Practical I

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, in case 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 technique to compare each & every element with the key element to be found, if both of them matches, the algorithm returns that found element, and its position is also found.

i] Unsorted Linear Search.

Algorithm:

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

Step 2 : Accept 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 use print statement to declare that the element is found along with elements position.

Coding:

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

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

Output:

① Linear Search
Enter the range : 5
Enter element 1
[1]
Enter element 2
[1, 2]
Enter element 3
[1, 2, 3]
Enter element 4
[1, 2, 3, 4]
Enter element 5
[1, 2, 3, 4, 5]
Enter search number 5
Found at position 4.

Coding :-

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

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.

ii] Sorted Linear Search:

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

Algorithm:

Step 1: Create an 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 & using .append(), add the elements in the list.

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

Step 5: Use 'if' statement to give the range in which element is found in given range then display "Element 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 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 position.

Step 10: 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 11: Attach the input & output of above algorithm.

Output:

```
Linear Search
Enter the range 5
Enter element 7
[7]
Enter element 5
[5, 7]
Enter element 6
[5, 6, 7]
Enter element 2
[2, 5, 6, 7]
Enter element 1
[1, 2, 5, 6, 7]
Enter Search number 6
Found at position 3.
```

2/12/19

Coding :

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

Practical II

Aim: Implement Binary Search to find a number in the list.

Theory:

Binary Search

Binary search, also known as half interval search, logirithmic search or binary chop is a search algorithm that finds the position of a target value within a sorted array.

If you're looking for the number which is at the end of the list, then you need to search entire list in linear search, which is time consuming. This can be avoided by making use of binary fashion search.

Algorithm:

Step 1: Create empty list & assign it to a vairable.

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

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

Step 4: Use sort() to sort the accepted element and assign it in increasing order. Print the list.

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

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

Step 7: Accept an argument & key of element that element has to be searched for.

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

Step 9: Use for loop & assign the range.

Step 10: If statement in list and still the element to search is not found then find the middle element (m).

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

Step 12: Repeat till you find the element. Display the output of the above algorithm.

Output:

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

Coding :

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

9/12/19

Practical III

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

Theory:

Bubble Sort

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

Algorithm:

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

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

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

Step 4: Again second & third elements are compared & swapped if necessary & this process goes on until last & second last elements are compared.

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: Stick the output & input of above algorithm of bubble sort stepwise.

Output:

```
Bubble Sorting
enter no of elements : 5
enter elements = 8
[8]
enter elements = 5
[8, 5]
enter elements = 10
[8, 5, 10]
enter elements = 2
[8, 5, 10, 2]
enter elements = 3
[8, 5, 10, 2, 3]
Elements after sorting : [2, 3, 5, 8, 10]
```

9/12/19

Coding :-

```
def quick sort (a list):
    help (a list, 0, len(a.list -1)

def help (a list, first, last):
    if first < last :
        split = part (alist, first, last)
        help (alist, first + 1, last)
        help (alist , split + 1, First)
def part (alist , First, last) :
    pivot = a list [First]
    l = first + 1
    r = last
    done = False.
    while not done :
        while l <= r and list [1] <= pivot :
            l+= 1
        while a list [r] >= pivot and r >= 1 :
            r -= 1
        if r < 1 :
            done = True
        else :
            temp = alist [1]
            alist [1] = 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 element : "))
    alist. append (b)
    n = len (alist)
```

Practical IV

Topic : 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 :

Step 1 : Quick sort first selects a value, which is called pivot value, first element serves as our first pivot value, since we know that the first will eventually end up as last in that list.

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

Step 3 : Partitioning begins by locating two position markers let's call them leftmark & 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 the wrong side with respect to pivot value while also converging on the split point.

Step 4: 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 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 content of split point and PV 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 split point are greater than PV. The list can now be divided at split point & quick sort can be invoked recursively on the 2 halves.

Step 8: The quicksort function invokes a recursive function, quicksort.

Output:

```
Enter range for the list : 4
enter the element : 6
enter the element : 8
enter the element : 4
enter the element : 9
[4,4,6,9]
```

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

Step 10: If length of the list is less than 0 equal to one, it is already sorted.

Step 11: If it is greater, then it can be partitionized & recursively sorted.

Step 12: The partition function implements the process described earlier.

Step 13: Display and stick the coding and output of the above algorithm.

Practical V

Aim: Implementation of stack 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 element in the stock are added or removed only from one position, i.e. the top most position. Thus, the stack works on the LIFO (Last In First Out) principle as the element that was inserted last will be removed first. A stack can be implemented using array as well as linked list stack has three basic operation of adding and removing the elements 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 and initialize the initial value and the initialize to an empty list.

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

```
# stack
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 += 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)
            k = self.l[self.tos] = 0
            self.tos -= 1.
    def peek(self):
        if self.tos < 0:
            print("stack is empty")
        else:
            a = self.l[self.tos]
            print("data =", a)
S = stack()
```

Output:

```
>>> s.l
    [0, 0, 0, 0, 0]
>>> s.push(10)
>>> s.push(20)
>>> s.l
    [10, 20, 0, 0, 0]
>>> s.peek()
    data = 20
>>> s.pop()
    data = 20
>>> s.l
    [10, 0, 0, 0, 0]
```

Step 4,8 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: Else, print Statement as insert the element into the stack & initialize the value.

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

Step 7: If in pop method, value is 1, then return that 'stack is empty' or else delete the element from the stack at topmost position.

Step 8: First, check whether the no. of element are zero while the second case whether tos is assigned any where. If tos is not assigned any value then it can be sure that stack is empty.

Step 9: Assign the element value in push method to print the given value is popped or not.

Step 10: Attach the input and output of above algorithm

Practical VI

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 ~~of queue~~ is inserted after rear and element of a 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():** creates a new empty queue
- **Enqueue():** Inserts an element at the rear of the queue and similar to that of insertion of linked using tail.
- **Dequeue():** Returns the element at the ~~rear of the queue~~ front. The front is moved to the successive element. A dequeue operation cannot remove element 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 enqueue(self, data)
        n = len(self.l)
        if self.r < n:
            self.l[self.r] = data
            self.r += 1
            print("element inserted :", data)
        else :
            print("Queue is full")
    def dequeue(self):
        n = (self.l)
        if self.f < n :
            print(self.l[self.f])
            self.l[self.f] = 0
            print("Element deleted ")
            self.f += 1
        else :
            print("Queue is empty")
q = queue()
```

Output:

```
>>> q.enqueue(10)
    element inserted : 10
>>> q.enqueue(20)
    element inserted : 20
>>> q.enqueue(30)
    element inserted : 30
>>> q.enqueue(40)
    queue is full
>>> q.dequeue
    element deleted.
```

Algorithm:

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

Step 2: Define an empty list and define enQueue() method with 2 argument. Assign the length of empty list.

Step 3: Use if statement that length is equal to rear then Queue is full, 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 & 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.

Practical VII

Topic: Evaluation of a Postfix Expression

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

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

Algorithm:

Step 1: Define evaluate as function then create an 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.

Coding:

```
def evaluate(s):
    k = s.split()
    n = len(k)
    stack = []
    for i in range(n):
        if k[i].isdigit():
            stack.append(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 = str(input("Enter string :"))
r = evaluate(s)
print("The Evaluated Value =", r)
```

Output :

Enter string 8 6 9 * +

The Evaluated Value = 62.

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 & input of above algorithm.

20/10/20

Practical 8

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

Theory: A linked list is a linear data structure which stores the elements in a node in a linear fashion but not necessarily contiguous. The individual element of the linked list called ~~Data~~ node. Node comprises of 2 parts i) Data ii) Next. Data stores all the information with respect to the element. for eg. roll no, name, address etc. Where 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 a very tedious task so linked list is used to solve this type of problems.

Algorithm:

Step 1: Transversing of a linked list means ~~visting~~ visiting all the nodes in the linked list in order to perform some operation on them.

Step 2: The entire linked list can be accessed with the first node of the linked list. The first node of the linked list in turn is refered by the head pointer of the linked list.

Coding :-

```
# linked list
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 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.
Start = linked list ().
```

Output :-

```
>Start.addL(50)
>Start.addL(60)
>Start.addL(70)
>Start.addL(80)
>Start.addB(40)
>Start.addB(30)
>Start.addB(20)
>Start.display().
>print("Fahad")
>>>
  20
  30
  40
  50
  60
  70
  80
 Fahad
>>>
```

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

Step 5: We should not use the head pointer to transverse the entire linked list because the head pointer is our only reference to the 1st node in the linked list. Modifying the reference of the head pointer can lead to changes which we cannot revert back.

Step 6: We may lose the reference to the 1st node in our linked list. So, in order to avoid making some unwanted changes to the 1st node, we will use a temperory node to transverse the entire linked list.

Step 7: We will use this temperory node as a copy of the node we are currently transversing. Since we are making temporory node a copy of current node, the datatype of the temperory node should also be node.

Step 8: Now that current is referring to the ~~fist~~ first node, if we want to accept 2nd node of list, we can refer it as the next node of the 1st node.

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

Step 10: Similarly, we can transverse rest of nodes in the linked list using some method by while loop.

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

Step 12: The last node is the linked list is referred by the tail of linked list. Since the last of node of linked list doesnot have any next node, the value in the next field of the last node is name.

Step 13: So we can refer the last node of linked list as self.s = name.

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

Step 15: Attach the coding / input and output of the above algorithm.

Practical 9

Aim: To sort a list using merge sort.

Theory: Like Quick sort, merge sort 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 a key process that assumes that arr[m+1...r] are sorted and merges the two sorted sub arrays into one. The array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

Applications:

i) Merge Sort is useful for sorting linked lists in O(n log n) tion. Merge sort accesses data sequentially and the need of random access is low.

ii) Inversion Count Problem

iii) Used in external sorting.

Merge sort is more efficient than quick sort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular where sequentially accessed data structure are very common.

Algorithm:

Step 1: Define the sort (arr, l, m, r):

Step 2: Store the starting position of both parts in temporary variables.

Step 3: Check if first part comes to an end or not.

Step 4: Check if second part comes to an end or not.

Step 5: Check which part has smaller element.

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

Step 7: Define the correct array in 2 parts.

Step 8: Sort the first part of array.

Step 9: Sort the second part of array.

Step 10: Merge the both parts by comparing elements of both the parts.

coding :-

```
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 += l
        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 .
```

P.T.O →

```
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, 56, 42, 45, 78, 86, 98].
```

Practical No 10.

Aim: Implementation of sets in python.

Algorithm:

Step 1: Define two empty set as set1 and set2. Now use for statement providing the range of above 2 sets.

Step 2: Now ~~add~~ use add() method used for adding the element according to given range, then print the sets after adding.

Step 3: Find the union and intersection of above 2 sets by using & (and), | (or) method. print the sets of union and intersection as set 3 & set 4.

Step 4: Use if statement to find out the subset and super set of set 3 and set 4. Display the above set.

Step 5: Display that element in set 3 is not in set4 using mathematical operation.

Coding:

```
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")
    print("\n")
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\n")
set5.clear()
print("After applying clear, set5 is empty set")
print("set5 =", set5).
```

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 set 1 and set 2 : set 3 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}.

Intersection of set 1 and set 2 : set 4 {8, 9, 10, 11}.

Set 3 is superset of set 4.

Set 4 is subset of set 3.

Elements in set 3 and not in set 4 : set 5 {1, 2, 3, 4, 5, 6, 7, 12, 13, 14}.

Set 4 and set 5 are mutually exclusive.

After applying clear, set 5 is empty set

set 5 = set ().

03/04/2022

Step 6: Use isdisjoint() to check that anything is common element is present or not. If not, then display that it is Mutually Exclusive event.

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

Practical No. 11

Aim: Program based on binary search tree by implementating inorder, preorder and post order 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 inturn might have left and right subtres.

ii) Visit the root node.

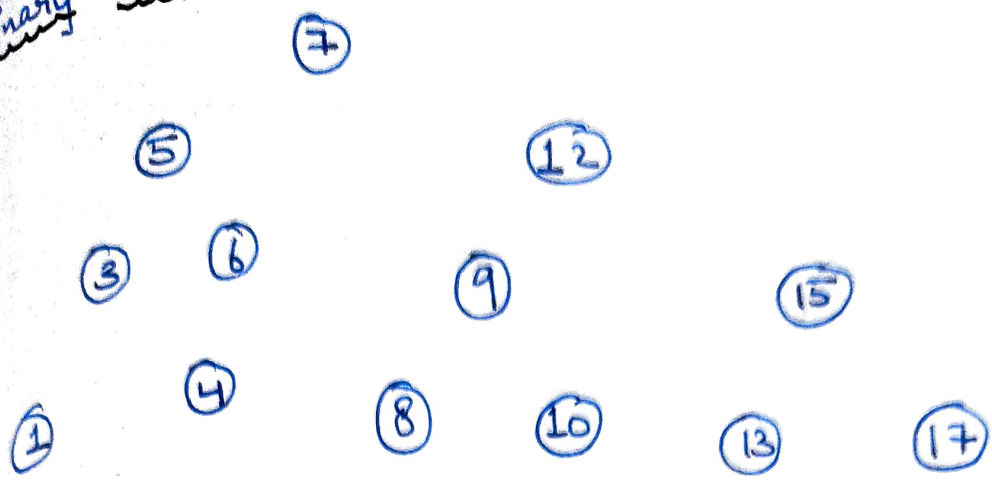
iii) Transverse the right subtree and repeat it.

→ **Preorder:** i) Visit the root node.

ii) Traverse the left subtree. The left subtree inturn might have left and right subtree.

iii) Traverse the right subtree. repeat it

Binary Search Tree.



Coding :-

```
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 = Node(val)
            h = self.root
            while True :
                if newnode.data < h.data :
                    if h.l != None :
                        h = h.l
                    else :
                        h.l = newnode.
```

P.T.O →

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

T = Tree()
T.add(100)
T.add(180)
T.add(70)
T.add(85)
```

P.T.O →

Postorder : i) Traverse the left subtree. The left subtree inturn might have left & right subtree.
ii) Traverse the right subtree.
iii) Visit the root node.

Algorithm :

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

Step 2 : Again define a class BST that is Binary Search tree ~~and assign the~~ 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 then use else statement for if node is less than the main node then put or ~~organi~~ arrange that in left side.

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

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

Step 7: After this, left sub tree and right subtree, repeat this method to arrange the node according to Binary Search Tree.

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

Step 9: In Inorder(), else statement used for giving that condition first left, root, then right node.

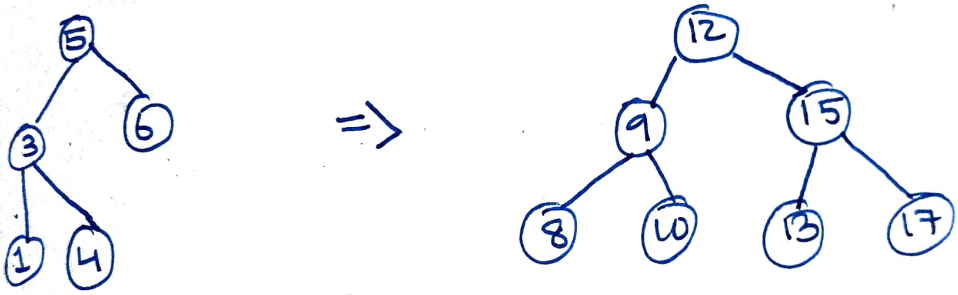
Step 10: For Preorder(), we have to give condition in else that first root, left and then right node.

Step 11: For Postorder(), In else part, assign left then right and then go for root node.

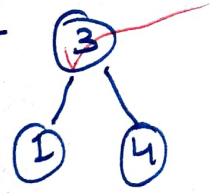
Step 12: Display the output & input of above algorithm.

```
T.add(10)
T.add(78)
T.add(60)
T.add(88)
T.add(15)
T.add(12)
print("preorder")
T.preorder(T.root)
print("inorder")
T.inorder(T.root)
```

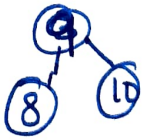
Binary Search Tree (In Order) (WR)



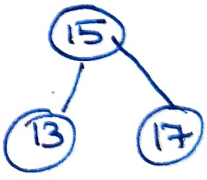
Step 2:-



5 6 7



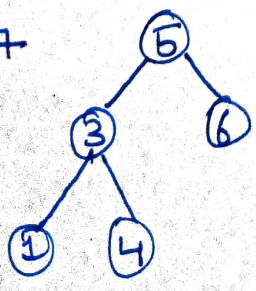
12

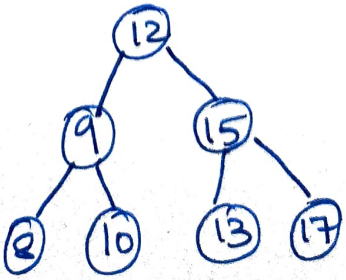


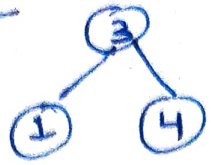
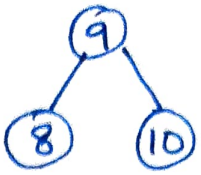
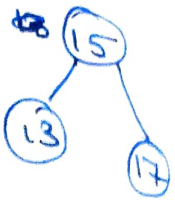
Step 5:- 1 3 4 5 6 7 8 9 10 12 13 15 17

post order (VLR)

Step 1 : 7

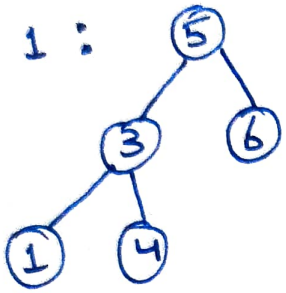
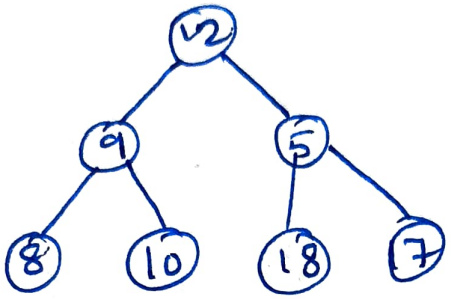


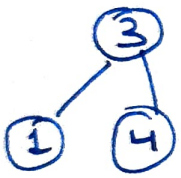
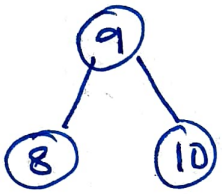
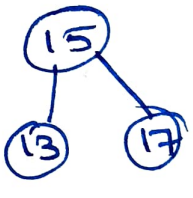


Step 2 : 7 5  6 12  

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

postorder : (LRV)

Step 1 :  

step 2 :  6 5   12 17.

Step 3 : 1 4 3 6 5 8 10 9 13 17 15 1 2 7.

Output :-

80 added on left of 100
70 added on left of 80
85 added on right of 70
10 added on left of 70
78 added on right of 70
60 added on right of 10
88 added on right of 85
15 added on left of 60
12 added on left of 15

Preorder	Postorder	Inorder
100	10	10
80	12	12
70	15	15
10	60	60
60	70	70
15	78	78
12	80	80
78	85	85
85	88	88
88	100	100.