PRACTICAL NO 1

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

Theory:

Linear Search

Linear search is one of the impliest 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 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 return that element found and its possible is also found.

1] Unsorted

Algorithm:

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

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

Step 3: Use for loop for adding the element into the list.

Step 4: Print the new list.

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

Step 6: Use for loop in range from '0' to the total no of statement to search the element from the list.

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

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

```
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)
    print (a)
c = int (input ("Enter a number to be searched"))
for i in range (0, n)
  if [a[i] == c)
    print ("found at position" i)
    break
  else:
    print ("not found")
```

Output:

linear search
enter search : 3
enter a number 1
[1]
enter a number 3
[1, 3]
enter a number 4
[1, 3, 4]
Enter a number to be searched 3
found at position 1

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

Step 10: Draw the output of given algorithm.

2] Sorted linear search

Sorting around mean to arrange the element in increasing or decreasing order.

Algorithm:

Step 1: Create empty list and assign it to 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 element in the list.

Step 4: Use sort ( ) method to sort the accepted element and assign in increasing order the list then print 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: The use of else statement, if element is not found in range to statisfy the given condition.

Step 7: Use for loop in range from 0 to the total number of element to be searched and search no from user using input statement.

Step 8: Use if loop that the element 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.

Step 10: Use another if loop to print that the element is not found if the element is not there in the list.

Step 11: Attach the input and output of above algorithm.

PRACTICAL NO 2

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

Theory:

Binary Search

Binary search is also known as f- intervals search, logarithmic search or binary chop is a search algorithm that find 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 than you need to search entire list in linear search which is time consuming. This can be avoided by using binary fashion search.

Algorithm:

Create empty list and assign it to variable.

Using input method, accept the range of given list

Use sort() method to sort the accepted element and assign it in increasing list after sorting.

```
a = []
n = int (input ("Enter the range"))
for b in range (0, n):
    b = int (input ("Enter the numbers"))
    a = append (s)
    a = sort ()
    print (a)
s = int (input ("Enter the number to search"))
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.
```

Output:

```
Enter the range : 4
Enter the number : 2
[2]
Enter the number : 8
[2, 8]
Enter the number : 6
[2, 8, 6]
Enter the number to be searched : 4
Element found at : 1
```

Use if loop to the given range in which element is found. Then use else format, if statements is not found in range then satisfy.

Accept one argument and key of the element that has to be searched.

Use for loop and assign the given range.

If statement in list and still the element to be searched is not found then find the middle element.

Else if the item to be searched is still less than the middle term.

Repeat this you will find the element. Stick the input and output of above algorithm.

PRACTICAL NO 3

Aim:
Implementation of bubble sort program on given list.

Theory:

Bubble - sort

Bubble sort is based on the idea of repeatedly comparing pairs of adjacent element and then swapping their position if they exist in the wrong order.

Algorithm:

1] Bubble sort algorithm starts by comparing the first two element of an array and swapping if necessary.

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

3] Again second and third element are compared and swapped if it is necessary and this process goes on until last and second last element is compared and support.

4] If there are n elements to be sorted then the process mentioned above should be repeated.

5] Stick the output and input of above algorithm of bubble - sort stepwise

```
print ("Bubble sort algorithm")
a[]
b = int (input ("Entee number of elements:"))
for s in range (0, b)
    s = int (input ("Enter the elements:"))
    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 ("Element after sorting are:", a)
```

Output

```
Bubble sort algorithm
Enter number of element: 4
Enter the elements: 3
[3]
Enter the elements: 1
[3,1]
Enter the elements: 5
[3,1,5]
Elements after sorting are [1,3,5]
```

PRACTICAL NO 4

QUICK SORT

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.

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

Step3: Partitioning begins by locating two positions markers. lets call them leftmark and rightmark at the beginning 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.

Step4: We begin by incrementing 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

P.V. We then decrement rightmark 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 rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point.

Step 6: The pivot value can be exchanged with the contents of split point and p.v (pivot value) is now in place.

Step 7: In addition, all the items to left of split point are less than PV & all the items to the right of the split point are greater than P.V. The line can now be divided at split point & quick sort can be invoked recursively on the two halves.

Step 8: The quicksort function invokes a Recursive function, quicksorthelper.

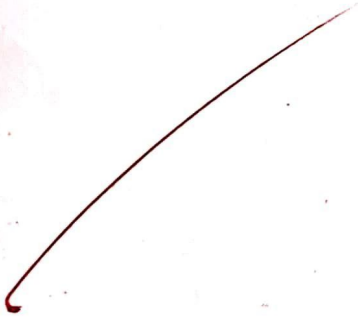
Step 9: Quicksorthelper begins with same base as the merge sort.

Step 10: The partition function implements the process described earlier.

Step 11: Display and stick the coding and output of above algorithm.

Sakshi

```
>>> s.push(20)
>>> s.l
[20, 0, 0, 0, 0]
>>> s.pop()
Data = 20
>>> s.l
[0, 0, 0, 0, 0]
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.push(50)
>>> s.l
[10, 20, 30, 40, 50]
```



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 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 and pop.

Algorithm:

S1: Create a class stack with instance variable items.

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

3. Define methods push and pop under the class stack.

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.

5. Or else print statement as insert the element into the stack and initialize the values.

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

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

8. First cond checks whether the no of elements are zero while the second case whether tos is assigned any value. If top is not assigned any values, then can be sure that stack is empty.

9. Assign the element values in push method to and print the given value is popped rort.

10. Attach the input and output of above algorithm.

```
print "sakshi"
class stack:
    global tos
    def __init__(self):
        self.l = [0,0,0,0,0,0,0]
        self.tos = 1
  def push(self,data)
   n = len(self.l)
   if self.tos = n-1:
      print "stack is full"
  else:
    self.tos = self.tos + 1
    self.l[self.tos] = data
 def pop[self]
   if self.tos < 0:
      print "stack is empty"
   else:
    k = self.l[self.tos]
    print "Data = ", k
    self.l[self.tos] = 0
    self.tos = self.tos - 1
def peek [self]
   if self.tos < 0!
      print ("stack empty")
      else:
        p = self.l(self.tos)
        print("top element = ", p)
S = stack()
```

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

PRACTICAL NO. 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 principle 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 at a data structure based on first in first out FIFO principle.

Queue(): Creates a new empty queue.

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

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

Algorithm:

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

Step 2: Define a empty list and define enQueue() method with 2 arguments. 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 and 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() and 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
```

PRACTICAL NO 7

evaluation of postfix expression

Aim: Program on evaluation of given string by using stack in python environment.

Theory: The postfix expression is free of any parathensis further we took are priorities of the operators in the program. A given postfix expression can easily be evaluated using stacks. Reading the expression is always from left to right in Postfix.

Algorithm:

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 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 and push the value onto the 'p'.

Step 6: If the token is an operator +, /, *, -, ^ it will need two operands. Pop the 'p' twice. The first pop is second operand and the second pop is the first operand.

Step 7: Perform the Arithmetic operation. Push the result back on the 'm'

Step 8: When the input Expression has been completely 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 the output and input of the above algorithm.

Output:

The evaluated value is 27

Sakshi

```
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):
```

PRACTICAL NO 8

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

Algorithm:

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

2] The entire linked list means can be accused as the first nades of the linked list the first node of the linked list in term of referred by the Head pointer of the linked list.

3] Thus the entire linked list can be traversed using the nodes which is referred by head pointer.

4] Now that we know that we can traverse the entire linked list using the head pointer we should only use it to refer the first node of list only.

5] We should not use the head pointer to transverse entired linked list because the head pointer. Due only reference to the 1st node in the linked list, modifying the reference of the head pointer can lead to changes which cannot revert back.

6] We may lose the reference to the 1st node in our linked list and hence most of one list so in order to avoid making some unwanted changes to the 2nd node we will use the temporary node to transverse.

7] We will use this temporary node as a copy of the node we are currently traversing since we are making temporary node a copy of current node the datatype of the temporary node should also be node.

8] But the 1st node is referred by current. so we can transverse to 2nd node as next

9] Similarly we can transverse rest of nodes in the linked list using same method by while loop.

10] One concern now is that to find terminating condition for while loop.

11] The last node in the linked list is referred to 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.

12] We can refer to the last of linked list self.s node

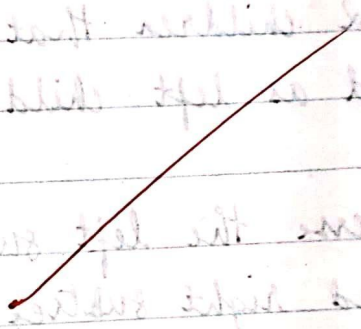
```
        head = self.s
        while head.next != None
            print (head.data)
            head = head.next
            print (head.data)
start = linkedlist ( )
start.add L (50)
start.add L (60)
start.add L (70)
start.add L (80)
start.add B (40)
start.add B (30)
start.add B (20)
start.display ( )
print ("Sakthi")
```

Output:

```
20
30
40
50
60
70
80
Sakthi
```

13] We have to now see how to start traversing the linked list how to identify whether we have reached the last node of the linked list or not.

14] Attach the coding or input and output of above algorithm.



02 - PRACTICAL NO 9

Aim: Program based on binary search by implementing Inorder, Preorder and Postorder transversal.

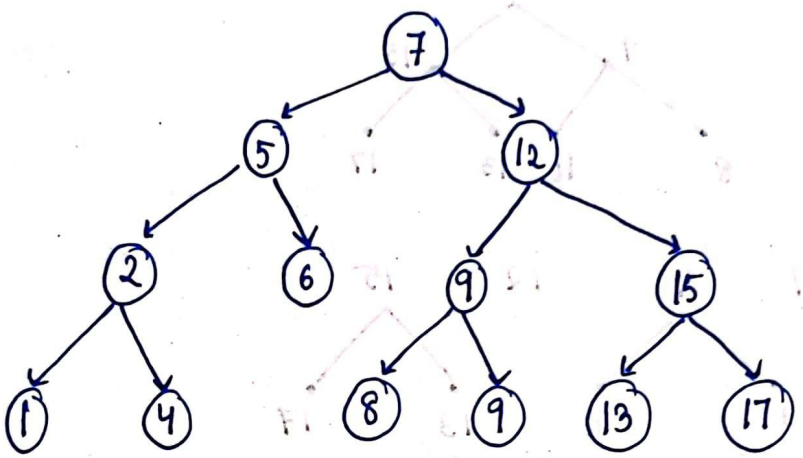
Theory:
Binary tree is a tree which supports maximum of 2 children for any node within the Tree. Thus any particular node can have 0 or 1 or 2 children that is ordered such that one child is identified as left child and other as right child.

* Inorder: (i) Transverse the left subtree, the left subtree intern might have left and right subtree.
(ii) Visit the root node
(iii) Transverse the right subtree and repeat it.

* Preorder: (i) Visit root node
(ii) Transverse the left subtree. The left subtree intern might have left and right subtree.
(iii) Transverse the right subtree repeat it.

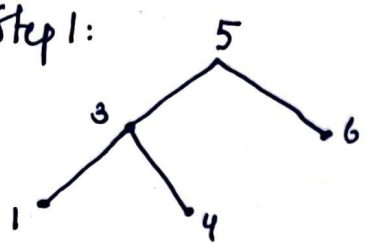
* Postorder: (i) Traverse the left subtree. The left subtree intern might have left and right subtree.
(ii) Traverse the right subtree.
(iii) Visit the root node.

Binary Search Tree

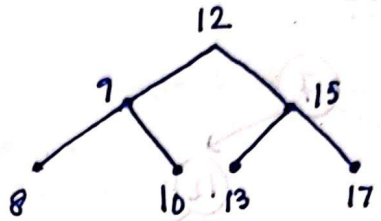


* Inorder : (LVR)

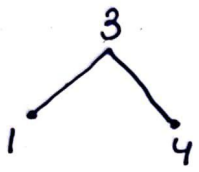
Step 1:



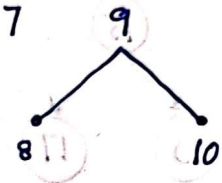
7



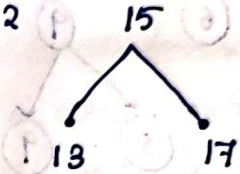
Step 2:



5 6 7



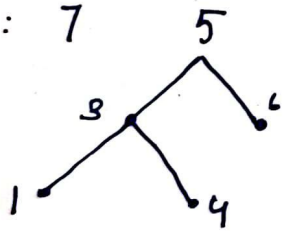
12

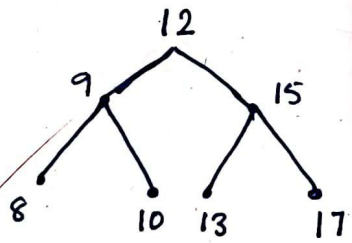


Step 3: 1 3 4 5 6 7 8 9 10 12 13 15 17

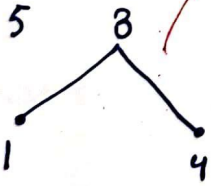
* Preorder: (VLR)

Step 1: 7

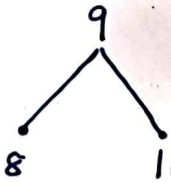


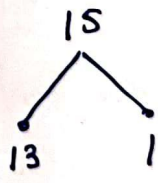


Step 2: 7 5



6 12





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

Algorithm:

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

2] Again Define a class BST that is Binary Search Tree with init ( ) method with self argument and assign the root is none.

3] Define add ( ) method for adding the node. Define a variable p that p = node (value)

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

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

6] Use if statement within that else statement from checking that node is greater than main node side and put it into right side.

7] After this, left subtree and right subtree merge and use this Method to arrange the node according to the Binary Search Tree.

8] Define Inorder (), Preorder () and Postorder () with root argument and use if statement that root is none and return that in all.

9] Inorder else statement used for getting that condition first left root and then right node.

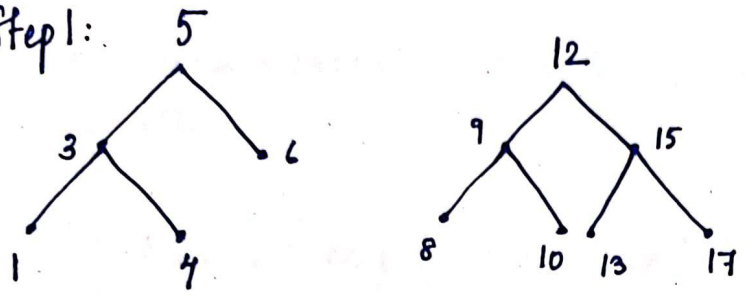
10] For Preorder, we have to give condition in else that first root not, left and then right node.

11] For Postorder, In else part, assign left then right and then go for root node.

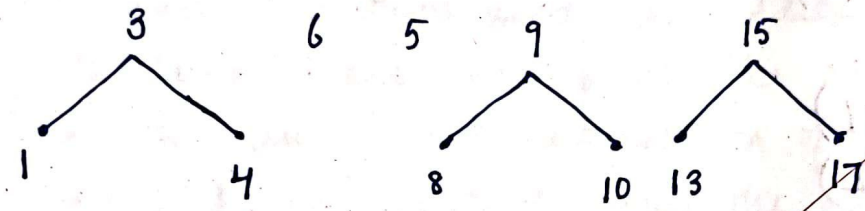
12] Display the output and input of above algorithm.

* Postorder : (LRV)

Step 1:



Step 2:



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

code

```
print ("Sakshi 1832")
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 set 1 and set 2 : set3", set3)
set4 = set1 and set2
print ("Intersection of set1 and set2 : set4 ", set4)
print ("\n")
if set3 > set4:
    print ("set3 is support of set4")
elif set3 < set4:
    print ("set3 is same as set4")
else:
    print ("set3 is same as set4")
if set4 < set3:
    print ("set4 is subset of set3")
    print ("\n")
set5 = set8 - set4
print ("Element in set3 & not in set4: set5", set5)
```

PRACTICAL NO 10

Aim : Implementation of sets using python

Algorithm :

1] Define 2 empty sets 0 and 1 and set 2 now. Use for statement providing the range of above 2 sets.

2] Now add ( ) method used for addition the element according to given range then print the after addition.

3] find the union and intersection of above 2 sets by using h (and), & (or) method. Print the sets of union and intersection of set 3.

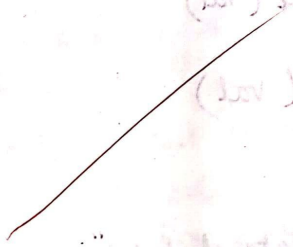
4] Use if statement to find out the subset and superset of set 3 and set 4. Display the above set.

5] Display element in set 3 is not in set 4 using mathematical operation.

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

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

```
print ("\n")
if set4 is disjoint (set5):
    print (" set4 & set5 are Mutually Exclusive \n")
set5.clear()
print (" after applying clear, set5 is empty set: ")
print (" set5 = ", set5)
```



```
class node:
    globe r
    globe l
    def __net__ (self, l):
        self.l = None
        self.delt.a = l
        self.r = None
class Tree:
    global root
    def __init__ (self):
    if self.root = = None
        self.root = node (val)
    else:
        new node = node (val)
        h = self.root
        while True:
            if newnode."data ch data":
                if h.l l = None:
                    h = h.l
                else:
                    h.l = newnode
                    print (newnode.data, "data on left", h.data)
                    break
            else:
                if h.r ! = None
                    h=h.r
                else:
                    h.r = newnode
                    print (new node.data, "added to right", h.data)
                    break
```

PRACTICAL NO 11

Aim: Program based on binary search tree by implementing inorder, preorder and postorder.

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.

- Inorder : (i) Traverse the left subtree. The left subtree inturn might have left and 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 intern might have left & right subtree.

(iii) Traverse the right subtree, repeat it.

- Post order : (i) Traverse the left subtree. The left subtree inturn 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 arguments. Initialize the value in this method.

Step 2: Again define a class BST that is Binary Search Tree with init () method with self argument 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 the use else statement for if node is less than the main node then put or arrange them in left side.

Step 5: Use while loop for checking the node is less than or greater than the main node 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.

```
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 dalete)
        self inorder ( start . r)
def postorder ( self , start ):
    if start ! = None:
        self . inorder ( self . l)
        self . inorder ( self . r)
        print ( start . data)
T = Tree ( )
T = add (100)
T = add ( 80 )
T = add ( 10)
T = add ( 85)
T = add ( 10)
T = add ( 78)
T = add ( 60)
T = add ( 80)
T = add ( 80)
T = add ( 15)
T = add ( 12)
print ( "preorder")
T. preorder ( T. root)
print ( "inorder")
T. inorder ( T. root)
```

Outputs:-

80 added on left of 100
70 added on left of 80
85 added on left of 70
10 added on left of 10

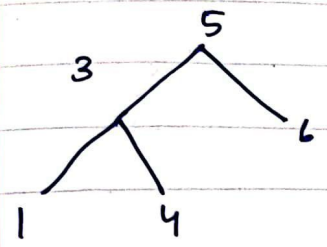
Preorder

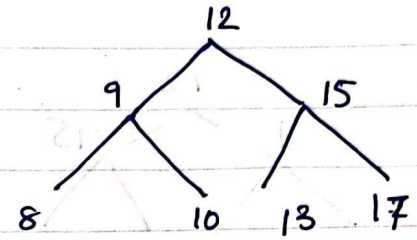
100
80
70
10
15
17
85

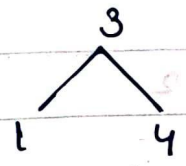
Inorder

10
15
17
70
85
100

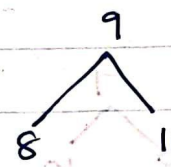
Inorder (LVR)



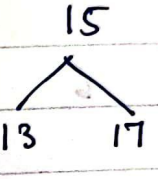




5 6 7



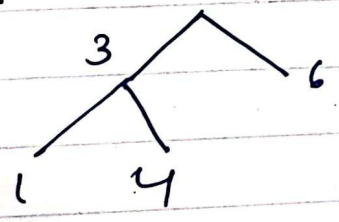
12

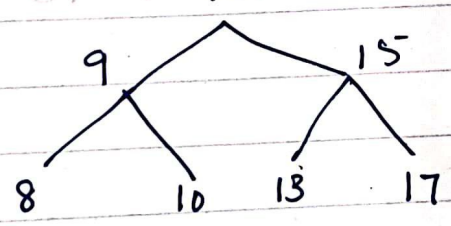


1 3 4 5 6 7 8 9 10 11 12 13 15 17

Preorder (VLR)

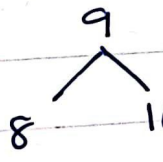
7

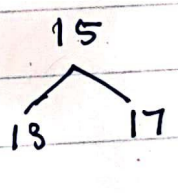




7 5

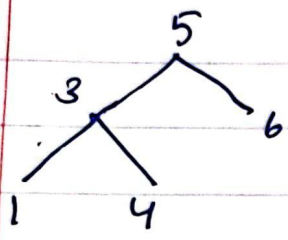
6 12

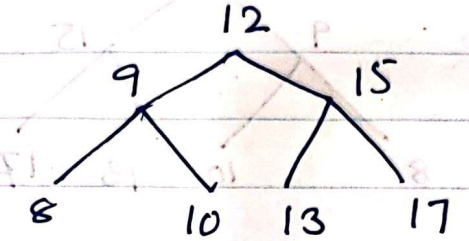


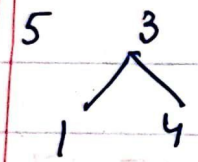


7 5 3 1 4 6 12 9 8 10 15 13 17

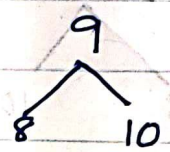
Postorder (LRV)

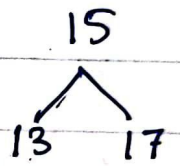






12





12

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

1 4 3 6 5 8 10 9 13 17 15