DATA STRUCTURE

Practical - 01

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

Theory:

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

It is worst searching algorithm with worst case time complexity. It is 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 & its position is also found.

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

```
>>> Enter element in array: 3 2 4 5 1
>>> Element in array are: [3 2 4 5 1]
>>> Element to be searched: 4
Element found at location 2
```


1] UNSORTED LINEAR SEARCH -

Algorithm:

step 1 - Create an empty list & 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 is 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 the elements position.

step 9 - Use another if loop to print that the element is not found if the element accepted from user is not in the l

Step 10 - Draw the output of given algorithm.

2] SORTED LINEAR SEARCH -

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

Algorithm -

step 1 - Create empty list & 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 it 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 element 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

PROGRAM -

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

```
>>> Enter element in array: 1 2 3 5
>>> Elements in array [1 2 3 5]
>>> Enter element to be searched: 3
>>> Element found at position 2
```

user using 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 in the list.

Step 11 - Attach the input & output of above algorithm.

PRACTICAL - 02

Aim: Binary search implementation to find an element from the list given.

Theory:

BINARY SEARCH

Binary search is also known as half interval search, logarithmic search or binary chop. It 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 & assign it to a variable.

Step2- Using input method, accept the range of given list.

Step3- Use for loop, add elements in list using append() method.

Step4- Use sort() method to sort the accepted element & assign it in increasing ordered list print the list after sorting.

PROGRAM-

```
a= list(input("Enter the list of elements:"))
   a.sort()
n= len(a)
s= int(input("Enter the number to be searched:"))
if (s>a[n-1] or s<a[0]):
   print("Element not found")
else:
   f=0
   l=n-1
   for i in range(0,n):
   m= int((f+l)/2)
   if s== a[m]:
      print("The element is found at:", m)
      break
   else:
      if s<a[m]:
         l= m-1
      else:
         f = m+1
```

>>> Enter the list of elements: 1 5 2 4
>>> Enter the number to be searched: 5
>>> The element is found at: 1

Step5- Use if loop to give the range in which element is found in given range then display a message "Element not found".

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

Step7- Accept an argument & key of the element that element has to be searched.

Step8- Initialize first to 0 & last to last element of the list as array is starting from 0 hence it is initialized 1 less than the total count.

Step9- Use for loop & assign the given range.

Step10- If statement in list & still the element to be searched is not found then find the middle element (m).

Step11- Else if the item to be searched is still less than the middle term then
Initialize last (h) = mid (m) - 1
Else
Initialize first (l) = mid (m) - 1

Step12- Repeat till you found the element stick the input & output of above algorithm.

Step 12- Repeat till you find the element. Stick the output of the above algorithm.

PROGRAM -

```
inp = input ("enter an element:").split()
a = [ ]
for ind in inp:
    a.append (int (ind))
print ("element before sorting", a)
n = len(a)
for i in range(0,n):
    for j in range(n-1):
        if a[i]< a[j]:
            tmp = a[j]
            a[j] = a[i]
            a[i] = tmp
print ("element after sort",a)
```

>>> enter an element : 2 5 8 6
elements before sorting [2, 5, 8, 6]
elements after sorting [2, 5, 6, 8]

Bubble sort

Aim : Implementation of Bubble sort program on a given list.

Theory : Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements & then swapping their position if they exist in the wrong 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 & swapping if necessary.

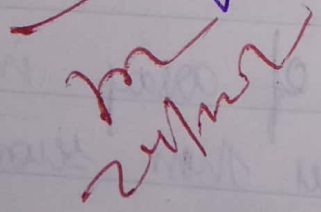
Step 2 - If we want to sort the elements of array in ascending order then first element is greater than second then, we need to swap the element.

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

step 5- Again second & third elements are compared & swapped if it is necessary & this process go on until last & second last element is compared & swapped.

step 6- 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 7- Stick the output & input of above algorithm of bubble sort stepwise.



```
print (" Vidhi Waghela")
class stack :
    global tos
    def __init__(self):
        self.l = [0,0,0,0]
        self.tos = -1
    def push(self,data):
        n = len(self.l)
        if self.tos == n-1:
            print ("stack is full")
        else :
            self.tos = self.tos + 1
            self.l[self.tos] = data
    def pop(self):
        if self.tos<0 :
            print ("stack empty")
        else :
            k = self.l[self.tos]
            print ("data =", k)
            self.l[self.tos] = 0
            self.tos = self.tos - 1
s = stack()
```


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 & removing the element is known as Push & Pop.

Algorithm:

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

Step 2: Define the init method with self argument & initialize the initial value & then initialize to an empty list.

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

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: Or else print statement as insert the element into the stack & initialize the value.

Step 6: Push method used to insert the element 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 with the second case whether tos is assigned any value. If tos is not assigned any value then it can be sure that stack is empty.

output:

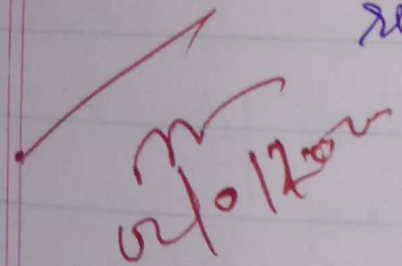
Nidhi Waghela

```
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.l
    [10,20,30,40]
>>> s.pop()
  data = 40
>>> s.l
   [10,20,30]
```

02/01/2020

step 9 : Assign the element values in push method to add & print the given value is poped or not.

step 10 : Attach the input & output of above algorithm.



PRACTICAL -5

Aim : Implement quick sort to sort the given list.

Theory : The quick sort is a secured algorithm then based on the divide & conquer technique.

Algorithm :

Step 1 - Quick sort first selects a value, which is called pivot value element serve as our first pivot value. Since we know that first will eventually end up as last in that list.

Step 2 - The partition process will happen next. It will find the split point & at the same time move other items to the appropriate side of the list either lets than or greater than pivot value.

Step 3 - Partitioning begins by locating two position markers lets call them leftmark & rightmark at the beginning & end of remaining items that are on the wrong side with respect to pivot value while also converging on the split point.

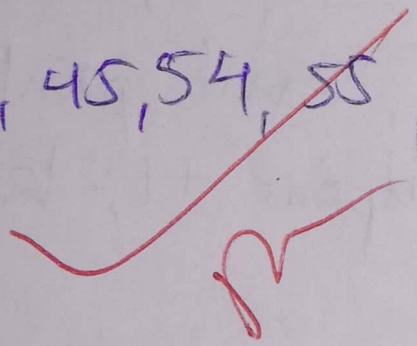
```
def quicksort(alist):
    quicksorthelper(alist, 0, len(alist)-1)
def quicksorthelper(alist, first, last):
    if first < last:
        splitpoint = partition(alist, first, last)
        quicksorthelper(alist, first, splitpoint-1)
        quicksorthelper(alist, splitpoint+1, last)
def partition(alist, first, last):
    pivotvalue = alist[first]
    leftmark = first+1
    rightmark = last
    done = False
    while not done:
        while leftmark <= rightmark and alist[leftmark] = pivotvalue:
            leftmark = leftmark+1
        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark-1
        if rightmark < leftmark:
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark] = alist[rightmark]
            alist[rightmark] = temp
    temp = alist[first]
    alist[first] = alist[rightmark]
    alist[rightmark] = temp
    return rightmark
```

84

```
a list = [42, 54, 45, 67, 89, 66, 55, 80, 100]
quick Sort (alist)
print (alist)
```

OUTPUT -

[42, 45, 54, 55, 66, 89, 67, 80, 100]



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

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

Step 6 - The pv can be exchanged with the content of split point & pv is now in place.

Step 7 - In addition all the items to left of split point are less than p.v & all the items to left to the right of split point are greater than pv. The list can now be divided at split point & quick sort can be invoked recursing on the two values.

Step 8 - The quicksort function invokes a recursive function quick sort helps.

Step - Quick sort helps, begins with same base as the merge sort.

Step 10 - If length of the list is less than 0 or equal one it is already sorted.

Step 11 - If it is greater than it can be partitioned & recursive function.

Step 12 - The partition function implement the process described earlier.

Step 13 - Display & stick the coding & output of above algorithm.

23/11/2020

#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()

"869 * +"
evaluate(s)
print("The evaluated value is :", r)
```

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 operations in the program. A given postfix expression can easily be evaluated using stacks. Reading the expression is always from left to right in Postfix.

Algorithm:

Step 1- Define evaluate as function then creat 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 & print it.

Step 4- Use for loop to assign the range of string Then give condition using if statement.

Step 5 - Scan the token list from left to right. If token is an operand, convert it from a string to an integer & push the value onto the 'P'.

Step 6 - If the token is an operator *, /, +, -, ^, it would need two operands. Pop the 'P' twice. The first pop is second operand & the second pop is the first operand.

Step 7 - Perform the arithmetic operation. Push the result back on the 'n'.

Step 8 - When the input expression has been completely processed the result is on the stack. Pop the 'P' & return the value.

Step 9 - Print the result of string after the evaluation of Postfix.

Step 10 - Attach output & input of above algorithm.

#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")
```

PRACTICAL - 6

Aim: Implementing a Queue using Python list.

Theory: Queue is a linear data structure which has 2 references front & rear implementing a queue using Python list 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 & 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 & similar to that of insertion of linked list using tail.
- Dequeue (): Returns the element which was at the front the front is moved to the successive element. A dequeue operation cannot remove element if the queue is empty.

Algorithm:

Step 1 - Define a class queue & assign global variables then define init() method with self argument in init(), assign or initialize the initial value with the help of self argument.

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

Step 3- Use if statement that length is equal to rear the queue is full or else insert the element in empty list or display that queue element added successfully. & 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 & using that delete the element from front side & increment it by 1.

Step 5- Now call the queue() function & give the element that has to be added in the empty list by using enqueue() & print the list after adding & some for deleting & display the list after deleting the element from the list.

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

#CODE -

```
class node :
    global data
    global next
    def __init__ (self, item):
        self.data = item
        self.next = None
class linked list :
    global s
    def __init__(self):
        self.s = None
    def add (self, item):
        newnode = 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):
        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)
```


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 function but not necessarily contigous. The individual element of the linked list called a node. Node comprises of 2 parts ① Data ② Next. Data stores all the information with 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 problems.

Algorithm:

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

Step 2- The entire linked list means can be accessed the first node of the linked list. The first of the linked list in turn is returned by the Head pointer of the linked list.

Step 3: Thus, the entire linked list can be transversal using the node which is refered 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 only use it to refer the first node of list only.

Step5 - We should not use the head pointer to transverse the entered linked list because the head pointer is our only refrance to the 1st node in the linked list, modifying the referance of the head pointer can lead to changes which we cannot revert back.

Step 6 - We may lose the refrence to the 1st node in our linked list & hence most of our listed 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 node as a copy of the node we are currently transversing. Since we all making temporary node a copy of current node the datatype of the temporary node should also be node.

```
stack = linked list()
```

OUTPUT:

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

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

step 9 - But the 1st node is refered by current. so we can transverse to 2nd nodes as h = h.next.

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

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

step 12 - The last node in the linked list is refered by the tail of linked list. Since the last node of linked list does not have any next node, the value in the next field of the last node is none.

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

step 14 - We have to now see how to start transversing the linked list & how to identify whether we have reached the last node of linked list.

13/02/2020

Step 15 - Attach the coding of input & output of a above algorithm.

PRACTICAL - 9

MERGE SORT

Aim: Implementation of merge sort by using python.

Theory: Merge sort is a divide & conquer algorithm. It divides input array in two halves, calls itself for the two halves & then merges the two sorted halves. The merge (arr, l, m, r) is key process that assumes that [l..m] and arr [m+1, r] are sorted & merges the two settled sub-array.

Algorithm:

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

Step 2- Now begins the sorting process. The i & j iterator traverses the whole lists & makes changes along the way.

Step 3- If the value at j L[i] = is average to the arr [l+i] sort & is incremented. If not then R[j] is chosen.

#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] = R[j]
            j += 1
            k += 1
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
def mergesort(arr, l, r):
    if l < r:
        m = int ((l+(r-1))/2)
        mergesort (arr, l, m)
        mergesort (arr, m+1, r)
        sort (arr, l, m, r
```

```
arr = [12, 23, 34, 56, 78, 45, 86, 98, 42]
print (arr)
n = len (arr)
mergesort (arr, 0, n-1)
print (arr)
```

Output

```
>>>
[12, 23, 34, 56, 78, 45, 86, 98, 42]
[12, 23, 56, 42, 45, 78, 86, 98]
```


step 4 - This way, the values being assigned through [l + i] are all sorted.

step 5 - At the end of this loop, one of the halves may not have been traversed completely. All values are simply assigned to the remaining slots in the list.

step 6 - Thus, the merge sort has been implemented.

PRACTICAL - 10

SETS

Aim: Implementation of sets using python.

Algorithm:

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

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

Step 3- Use of statment to find out the subset & super set of set 3 & set 4. Display the above set.

Step 4- Display that element in set 3 is not in set 4 using mathematical operation.

Step 5- Use is disjoint( ) to check that anything is common or element is present or not. If not then display that it is mutually exclusive event.

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 and set2 : set4", set4)
print("\n")
if set3 > set4:
    print("set3 is superset of set4")
elif set3 < set4:
    print("set3 is same of set4")
if set4 < set3:
    print("set4 is subset of set3")
    print("\n")
set5 = set3 - set4
print("Elements in set3 & not in set4:
                    set5, set5)
print("\n")
```

if set 4 is disjoint (set 5):
print ("After applying clear, set 5 is empty set :")
print ("set 5 = ", set 5)

Output :

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

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

Binary search Tree

Aim: Implementation of binary search tree using python & 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, 1 or 2 children. There is another identity of binary tree that it is ordered such that one child is identified as left child & other as right child.

1) Inorder: (i) Traverse the left sub-tree. The left sub tree inturn might have left & right sub trees.
ii) Visit the root node.
iii) Transverse the right subtree & repeat it.

2) Pre order: i) Visit the root node.
ii) Transverse the left sub tree. The left sub tree.
iii) Transverse the right sub tree. repeat it.

3) Post order: i) Transverse the left sub tree. The left subtree in turn might have left & right sub trees.
ii) Transverse the right sub trees.
iii) Visit the root node.

CODE:


```
class node:
    def __init__(self, value):
        self.left = None
        self.value = value
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def add(self, value):
        p = node(self, value):
        if self.root == None:
            self.root = p
            print("Root is added sucessfully", p.val)
        else:
            h = self.root
            while True:
                if h.left == None:
                    h.left = p
                    print(p.val, "Node is added to leftside
                                successfully")
                    break
                else:
                    h = h.left
            else:
                if h.right == None:
                    h.right = p
                    print(" p.val, "Node is added to
                                rightside sucessfully")
                    break
```

```
            else :
                h= h.right
        def  Inorder (root):
            if root == None:
                return
            else:
                Inorder(root.left)
                print(root.val)
                Inorder(root.right)
        def  Preorder (root):
            if root == None:
                return
            else:
                print (root.val)
                Preorder (root.left)
                Preorder(root.right)
        def  Postorder (root):
            if root == None:
                return
            else:
                Postorder (root.left)
                Postorder (root.right)
                print (root.val)
    t = BSTC()
```

OUTP

```
>>> t.add(1)
>>> t.add(2)
>>> t.add(3)
>>> t.add(4)
>>> t.add(5)
```

Algorithm :

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

Step 2 - Again, define a class BST that is Binary search Tree with in it() method with self argument & assign the root is node.

Step 3 - Define add() method for adding the node. Be

Step 4 - Use if statement for checking the condition that root is none than use else statement. For if node is less than the main node then put or arrange that in leftside.

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

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

Step 7 - After this, left sub tree & right sub tree repeat this method to arrange the node according to Binary search Tree.

Step 8- Define Inorder(), Preorder() & Postorder() with root argument & use if statment that root is none & return that in all.

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

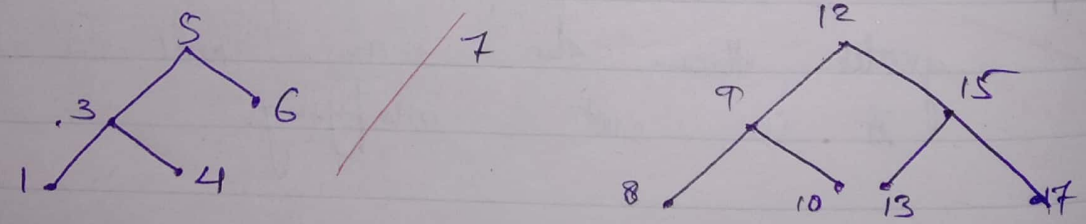
Step 10- For preorder, use have to give condition in else that first root, left & then right node.

Step 11- For postorder, in case part, assign left ~~that~~ then right & then go for root node.

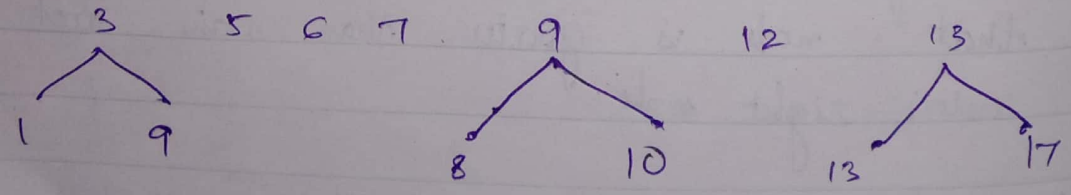
Step 12- Display the output & input.

- Inorder: (LVR)

Step 1 :



Step 2 :

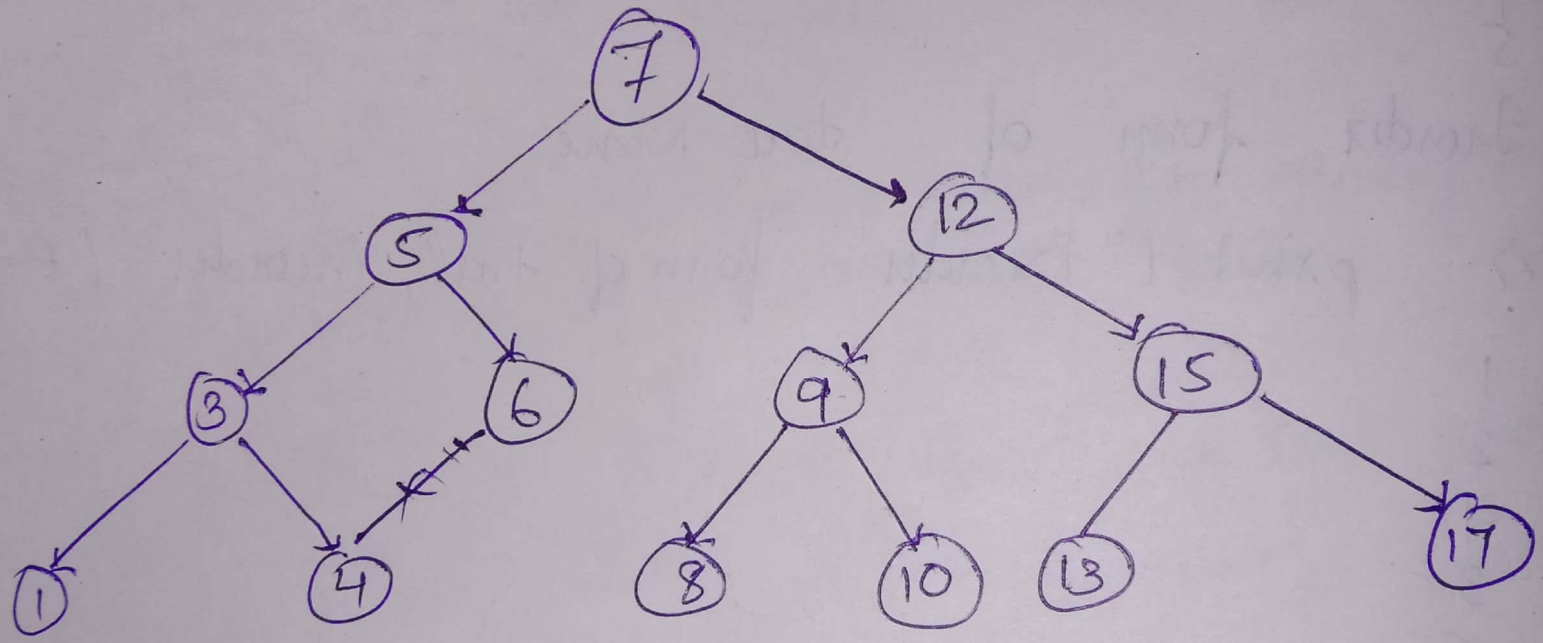


Step 3 : 1 3 4 5 6 7 8 9 10 12 13 15 17

OUTPUT -

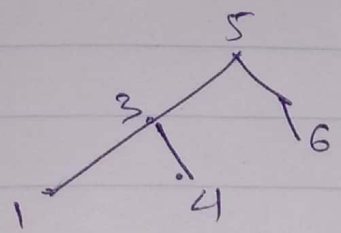
```
>>> print ("\n Inorder form of tree", Inorder(t.root))
1
2
3
4
5
Inorder form of tree None
>>> print ("Preorder form of tree", Preorder (t.root))
1
2
4
3
5
Preorder form of tree None
>>> print ("Post order form of tree", Post order (t.root))
3
5
4
2
1
Postorder form of tree None
```

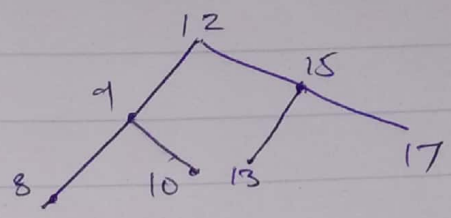
BINARY SEARCH TREE



- Pre order : (VLR)

step 1 : 7

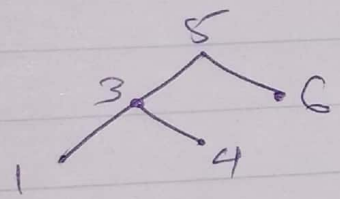


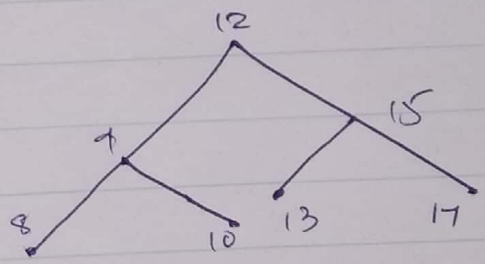


step 2 : 7 5 3 1 4 6 12 9 10 15 13 17

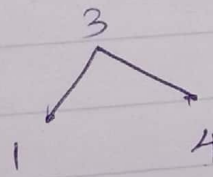
- Post order : (LRV)

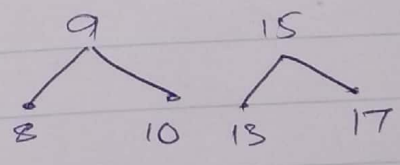
step 1 :





step 2 : 6 5





12 7

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

20/02/2025