Practical 01:- Linear Search

Aim:- To search for a particular element using the linear search algorithm

Step 01:- Declare the empty list use the while condition statement to run an infinite ~~loop~~ loop that input value, for the list use the if condition statement to check.

Step 2:- Take the element to be searched as an input, use 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


```
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 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
```

```
Enter an integer: 5
Enter the element on the list:
1
4
3
5
7
Enter an element: 5
Found at position: 4
```

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


Practical 02:- 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 (list lb) /ub

Step 3: Use the while conditional statement to run an infinite loop, then use the if conditional statement to check whether element at present at m and terminals the loop elif if greater than m^{th} 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.

Output:

```
Enter the element of the array
21
23
53
52
26
76
n
```

The sorted liste is [12, 21, 23, 26, 53, 76]

Enter the element to be searched: 21

The no. '21' is found at position 2.

```
a=[]
print("Enter the integer value for the list and press '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(a)-1):
    for j in range(len(a)-i-1):
        if(a[j]>=a[j+1]):
            a[j],a[j+1]=a[j+1],a[j]
print("The sorted list is:",a)
```

Output:

```
Enter the integer value for the list and press 'n' to stop
entering:
5
12
32
123
100
99
69
n
The unsorted list is:[5,12,32,123,100,99,69]
The sorted list is:[5,12,32,69,99,100,123]
```

Practical 03: Bubble Sort

Aim:- To sort a list in average, enter using bubble sort algorithm.

Step 01:- Define an empty list and use the while conditional loop to get input as many values as the list

Step 02:- Print the unsorted list, using the for conditional statement using 'i' in the range len (list) as iterable that from 0 to len(list) - i - 1.

Step 03:- Use if conditional statement to check whether question for sorted elements, accordingly sort the values if true.

Step 04:- Terminate the program.

In order now

Practical 04: 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 topmost position. Thus it works in the LiFo principle (Last in first out) It has 3 basic operations namely: push, pop, peek
- **Algorithm:-**

Step I: Create a class stack with instance variable items

Step II: Define the init method with self argument and initialize the initial value and then initialise the initial value and then initalize to an empty list.

Step III: Define methods push and pop under the class stack

Step IV: 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 V: Else statement to print a statement as insert the element into the stack and initialize the value.

```
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 = stack()
```

Output:-

```
>>> x.push(10)
>>> x.push(7)
>>> x.push(8)
>>> x.push(79)
>>> x.push(72)
>>> x.push(69)
The stack is full
>>> x.pop()
72
>>> x.l
[10, 7, 8, 79, 0]
```

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

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

Step VIII: Assign the element values, in push method to and print the given value

Step IX: Attach the input and output of above algorith.

Step X: 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.

09/01/22

Practical 05: Quick sort

Aim: Implement quick sort to sort the given list

Theory: The quick sort is a recursive algorithm, so it is based on the divide and sort technique.

Algorithm:

Step I: 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 II: The partition process will happen next. It will find the split point and at the same time move other items to the appropiate side of the list, then organize the pivot value.

Step III: Partitioning begins by locating two position marked lets call them eliminate and right mark. at the beginning and end of remaining items in the list are Goal of the position is to move items in the list

Step IV: We begin by increasing ~~remain~~ leftmark until we locate a value that is greater than pivot value

#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, pi+1, high)
x = input("Enter elements in the list:").split()
alist = []
print("Elements in list are:", alist)
n = len(alist)
quicksort(alist, 0, n-1)
print("Elements after quick sort is:", alist)
```

Output:

```
Quick Sort
Enter elements in the list: 21 20 22 30
Elements after quick sort is: 20 21 22 30
```

Step V: At the pivot where rightmark becomes less than leftmark or top.

Step VI: The pv can be expanded with the content of split point and pv is now in place

Step VII: In addition all the items to left the split point are the less than pv

Step VIII: The quicksort function involves a recursive function to sort

Step IX: Quicksort helps begins with some base or merge sort

Step X: Display & stick the contrary & output of above algorithm

Practical 6: Queue

Title: Implementing a queue using Python list

Theory: Queue is a linear data structure which has 2 references front and rear. It is based on the principle that a new element is inserted at rear after a rear element in the queue is deleted which is at front in simple terms a queue can be described as a data structure or first in - first out.

Queue(): Creates a new empty queue

Enqueue(): Insert a element at the rear of the queue

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

Algorithm:

Step I: Define a class queue and assign a global variable, then define init() method with self argument to initialize the initial value with help of self argument

#Code: Queue

```
class node:
    global data r
    global next f
    def __init__(self, item):
        self.data = item 0
        self.f = 0
        self.next = None 0
        self.l = [0, 0, 0, 0, 0, 0]
class linkedlist
    def add (self, data):
        n = len (self.l)
        if self.r < n - 1:
            self.l [self.r] = data
            self.r = [self.r + 1]
        else:
            print ("Queue is full")
    def remove (self):
        n = len (self.l)
        if self.f < n - 1:
            print (self.l [self.f])
            self.f = self.f + 1
        else:
            print ("Queue is empty")
q = Queue()
```

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
```


Step II: Define an empty list and define enqueue() method with 2 argument assigned, the length of empty list.

Step III: Use if statement to check whether the length is equal to rear then print queue is full or else insert the element in empty list and increment the counter variable by 1.

Step IV: Define Dequeue() with self argument, then use if statement that will check front is equal to length of list other display queue is empty or else give that front is at 0 & using that delete the element from front side & increment by 1.

Step V: Now all the Queue() function & give the element that has to be added in the empty list using enqueue() & print list after ~~after~~ adding & same for deleting.

Practical 7

Aim: Implementation of singly linked list by adding the node from last position

Theory: A linkedlist is a linear data structure which is storing elements in a node in a linear manner. The individual element of linked list is called as node and it comprises of two parts:
① Data ② Next
Data stores all information whereas next refers to the next node

→ Algorithm:

Step I: Traversing of a linked list means positioning all the nodes in the linked list in order to perform some operation on them.

Step II: The entire linkedlist means can be accessed as the first node of the linked list.

Step III: Thus, the entire linkedlist can be traversed using node which is referred by the head pointer of linked list

Step IV: Now we know that we can traverse the entire linkedlist using the head pointer we should only use it to refer the first node of list only.

#Code:

```
class node:
    global data
    global next
    def __init__(self, item):
        self.data = item
        self.next = None
class linked list:
    global s
    def __init__(self):
        self.s = None
    def addl(self, item):
        newnode = node(item)
        if self.s == None:
            self.s = newnode
        else:
            head = self.s
            while head.next != None:
                head = head.next
            head.next = newnode
    def addB(self, item)
        newnode = node(item)
        if self.s == None:
            self.s = newnode
        else:
            newnode.next = self.s
            self.s = newnode
```

```
    def display(self):
        head = self.s
        while head.next != None:
            print(head.data)
            head = head.next
        print(head.data)
start = LinkedList()
```

Output:

```
>> start.add1(80)
>> start.add1(70)
>> start.add1(60)
>> start.add1(50)
>> start.add1(40)
>> start.add1(30)
>> start.add1(20)
>> start.display()
>>> 20
    30
    40
    50
    60
    70
    80
```


Step VI: We should not use the head pointer to traverse the entire linkedlist because ~~the head~~ it is our only reference to 1st node

Step VI: We may use the reference to 1st node in the linkedlist & hence most of our linked list so in order to avoid making some unwanted changes to the 1st node we'll use temporary node to terminate the entire linked list.

Step VII: 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 should also be considered as a node.

Step VIII: Now that ~~or~~ referring to the first node if we want do access to 2nd node of list, we need to refer it to as next node of 1st node; we can traverse to 2nd node as n = n.next

Step IX: Similarly we can traverse rest of nodes in the linked list, using while loop.

Step X: The last node of linked list is referred as tail node and we can ~~start~~ start traversing from first to last node with help of head pointer.

Practical 8

Aim: Program on Evaluation of given string using stack in Python environment i.e Postfix

Theory: The postfix expression is free of any parenthesis. Further we took care of priorities of operators in the program, it is easily evaluated using stack and it reading is done from left to right.

Algorithm:

Step I: Define evaluate as function then create an empty stack in Python.

Step II: Convert the string to a list by listing the string method 'split'.

Step III: Calculate the length of string & print it.

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

Step V: Scan the token list from left, if token is an operand convert it from a string to an integer & push the value onto 's'.

Code:


```
def evaluate(s):
    k = s.split()
    n = len(k)
    stack = []
    for i in range(n):
        if k[i].isdigit():
            stack.append(int(k[i]))
        elif k[i] == '+':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) + int(a))
        elif k[i] == '-':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) - int(a))
        elif k[i] == '*':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) * int(a))
        else:
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) / int(a))
    return stack.pop()
```

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

Output:

The evaluated value is : 62

(VI) If the token is an operand *, /, +, -, ^ it will need two operands. pop the 'p' twice. The first pop is second operand & the second pop is of the first operand.

(VII) Perform the arithmatic operations on. Push the result back on the 'm'.

(VIII) When the input expression has been completely processed the result is on the value.

(IX) Print the result of string after evaluation of postfix

(X) Attach output & input of above algorithm

Practical 9

→ **Aim:** Implementation of Mergesort using Python

→ **Theory:** Mergesort is a divide and conquer algorithm. It divides the array into two halves and then merge the two sorted halves. The merge function is used for merging two halves.

* Algorithm:-

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

Step II: Now begin the sorting process. The i & j of iterators I traverse the two halves in each call. Then k iterators traverse the whole list & makes the changes in position accordingly.

Step III: If the value at i is smaller then the value at j is assigned to the arr[i+1] slot and i is incremented. If not then R[j] is chosen

Step IV: In this way, the values being assigned through h[i+1] are all sorted

Step V: At the end of this loop, one of the halves is not ~~been~~ being traversed completely, keeping remaining slots in the list unchecked.

Code:-


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

```
        while j<n2:
            arr[k]=R[j]
            j+=1
            k=k+1
def mergesort (arr,l,r):
    if L<r:
        m=int ((L+(r-1))/2)
        mergesort (arr, L,m)
        mergesort (ar, m+1, r)
        sort (arr, L,m, r)
arr =[12, 23, 34, 56, 48, 45, 86, 98, 42]
print(arr)
n= len(arr)
mergesort (arr, 0, n-1)
print (arr)
```

#Output:

[12, 23, 34, 56, 48, 45, 86, 98, 42]

[12, 23, 56, 56, 42, 45, 48, 86, 98]

Step VI: After the sorting, the elements are placed in the particular order and thus the merge sort has been implemented.

Practical-10

* Aim:- Implementation of set using Python

* Algorithm:-

Step I:- Define two empty set as set 1 and set 2, now for statement providing the range of above 2 sets

Step II:- Now use add() method for addition of the elements according to given range then print the sets for addition.

Step III:- Find the union and intersection of above 2 sets by using / method, then print the sets of union & intersection of set 3.

Step IV:- Use if statement to find out the subset and superset of set 3 and set 4. Display the sets.

Step V:- Display that element in set 3 is not present in set 4 using mathematical operations.

#Code:-

```
set1=set()
set2=set()
for i in range(8,15):
    set1.add(i)
for j in range(1,12):
    set2.add(j)
print("Set 1:", set1)
print("Set 2:", set2)
print("\n")
set3=set1|set2
set4=set1&set2
print("Union of Set1 and Set2:", set3)
print("Intersection of Set1 and Set2:", set4)
print("\n")
if set3>set4:
    print("Set 3 is superset of set 4")
elif set3<set4:
    print("Set 3 is subset of Set 4")
else:
    print("Set 3 is same as Set 4")
if set4<set3:
    print("Set 4 is subset of Set 3")
    print("\n")
set5=set3-set4
```

```
print ("Elements in set 3 and are not in set 4:",
    set5^set4),
print ("\n")
if set4.isdisjoint(set5):
    print ("Set 4 and set 5 are mutually exclusive
        \n")

set.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 Set1 and Set2
{8, 9, 10, 11}

Set 3 is superset of Set 4

Elements in set 3 are 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()

Practical 11

→ **Aim:-** Program based on binary search tree by implementation Inorder, Preorder & Postorder transversal.

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

- **Inorder:-** Transverse the left subtree. The left subtree information might have left and right substance.
- **Pre-order:-** Visit the root node traverse the left subtree & right subtree.
- **Post-order:-** Transverse 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 in binary search tree with init() with self argument assign the root is None.

3) Define add() for adding the node, define a variable 'p' that p-node(value).

4) Use if statement for checking the condition that root is None then use else statement for if node is less than main nod then put an argument that is left side

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

6) Use if statement within that else statement declaring that node is greater than main root then put it into right side.


```
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(value)
        if self.root == None:
            self.root = p
            print("Root is added successfully",p.val)
        else:
            h = self.root
            if p.val < h.val:
                if h.val = None:
                    h.left = p
                    print(p.val, "Node is added
                        successfully")
                    break
                else:
                    h = h.left
            else:
                if h.right = None:
                    h.right = p
```

```
            print(p.val, "Node is added 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)
        preorder(root.left)
        preorder(root.right)
def Postorder(root):
    if root==None:
        return
    else:
        print(root.val)
        postorder(root.left)
        Postorder(root.left)
t=BST()
```


7) After this for leftside tree & right tree 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 ~~none~~ None & return that all.

9) Use Inorder, else statement for giving that condition first left ~~root~~ node & then right root

10) For Preorder ~~use~~ you have to give condition in else that first root left & the right node

11) For Postorder in else part assign left and right root, so display input & output

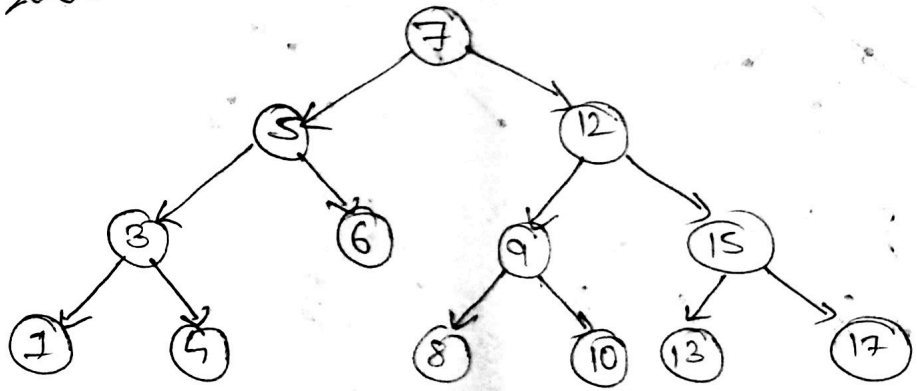
→ Output:-

```
>>> t.add(1)
root is added successfully
>>> t.add(2)
node is added to ~~right~~ left side.
>>> t.add(4)
node is added to right side
>>> t.add(5)
node is added to left side
>>> t.add(3)
node is added to right side

>>> print("\n Inorder: ", Inorder(t.root))
 Inorder:
1
2
3
4
5
Inorder: None

>>> print("\n Preorder: ", Preorder(t.root))
Preorder
1
2
4
3
5
```



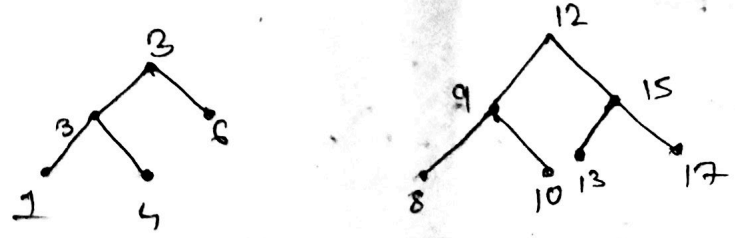
```
>>>print("Postorder: ", Postorder(t.root))
Postorder:
3
5
4
2
1
Postorder: None
```

* Binary Tree:-



Postorder (~~LVR~~ 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 ~~LVR~~)

1)

2)

2)

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

Preord (VLR ~~RLL~~)

(1)

2) 7 5 3 6 12 9 15

3) 7 5 3 1 4 6 12 9 8 10 15