```
a = [ ]
n = int (input ("Enter an integer : "))
print ("enter the element of the list : "))
for i in range (n):
    x = int (input ())
    a. append (x)
t = int (input ("enter an element : "))
for i in range (n):
    if a [i] == t :
        print ("found at position", i+1)
        Break
If t is not in a :
    print ("no. not in list")
a = sorted (a)
for i in range (n):
    if a [i] == t :
        print ("for sorted list found": i+1)
        Break.
```

output : Enter an integer : 7
enter the element on the list :

1
4
3
5
7
8
9

found at element position = 5

enter an element = 7

Practical No. 1. Linear Search

Aim: To search for a particular element using the linear search algorithms.

step 1 : Declare the empty list use the while condition statement to run an infinite loop that input value, for the list use if condition statement.

step 2 : Take the element to be searched on an input use the for conditional statement to the list print the list and exit.

step 3 : If the element is not in list print it is not in list use the sort () method and repeat the for conditional statement from step 2.

step 4 : End the program.

Practical NO. 2. Binary search.

Aim: To search an element of array using binary search algorithm.

step 1: Declare an empty list use the while loop to input value for the list, which are appended to the list, which are appended to the list and break using the 'break' statement.

step 2: Take the element to be searched as input print the sorted list (sorted ())
Define a variable also define x (ubt/lb)/2

step 3: Use the while condition statement to run an infinite loop use the if conditional statement to check whether element of present at 'n' and terminate the loop elif if greater than element put lb = x+1, x (ub + lb/2) elif the element in present in the list print that the element is not present & break the loop.

step 4: Terminate the program.

```
a = [ ]
print ("enter 'n' to quit")
print ("enter the element of the array")
while (True):
    n = input ()
    if (n.casefold () == 'n'):
        Break
    else :
        a.append (int (n))
a = sorted (a)
print ("The searched that is", a)
y = int (input ("Enter the element to the searched")
lb = 0
ub = len (a)
n = (ub + lb)/2
while (True)
    if (y == a [int (x)])
        print ("the number "a" was found at
                position {}. format . int (x+1))
        Break
    elif (y > = a [int (x)])
        lb = x+1
        n = (ub + lb)/2
    elif y not in a :
        print ("The no was not found")
    Break.
```

032

output:

```
Enter 'n' to quit
Enter the element of the array
21
23
53
12
26
76
n
The sorted list is [12, 21, 23, 26, 53, 76]
Enter the element to be searched : 21
The no. '21' was found at position 2.
```

Practical No. 3. Bubble sort.

Aim: To sort a list in a range order using bubble sort algorithm.

step 1: Define a empty list at the while conditional loop

step 2: Print the unsorted list using the for conditional statement i in range len (list) as iterable that from 0 to len (list) i-1

step 3: Use the if conditional statement to check greater than element the value if true.

step 4: Terminate the program.

```
a = [ ]
print ("enter the integer value for the list and 'n' to stop entering:")
while (True):
    x = input ()
    if (x.casefold (x) == "n"):
        Break
    else:
        a.append (int (x))
    print ("the unsorted list is", a)
    for i in range (len (x) - 1 (1):
        for i in range (len (a) - 1 - i):
            if (a [j] == a [j] + 1):
                a [j] = a [j+1] = a [j+1], a [j]
    print ("the sorted list \n", a)
```

Output:

```
enter the integer value for list & type
5
12
82
123
100
91
69
```

The unsorted [5, 12, 22, 123, 100, 99, 69]

The sorted [5, 12, 822, 69, 99, 100, 123]

```
class stack:
    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("The stack is full")
        else:
            self.tos += 1
            self.l[self.tos] = data
    def pop(self):
        if self.tos < 0:
            print("The stack is empty")
            self.l[self.tos] = 0
            self.tos -= 1
    x = stock()
```

output:

```
x.push(10)
x.push(7)
x.push(44)
x.push(80)
x.push(89)
x.push(8)
  The stock is full
```

```
x.pop()
89
x.l
[10, 7, 44, 80, 0]
```


Practical No. 4. Stack.

Aim: Implementation of stack using python list

Theory: A stack is linear data structure that can be represented in a real world form by physical stack or pile. The elements in the form stack are the position. Thus it work in the LIFO principle (Last in first out). It has 3 basic operations namely push, pop, peek.

Algorithm:

Step1: Create a stack with instance variables item.

Step2: Define the init() method with the self and initialise the initial value and then initialise to an empty list.

Step3: Define method push and pop under the class stack.

Step4: Use if statement to give the condition that if length of given list is greater than the range of list than print stack is full.

step5 : Else state to print a statement as input the element into the stack and initialise the value.

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

step7 : If in pop method, value is less than 1, then return is empty, or else delete the element from stack at top most position.

step 8 : Assign the element value, in push method and print the given value.

step 9 : Attack the input and output of above Algorithm.

step 10 : First condition checks whether the no. of elements are zero while the second case whether top is assigned any value. If top is not assigned any value then print that the stack is empty.

29/01/20

code.

```
print (Quick sort)
def partition (arr, low, high):
    i = low-1
    pivot = arr [high]
    for j in range (low, high):
        if arr [j] <= pivot:
            i = i+1
            arr [i], arr [j] = arr [j], arr [i]

    arr [i+1], arr [high] = arr [high], arr [i+1]

    return i+1
def quicksort (arr, low, high):
    if low < high
        pi = partition (arr, low, high)
        quicksort (arr, low, pi-1)
        quicksort (arr, pi+1, high)
a1 = input ("enter elements in the list). split()
alist1 = []
print = ("elements in the list are: ", alist1)
n = len (alist1)
quicksort (alist, 0, n-1)
print ("elements after quicksort is", alist1)
```


Practical No. 5.

Aim: Implement Quick sort to sort the given list.

Theory: The quicksort is a recurrence algorithm thus based on the divide and technique.

Algorithm:

step 1: Quick sort first selects a value, which is called pivot value first 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 and at the same time move other items to the appropriate side of the list either less than or greater than pivot value.

step 03: Partioning begins by locating two position marked lets call them eliminate and right mark at the beginning and end off.

remaining items in the list. The goal of partition process is to move items in the list.

step 4: we begin by increasing leftmark until we locate a value that is greater than the pv we then decrement rightmark until we find value that is less than the pv.

step 5: At the point where right mark becomes less than leftmark we stop

step 6: The pv can be exchanged with the content of split point and pv is now in place.

step 7: Display & stick the coding & output of above algorithm.

output:

Quicksort:

038

enter elements in list 21 20 22 30 43 41

elements after quick sort are

20 21 22 30 41 43

code:

```
class Queue:
    global r
    global y
    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(self.l)
            print(self.l[self.f]
            self.f = self.f+1
        else:
            print("Queue is empty")
q = Queue()
```


Practical no. 6.

Implementing a Queue using python list.

Theory: Queue is a linear data structure which has 2 references front & rear implementing a queue using python.

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:

Step1: Define a class queue and assign global variables then define init () method with self argument.

Step2: Define a empty list and define enqueue() method with 2 argument assign the length of empty list.

Step 03: Use the if statement that length is equal to rear then queue is full or else insert the element in empty list.

Step 04: Define dequeue () with self argument use if statement that front is equal to length of list then display Queue is empty or else give that is at 0.

Step 05: Now all the Queue () function & give the element that has to be added in the empty list by using enqueue ()

output:

```
>>> Q.add (30)
    Q.add (40)
    Q.add (50)
    Q.add (60)
    Q.add (70)
    Q.add (80)
    Q.add (90)
Queue is full
Q remove ()
 30
Q remove ()
 40
Q remove ()
 50
Q remove ()
 60
Q remove ()
 70
Q remove ()
 80
Q remove ()
Queue is empty.
```

```
# 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))
```

Practical no. 7

Program on Evaluation of given string by using stack in python environment i.e postfix.

Theory: The postfix expression is free of any paranthesis. Further we take care of the priorities of the operators in the program.

Algo:

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 & print it.

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

Step 5: perform arithmetic operation on push the result back on the 'm'

Step 6: print the result of the string after the evaluation of postfix.


```
else:
    a = stack.pop()
    b = stack.pop()
    stack.append(int(b) / int(a))
return stack.pop()
s = "869*+"
r = evaluate(s)
print("The evaluated value is:", r)
```

Output:

The evaluated value is: 62

13/02/2020

```
class node:
    global data
    global data
    def __init__(self, item):
        self.data = item
        self.next = name

class linkedlist:
    global s
    def __init__(self):
        self.s = none
    def addL(self, item):
        newnode = node(item)
        if self.s == node:
            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:
```


Practical no. 8.

Implementing of adding of single linked list by nodes from last position.

Theory: A linked list is a linear data structure which is storing the element in a note in a linear fashion but no necessary continue the information

Algo:

step 1: Traversing of a linked list mean position all the node in the linked list in order to perform some operation.

step 2: The entire linked list mean can be accessed as the first node of the linked list.

step 3: Thus the entire linked list can be traversed using node which is refered by the pointed of linked list.

step 04: Now that we know that we can traverse the entire linked list using the head pointer to refer the first node of list only.

step 05: We should not use the head pointer to traverse the entire linked list because the head pointer is our reference to node 1st.

step 06: Temporary node as a copy of the node we are currently traversing.

step 07: Now the current is reference to the first node if we want to access 2nd node of the list. we need to refer it a node of 1st node.

```
        self.s = newnode
    else:
        newnode.next = self.s
def display(self):
    head = self.s
    while head.next != none:
        print(head.data)
start = Linked list()
```


output:

```
start.add(80)
start.add(70)
start.add(60)
start.add(50)
start.add(40)
start.display()
```

```
>>> 40
    50
    60
    70
    80
```

```
def sort (arr, l, m, r)
    n1 = m1 - l+1
    n2 = r-m
    L = [0] * (n1)
    R = [0] * (n2)
for i in range (0, n1)
    L[i] = arr [i+1]
for j in range (0, n2)
    R[i] = arr [m+i+j]

i=0
j=0
k=l
while i<n1 and j<n2:
    if L[i] <= R[i]
        arr[k] = L[i]
    else:
        arr[k] - R[i]
        j = j+1
        k = k+1
        i = k+1
```


Practical NO.9

Aim: Implementation of merge sort by using python.

Theory: Merge sort is a divide and conquer algorithm. It divide input array into two halve and then merge() function is used for two values.

Algorithm:

Step 1: The list is divided into left and right in push recursive call until two adjacent or obtain.

Step 2: Now begin the sorting process then use if iteration transit the two halves value in each than iteration transverse the whole value & list & values change alongside.

Step 3: Do the value is smaller than the value at j(i) is sorting to the arr [0+1] the increasing it first R[j] is sorting.

Step 4: This way the value being assigned through h [i+1] are all sorted.

step 05: At end of the loop one of the halve will not have been transversed because completely reverse slot in the list.

step 6: Thus the merging sort has been completed.


```
        arr[k] = L[i]
        i = i+1
        k = k+1
    while i < n2
        arr[k] = R[i]
        j += 1
        k = 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)
merge sort(arr, 0, n-1)
print(arr)
```

output :

[12, 23, 84, 56, 78, 45, 86, 98, 42]

[12, 23, 36, 56, 42, 45, 78, 86, 98]

```
set1 = set()
set2 = set()
for i in range (8, 15)
    set1.add(i)
print ("set1: ", set1)
print ("set2: ", set2)
print ("\n")
set3 = set1 / set2
print ("union of set1 and set2: set3", set3)
print ("\n")
if set3 > set4:
    print ("set is superset of set4")
elif
    set3 < set4:
    print ("set3 is subset of set4")
else:
    print ("set3 is same set4")
```


Practical NO. 10.

Aim: Implementation of set using python 3.4.3

Algorithm:

Step 1: Define two empty set at 1 and now user state provide then range of above 2 sets.

Step 2: Now add() method is used for addition of two element according to give range the print it sets for addition.

Step 3: find the union and intersection of above 2 set by using 1 method print the set by given & intersection of 3.

Step 4: Use if statement to find out the subset and superset of at 3 and 4 display the set.

Step 5: Display that element in at 3 is use its string using mathematical operation.

Step 6: Use discard() to remove and delete the sets and print the sets after clearing them element present in the set.

```
if set 4 < set 3
    print ("set 4 is subset of set 3") 048
    print ("\n")
set 5. set 3. set 4
    print ("element in set 3 and list in set 4
            set 5 , set)
    print ("\n")
    if set 4 is disjoint (set 5):
        print ("set 4 and set 5 are comtaisly
                exclusive")
    set clisca()
    print ("After applying exclusive \n")
    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 set 1 and set 2: set 3

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}

intersection of of set1 and set2

{8, 9, 10, 11} set 3 is superset

after applying clear set 5 = set()

Practical NO. 11

Program based on library binary search tree by implementation, inorder, preorder, & postorder traversal.

Theory: Binary tree is a tree which support maximum of 2 children for any node within the tree. Thus any particular node on either have 0 or 1 or 2 children. There is another identify of binary tree if is ordered such that one child is identified as left child and other as right child.

Inorder: Traverse the left structure. The left subtree information might have left and right substance.

Preorder: Visit the root node traverse the left subtree & right subtree.

Post order: Traverse the left subtree, the left subtree in return might have left & right subtree.

Algorithm:

1. Define the class Node and init () which has 2 arguments initialization the value in this method.
2. Again define a class BST that is Binary search tree with init () with self argument assign to the root is None.
3. Define add () for adding the nod, define a variable 'p' that p-Node (value)
4. Use if statement for checking the condition that node is none then use else statement to if node is less than main ~~root~~ node ~~node~~ then put on argument that is left side.
5. Use while loop for checking the node is less than & greater than the main node & break the loop if it is not satisfying.
6. Use if statement with that else statement defining that node is greater than main root than put into the right side.

code:

```
class node
    def __init__ (self, value):
        self.left = None
        self.val = value
        self.right = None
class BST:
    def __init__ (self):
        self.root = None
    def add (self, value)
        p = node (self)
        if self.root == None
            self.root = P
            print ("Root is added successfully
                P.value")
        else:
            h = self.root
            if p.val < n.val
                if h.val = None
                    h.left = P
                    print (Pval, "Node is added succes
                    break
                else:
                    h = h.left
            else: if h.right = None
                h.right = P
```

```
        if h.right == None
            print (pval, "Node is added successfully
                        to the right node")

        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)
        print (root, left)
        print (root, right)
def Postorder (root):
        if root == None
            return
        else:
            print (root, val)
            print (root, left)
            print (root, left)
t = BST ()
```


7. After this left side then right then repeat this method to arrange the node according to binary search tree.

8. Define Inorder(), Preorder() Postorder() with root as argument & use if statement that root is return that all.

9. Inorder use condition else statement is giving that node first left node then right

10. for preorder you have to give condition in else that root left & the right node.

11. for postorder in else part assign left and right root, so display input and output.

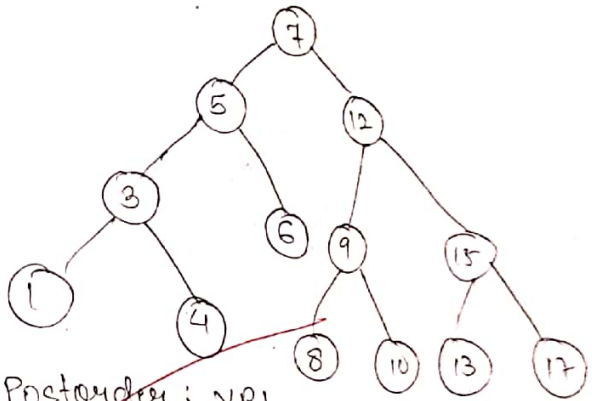
~~try~~

Output :-

```
>> t.add(1)
   root is added
>> t.add(2)
2 node is added to left side
>> t.add(4)
4 node is added to right side.
>> t.add(5)
5 node is added to right side
>> t.add(3)
3 node is added to right side.
   print("\n Inorder :", Inorder(t.root))
   Inorder :
1
2
3
4
5
Inorder: Now.
print("\n Preorder :", Preorder(t.root))
preorder
1   4   5
2   3
```

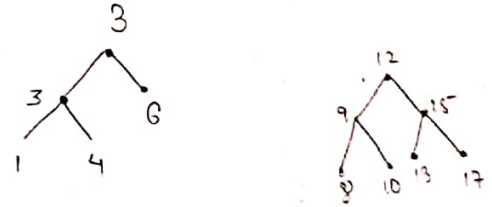
```
print("Postorder:", Postorder(t.root))
3
5
4
2
1
postorder : None.
```


* Binary Tree.

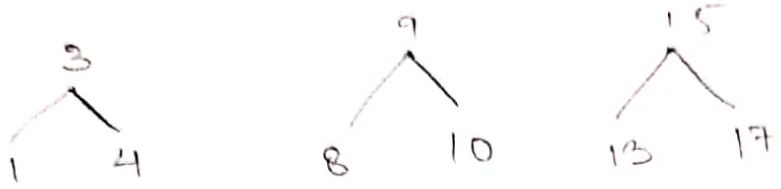


Postorder : VRL

1.

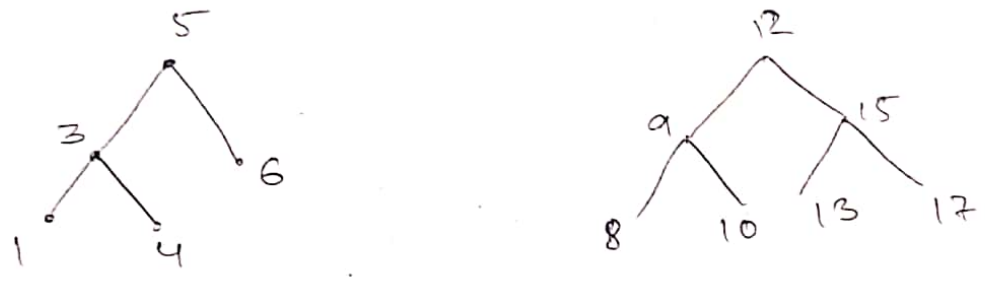


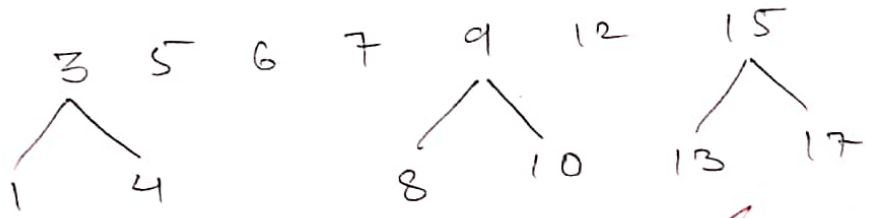
2. 3 6 5 9 15 12 7

2. 

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

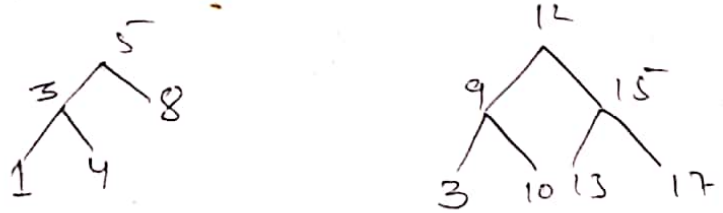
Inorder (LVR)

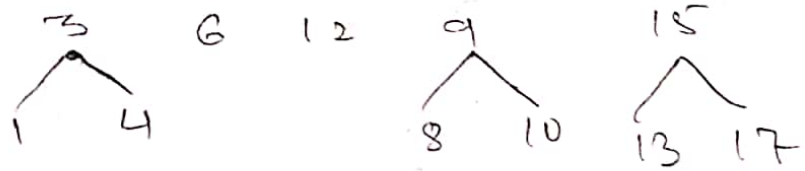
1. 

2. 

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

Preorder (VLR)

1. 

2. 

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