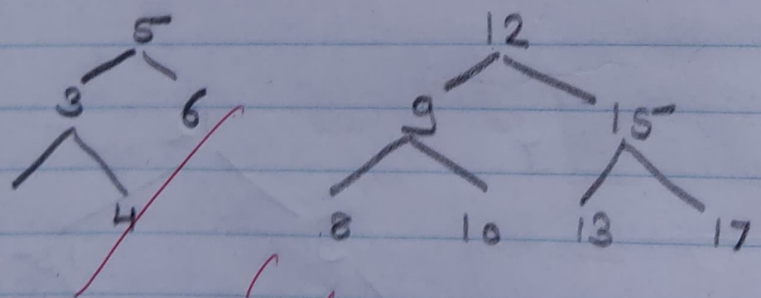
step 2: 5



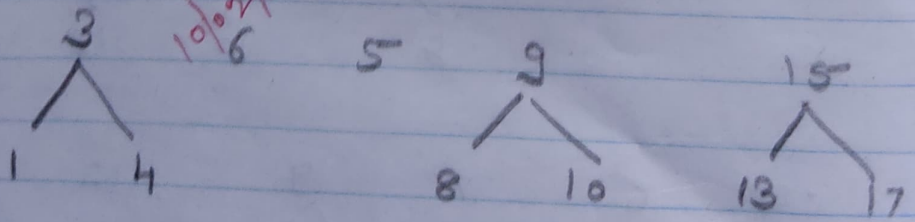
step 3: 7 5 3 1 4 6 12 9 8 10 15 13 17

Postorder (LRV)

step 1:



step 2:



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