PRACTICAL - 01

AIM: To search a number from the list using linear unsorted

Algorithm:

step 1: The data is entered in random manner

step 2: User needs to specify in the element to be searched in the entered list

step 3: Check the condition that whether the entered number matches, if it matches then display the location & increment by 1 as data is stored from location zero.

step 4: If all element are checked one by one & element not found then prompt message number not found.

THEORY

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

It is worth searching algorithm with worst case time complexity.

It is a basic 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 a technique to compare each is every element with the key element to be found, if both of them matches the algorithm return the element found, in its position is also found.

Sorted linear search:

Algorithm:

1) Assign a list in a variable taken from user and sort it.
2) Print the list and take input for number to be searched.
3) Use for conditional statment to iterate over the list and check whether the input variable is equal to element in a list.
4) Print the position if found or else print not found.

```
a = list(input("enter list"))
a.sort()
Print a
b = int(input("enter search no:"))
For i in range(len(a)):
    if b == a[i]:
        Print ("Found at:", i+1)
        break
else:
    Print("Not Found")
```

Output

Enter list : 4,9,2,1,3

[1,2,3,4,9]

Enter search no: 2

Found at 2.

Source code.

```
a = list(input("enter list:"))
a = sort()
c = len(a)
s = int(input("enter search no:"))
if (s > a[c-1] or s < a[0]):
    print("not in a list")
else:
    first, last = 0, c-1
    for i in range(0, c):
        m = int((first + last)/2)
        print("found at", m)
        if s == a[m]:
            print("number found")
            break
        else:
            if s < a[m]:
                last = m-1
            else:
                first = m+1
```

Output

enter list 8, 5, 9, 2, 1

[1, 2, 5, 8, 9]

Enter search no: 9

(found at, 2)

step 4: Use sort() method to sort the accepted element & assign it in increasing ordered list print after the list after sorting.

step 5: Use if loop to give the range in which element is found in given range after then display a message 'element not found'

step 6: Then use else statment, if statment is not found in range then satisfy the below condition

step 7: Accept an argument by key of the element that element has to be searched.

step 8: Initialise list to 0 and last to last element of the list as array is starting from 0. Hence it is initialised 1 less than the total count.

step 9: Use for loop & assign the given range

PRACTICAL 02

AIM: Implement binary search to find an searched no in the list

THEORY:

BINARY SEARCH

Binary search is also known as half interval search, logarithmic search or binary search is a search algorithm that finds the position of a target value with in a sorted array. If you are looking for the number which is at the end of the list then you need to search entire list in linear search which is time consuming. This can be avoided by using binary fashion search.

Algorithm:

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

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

Step 3: Use for loop, add elements in list using append method.

PRACTICAL 3

Bubble Sort

AIM: Implementation of bubble sort program on given list

THEORY: Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in 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

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

2) If the element is smaller than second then we do not swap the element

step 10 : If statment in list is still the element to be searched is not found then the middle element (m).

step 11 :- Else if the item to be searched still less than the middle term then initialize first (e) = mid (m) - 1

step 12 : Repeat till you found the el

Source code.

038

```
a= list(input("Enter the element :"))
for i in range(0,len(a)):
    for j in range(0,len(a)-1):
        if a[j]>a[j+1]:
            a[j],a[j+1]=a[j+1],a[j]
print(a)
```

Output :

4) Again 2nd and 3rd element are compared & swapped if it is necessary and this process go on until last and second last element is compared & swapped.

5) There are n element to be sorted then the process mentioned above should be repeated n-1 to get the required result.

6) Trick the output & input of above algorithm of bubble sort stepwise.

m

```
def quick(alist):
    help(alist, 0, len(alist)-1)
def help(alist, first, last):
    if first < last:
        split = part(alist, first, last)
        help(alist, first, split-1)
        help(alist, split+1, last)
def part(alist, first, last):
    pivot = (alist[first])
    l = first+1
    r = last
    done = false
    while done done:
        while l<r and alist[l]<=pivot:
            l=l+1
        while alist[r]>=pivot and l>=1;
            r=r-1.
        if r<1:
            done = True
        else:
            t = alist[l]
            alist[first] = alist[r]
            alist[r] = t
```

m

```
    t = alist[first]
    alist[first] = alist[r]
    alist[r] = t
    return r
x = list(input(" Enter range for list ~~the element~~: "))
alist=[]
for b in range(0,x):
    b = input("enter element")
    alist.append(b)
    n = len(alist)
```

PRACTICAL- 4

AIM: Implement Quick sort to sort the given list.

THEORY: The quick sort is a Recursive algorithm based on the divide & conquer technique.

Algorithm:

1) 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.

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

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 remaining

quick (alist)
print (alist)

Output

Enter range for list: 5
Enter element: 4
Enter element: 2
Enter element: 1
Enter element: 3
Enter element: 7

1, 2, 3, 4, 7

20/12/19

9) Quick sort helper begins with some base called as a merge sort.

10) If length of list is less than or equal to then, it is sorted.

11) If it is greater, then it can be partitioned and recursively sorted.

12) Display and stick the coding of output of above algorithm.

that are on wrong side with respect to pivot value while also with respect to pivot value while also converging on the split point.

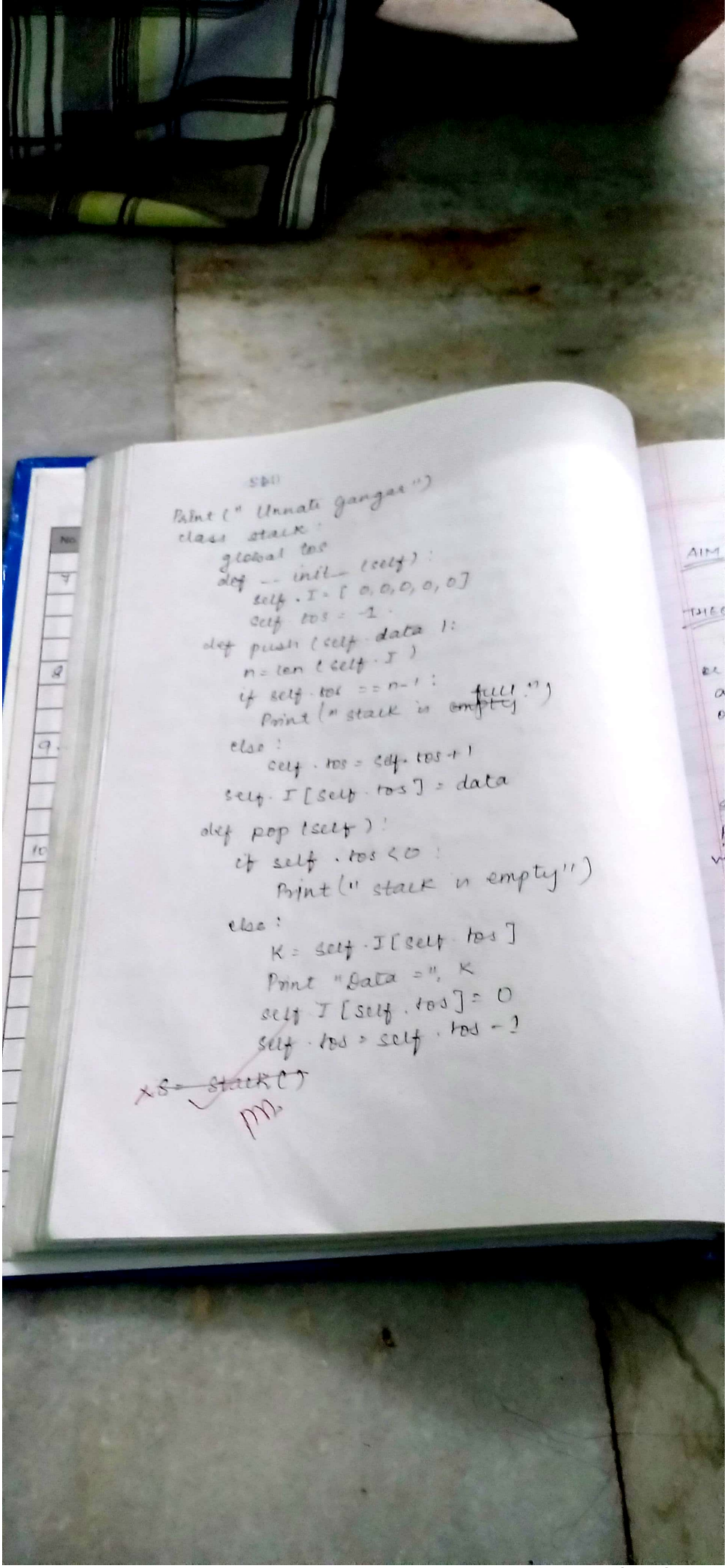
4) We begin by incrementing left mark until we locate a value that is greater than the P.V we then decrement right mark until we find value that is less.

5) At the point where right mark becomes less than left mark we stop. The position of right mark is now the split point.

6) The pivot value can be exchanged with the content of split point and P.V is now in place.

7) In addition all the items to left of split povit are less than PV and all the items to the right of split point. The list can now be divided at split point and quick sort can be invoked recursively on the two halves.

8) The quick sort function invokes a recursive func quick short help.



```
SDD
Print (" Unnati gangar")
class stack:
    global tos
    def __init__(self):
        self.I = [0,0,0,0,0]
        self.tos = -1
    def push(self, data):
        n = len(self.I)
        if self.tos == n-1:
            Print("stack is ~~empty~~ full")
        else:
            self.tos = self.tos + 1
        self.I[self.tos] = data
    def pop(self):
        if self.tos < 0:
            Print("stack is empty")
        else:
            K = self.I[self.tos]
            Print "data =", K
            self.I[self.tos] = 0
            self.tos = self.tos - 1
~~s = stack()~~
```

PRACTICAL NO : 05

AIM : Implementation of stacks using python list

THEORY : A stack is a linear data structure that can be represented in the real world in the form of a physical stack or a pile. The elements in the stack are added or removed only from one position ie 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 element is known as the 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 then initialize to an empty list

output :

Unnati Gangar

```
>>> s. push (20)
>>> s. I
[20, 0, 0, 0, 0]
>>> s. pop()
('data:', 20)
>>> s. I
[20, 0, 0, 0, 0]
>>> s. Push (10)
    s. Push (20)
    s. Push (30)
    s. Push (40)
    s. Push (50)
>>> s. a
[10, 20, 30, 40, 50]
>>> s. pop ()
('data:', 50)
```

s.peek()

```
s. peek ()
stack empty
```

05/01/2020

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

Step 4: Use if statment to give the condition that if length of given list is greater than the range of list then print stack is full.

Step 5: Or else print the statment as insert the elements into the stack and initialize the values.

Step 6: Push method used to insert the element but POP method used to delete the element from the stack

Step 7: If pop method value is less than 1 then return the stack is empty or else delete the element from stack at the top most position.

Step 8: First condition checks whether the no. of element are zero while the second case whether tos is assigned any value. If tos is not assigned any value, the

output

Unna
>>> s
>>> s.
[20,
>>> s po
(' de
>>> s.a
[20,
>>> s.
s
s
s
s.
>>> s.
[10,
>>> s
(

Continuation of source code.

```
def Peek (self):
    if self.tos < 0:
        Print ("stack empty")
    else:
        a = self.I [self.tos]
    Print ("data =", a)
s = stack ()
```

value can be store that stack is empty.

step 9: Assign the element values in push method to an " ", and print the given value in pop at not.

step 10: Attack the input and output of above algorithm.

03/01/22

Queue:

```
class queue
    global f
    global r
    global a
    def __init__(self):
        self.f = 0
        self.r = 0
        self.a = [0,0,0,0,0]
def enqueue(self, value):
    self.n = len(self.a)
    if(self.r == self.n):
        Print("Queue is full")
    else:
        self.a[self.r] = value
        self.r += 1
        Print("Queue element inserted", value)
def dequeue(self):
    if(self.f == len(self.a)):
        Print("Queue is empty")
    else:
        value = self.a[self.f]
        self.a[self.f] = 0
        Print("Queue element deleted", value)
        self.f += 1
b = queue()
```

PRACTICAL NO : 06

AIM : Implementing a queue using python' list

THEORY : Queue is a linear data structure which h
2 references front and rear. Implementing a
queue using python list is the simplest as s
python list provides inbuilt functions to perfor
the specified operations of the queue. It is based
the principal that a new element is inserted a
rear and element of queue is deleted after
which is at front. In simple terms, a queue
can be described as a data structure based on
first out (FIFO) principle.

Queue () - Create a new empty queue

Enqueue () - Insert an element at the rear of th
queue q similar to that of insertion of
linked using tail

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

output :

```
>>> b.enqueue(4)
('Queue element inserted', 4)
>>> b.enqueue(5)
('Queue element inserted', 5)
>>> b.enqueue(6)
('Queue element inserted', 6)
>>> b.enqueue(7)
('Queue element inserted', 7)
>>> b.enqueue(8)
('Queue element inserted', 8)
>>> b.enqueue(9)
('Queue element inserted', 9)
Queue is full
>>> print(b.a)
[4,5,6,7,8]
>>> b.dequeue()
('Queue element deleted', 4)
>>> b.dequeue()
('Queue element deleted', 5)
>>> b.dequeue()
('Queue element deleted', 6)
>>> b.dequeue()
('Queue element deleted', 7)
>>> b.dequeue()
('Queue element deleted', 8)
>>> b.dequeue()
Queue is empty
>>> print(b.a)
[0,0,0,0,0]
```

Algorithm:

step 1: Define a class queue and assign global variables then define init (method) with a argument is init (), assign or initalize the values with the help of 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 statment that length is equal to rear the queue is free or else insert the elements in empty list or display that queue element added succesfully & increment by 1.

Step 4: Define dequeue () with self argument under this use if statment 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 & increment it by 1.

Step 5: Now call the Queue () function and give the element that has to be added is the empty list by using enqueue () and print the list after adding and same for deleting & display the list of deleting the element from the list.

SDD

PRACTICAL - 07

AIM : Program on Evaluation of given string by using stack in python environment ie, 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 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 statment

```
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 = " 8 6 9 * + "
r = evaluate(s)
print("The evaluated value is:",r)
```

Step 5: 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 is 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 output and input of above algorithm

17/01/2

Print ("Unnati Gangar")

output

(The evaluated value is :) 62)

Unnati Gangar.

SOURCE CODE:

```
Class node:
        global data
        global next
        def - - (init) - - (self, item):
                Self data = item
                self next = None
Class linked list:
        globals
        def - - init - - (self):
                self.s = None
        def add L (self, item):
                new node = node (item)
                if self.s = = None:
else:
        head = self.s
        while head.next! = None
                head = head.next
                head.next = new node
def add B (self, item):
        new node = node (item)
        if self.s = = None:
                self.S = new node
else: new node.next = self.a
def display (self):
        head = self.s
        while head.next 1 = None:-
        print (head, data)
```

PRACTICAL NO : 08

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 rock in a linear fashion but not necessarily contiguous. The individual element of the linked list called a node. Node comprises of 2 parts 1) Data 2) Next. Data stores all the information w.r.t the element for example - roll no name, address etc. Whereas next refers to the next node. In case of the largee list, if we add / remove any element from the list, all the elements of list has to adjust itself.

Algorithm :-

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

Step 2 : The entire linked list means can accessed to use the first node of the linked list.

050

head = head.next
print (head.data)

Step 3: Thus, the entire linked list can be traversed using this node which is linked list. the head pointer of the

Step 4: Now that we know that we can transverse the entire list using the head pointer, we should only use it to refer the first node of list only.

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

Step 6: We will use the temporary node as a copy of the node. We are currently transversing. Since we are making temporary node a copy of current node. The datatype of the temporary node should also be note.

Step 7: Now that current is displaying to the 1st node, if we want to access 2nd node of list we can refer as the next node of the 1st node.

Step 8: But 1st node is refered by current node so we can transverse it to 2nd nodes as h = h.next.

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

OUTPUT :-

```
>>> Start.add_L(40)
>>> Start.add_L(30)
>>> Start.add_L(20)
>>> Start.add_B(10)
>>> Start.add_B(50)
>>> Start.display()
50
10
10
40
```

PRACTICAL - 09

AIM : Program based on binary search tree by implementing Inorder, Preorder & postorder Traversal

THEORY : Binary tree is a tree with support maximum of 2 children from 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 another as right child.

Inorder : (i) Transverse the left subtree inform might have left and right subtrees.
(ii) Visit the root node.
(iii) Tranverse the right subtree & repeat it

Preorder : (i) Visit the root node
(ii) Transverse the left subtree. The left subtree inform might have left and right subtree. Repeat it.

E20

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

Algorithm:

Step 1: Define class node and define init() method with 2 argument. Initializing 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 in node. none.

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

Step 4: Use If statment for checking the condition that root is none then use else statment for, if node is less than the main node then put or arrange that in leftside.

CODE:
class no
d
class

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(value)
        if self.root == None:
            self.root = p
            print("root is added success")
        else:
            n = self.root
            while True
                if p.val < n.val:
                    if n.left == None:
                        n.left = p
                        print(p.val, "node is added
                        to left side successfully at in val")
                        break
                    else:
                        n = n.left
                else:
                    if n.right == None:
                        n.right = p
```

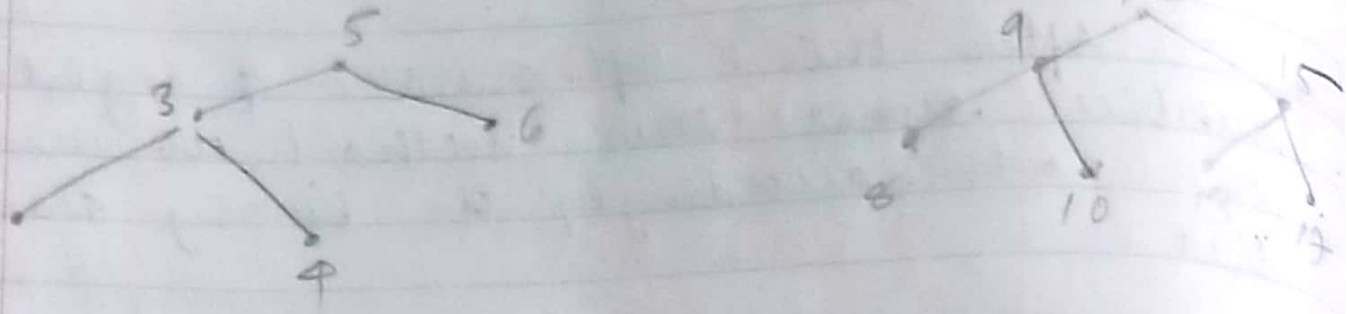
```
            print(p.val, "node is added to rightside sucessfully at", h.val)
            break
        else:
            h = h.right
def Inorder(root):
    if root == None:
        return
    else:
        Inorder(root.left)
        print(root.val)
        Inorder(root.right)
def Preorder(root):
    if root == None:
        return
    else:
        print(root.val)
        Preorder(root.left)
        Preorder(root.right)
def Postorder(root):
    if root == None:
        return
    else:
        Postorder(root.left)
        Postorder(root.right)
        print(root.val)

t = BST()
```

Step 11: For postorder, in else part assign left then right and then go for root node.

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

Inorder: LVR.

step 1



step 2

step 3

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

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 statment within that else statment for checking that node is greater than main root then put it into rightside.

step 7: After this, left subtree & right subtree repeat this method to arrange the node accordingly to binary search tree

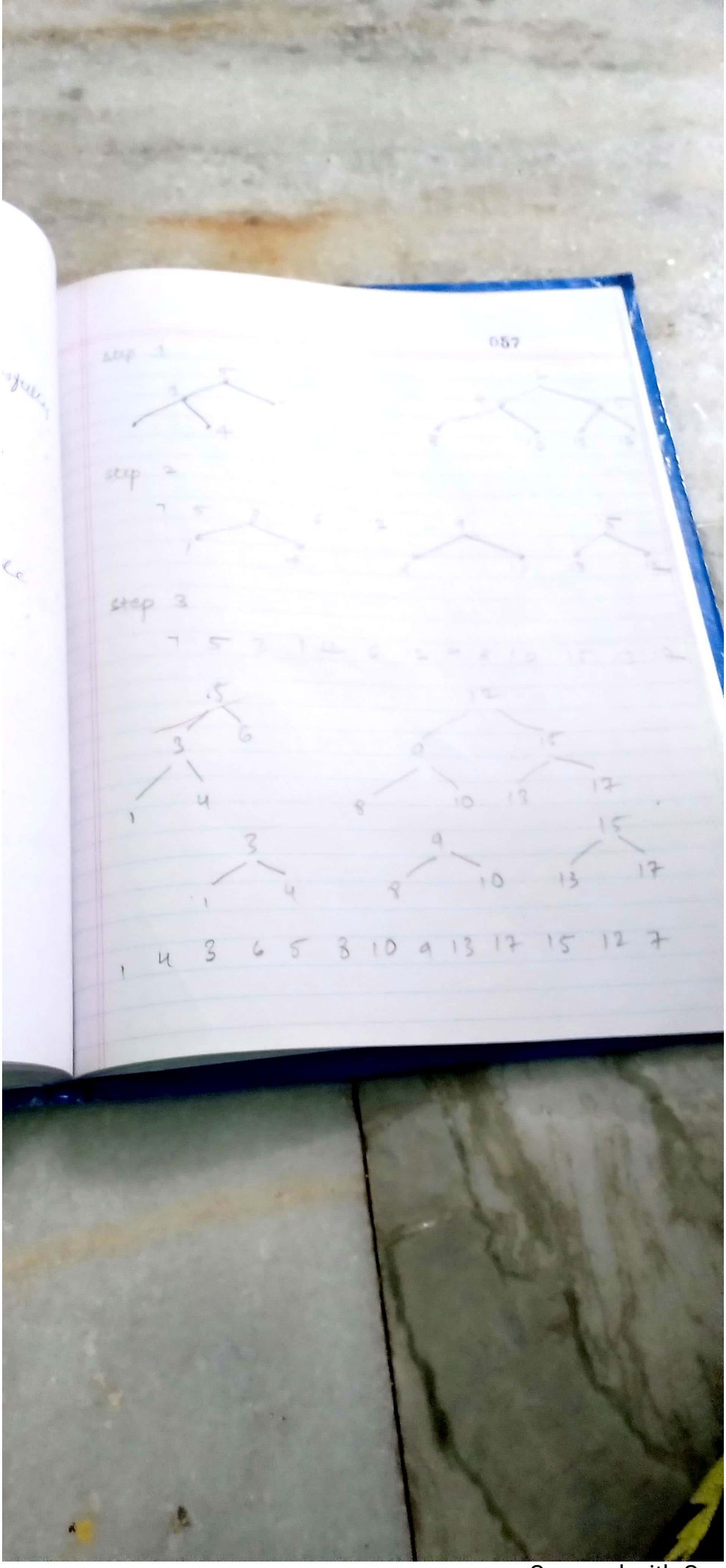
step 8: Define Inorder(), preorder() and postorder() with root argument and use if statment that root is none and return that in all.

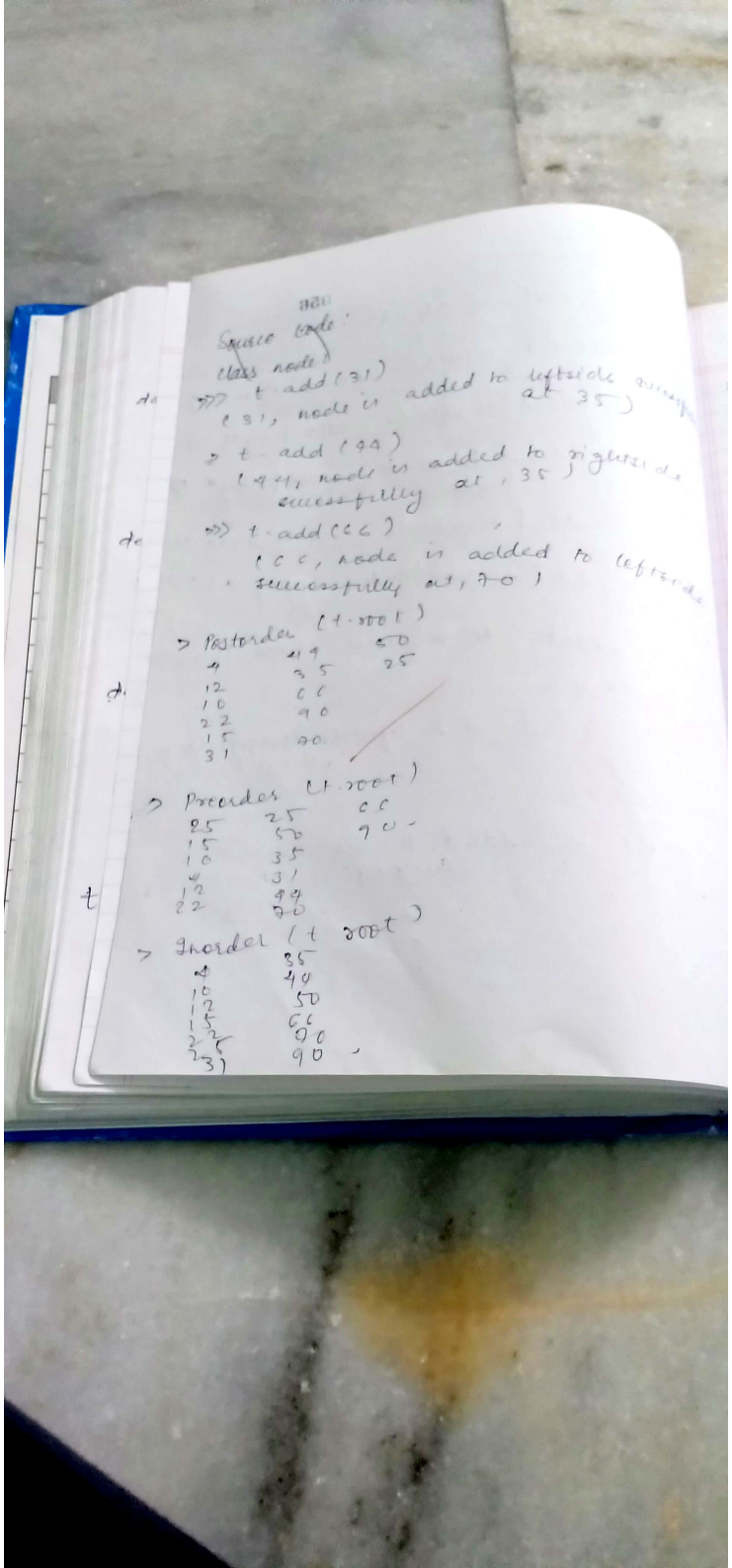
step 9: In Inorder, else statment used for giving that Condition first left, root and then right node.

step 10: For preorder, we have to give Condition in else that first ~~node~~ root, left and then right node.

Output

t.add(25)
root is added successfully at 25.
t.add(15)
>>>
(15, node is added to leftside successfully at 25)
t.add(50)
>>>
(50, node is added to leftside, successfully at 15)
>>> t.add(10)
(10, node is added to leftside, successfully at 15)
>>> t.add(22)
(22, node is added rightside successfully at 10
t.add(4)
(4, node is added rightside successfully at 22)
t.add(12)
>>>
(12, node is added to rightside successfully at 10
>>> t.add(35)
(35, node is added to leftside successfully at 50)
>>> t.add(70)
(70, node is added to rightside successfully at 50),





PRACTICAL - 10

AIM: To sort a list using mergesort

THEORY: Like Quicksort, mergesort is divide and conquer algorithms. It divides input array into halves calls itself for the two halves & then merge the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] & arr[m+1..r] are sorted sub arrays into one. The array is recursively divided into two halves till size comes into action and starts merging arrays back till complete array is merged.

Application

Mergesort is useful for sorting linked lists in O(nlogn) time.

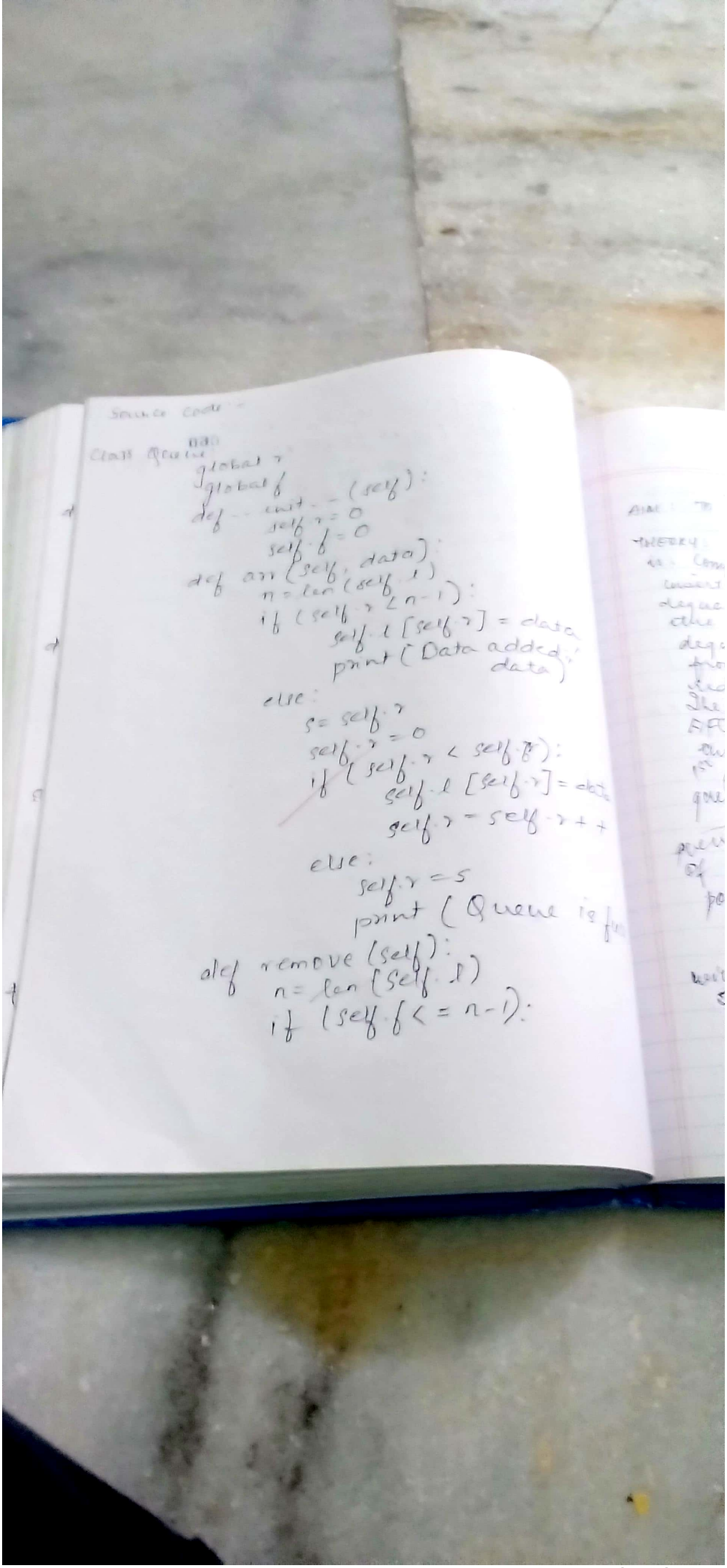
Mergesort is useful for sorting access data sequentially & the need of random access is low.

2. Inversion Count problem.
3. Used in External Sorting

Source code:

```
def mergesort(arr):
    if len(arr)>1:
        mid = len(arr)//2
        lefthalf = arr[:mid]
        righthalf = arr[mid:]
        mergesort(lefthalf)
        mergesort(righthalf)
        i=j=k=0
        while i<len(lefthalf):
            arr[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j<len(righthalf):
            arr[k]=righthalf[j]
            j=j+1
            k=k+1
arr=[27,89,55,62,99,45,14,10]
Print("Random list:",arr)
mergesort(arr)
Print("\n mergesorted list:", arr)
```

Mergesort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially and thus is popular in languages where sequentially accessed data structure are very common.



Source code :

```
Class Queue:
    global r
    global f
    def __init__(self):
        self.r = 0
        self.f = 0
    def add(self, data):
        n = len(self.l)
        if (self.r < n-1):
            self.l[self.r] = data
            print("Data added:", data)
        else:
            s = self.r
            self.r = 0
            if (self.r < self.f):
                self.l[self.r] = data
                self.r = self.r + 1
            else:
                self.r = s
                print("Queue is ful
    def remove(self):
        n = len(self.l)
        if (self.f <= n-1):
```

output:

Random list : [27, 87, 70, 55, 62, 99, 48, 14, 10]

Mergesort list : [10, 14, 27, 48, 55, 62, 70, 89, 94]

Practical - 11

AIM : TO demonstrate use of circular queue.

THEORY : In a linear queue, once queue is completely full, its not possible to insert more elements. Even if we dequeue the queue to remove some of the elements can be inserted when we dequeue any element to remove it from the queue forward, thereby reducing the overall size of the queue. The only way to add next linear queue is FIFO, but instead of ending the queue at the last position after the last, hence making the queue behave like a circular data structure

In case of a circular data structure queue head pointer will always point from of the queue & tail pointer will always point end of the queue.

Initially the head & tail pointer will be pointing to the same location this would mean the queue is empty.

New data is always added to location pointed by the tail pointer and once the data is added, tail pointer is incremented to point to the next available location.

Applications

Below we have some real-world examples

1. Computer Controlled traffic signal system
2. CPU scheduling & memory management.

```
        print("Data removed:", self.l[self.b])
        self.l[self.b] = 0
        self.b = self.b + 1
    else:
        s = self.b
        self.b = 0
        if (self.b < self.r):
            print(self.l[self.b])
            self.b = self.b + 1
        else:
            print("Queue is empty")
            self.b = s
q = Queue()
```

OUTPUT:-

```
q.Queue()
q.add(10)
data added: 10
q.add(20)
data added: 20
q.l
[10, 20, 0, 0, 0, 0, 0]
q.remove()
[0, 20, 30, 0, 0, 0]
```