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) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                arr[k] = lefthalf[i]
                i = i+1
            else:
                arr[k] = righthalf[j]
                j = j+1
            k = k+1
        while i < len(lefthalf):
            arr[k] = lefthalf[i]
            i = i+1
            k = k+1
arr = [27, 89, 70, 55, 62, 99, 45, 14, 10]
print("Random list : ", arr)
mergesort(arr)
print("\n merge sorted list : ", arr)
```

output:

Random list : [27, 89, 70, 55, 62, 99, 45, 14, 10]

merge sorted list [10, 14, 27, 45, 55, 62, 70, 89, 99]

Practical no: 11

Aim: To sort a list using merge sort

Theory: 2 we quicksort mergesort is a divide and conquer algorithm. It divides input array in 2 halves and is key progress that assumes that arr[l...m] and merges the two sub array into one. The array is recusing divided in two halves till the size one the size because is the merge process comes into action and starts merging back till it

Applications.

Mergesort is useful for sorting linked it in of nodes) the need of random access is low Inversion count problem used in external sorting.

Mergesort is merge efficiently quicksort sometypes of list if the data to be sorted can only be efficiently accessed potentially

28/02/2025

i) computer controlled Traffic signal system uses circular queue
ii) CPU scheduling and memory management


```
else:
  s = self.f
  self.f = 0
  if (self.f (self.s):
     print (self.f{self.f})
     self.f = self.f+1
else:
     print ("queue is empty")
     self.f = 5

q= queue()
```

output

```
>>> q.add (100)
  Data added : 100
>>> q.add (200)
  Data added : 200
>>> q.add (300)
  Data added : 300
>>> q.remove ()
  Data removed : 100
>>>
  {0.200.300.0,0,0}
```

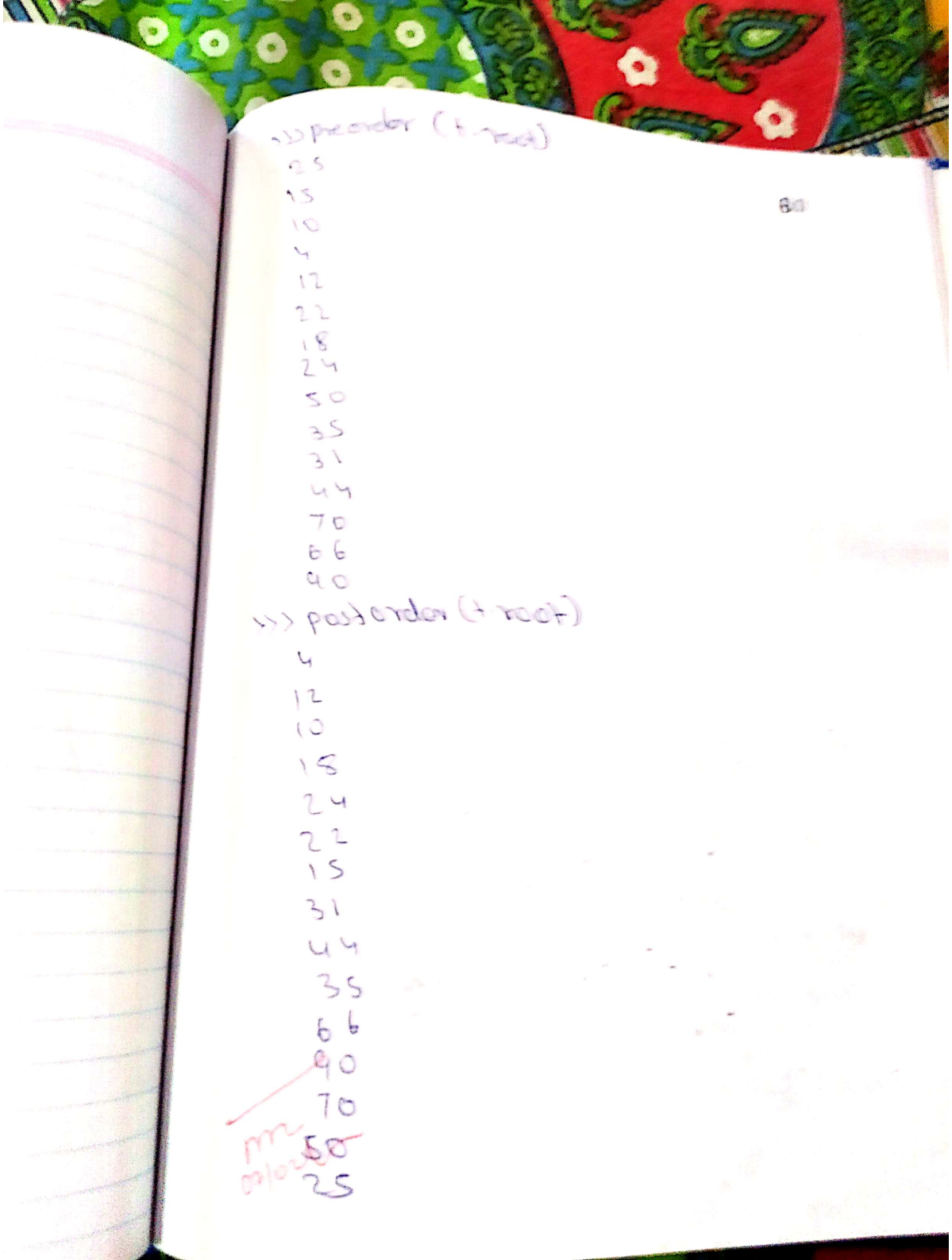
Scanned with CamScanner

```
class Queue
    global r
    global f
    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
            print("data added:",data)
            self.r=self.r+1
        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 Full")
    def remove(self):
        n=len(self.l)
        if(self.f<=n-1):
            print("Data removed:",self.l[self.f])
            self.l[self.f]=0
            self.f=self.f+1
        else:
            print("Queue is empty")
            self.f=f
q=Queue()
```


Practical No - 10

Aim:- To demonstrate the use of circular queue

Theory: In a linear queue once the queue is completely full, its not possible to insert more elements even if we dequeue the queue to remove some moving the moments when we dequeue and element we also actually and we cannot insert new element because the rear pointer is still at the end of the queue the only way is to reset the linear queue for a fresh start. Circular queue is also a linear data also Q's which follows the principle of FIFO but instead of ending the queue at the last option it again starts from the first option after the last, hence making the queue behave In case of a circular queue head pointer will change ~~In case~~ Initially the head and tail pointer will be pointing to the same location this would mean that queue is empty. New data is always added to the location pointed by the tail pointer is incremented by the point to the next ~~m~~ available location application before we have some common real world examples where circular queues are used



post order (VLR)

step1

step2

step3 7

post order (LRV)

step1

step2

step3

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

step: In order else statement used for giving that condition first left root and run right node

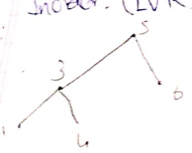
step: for preorder we have to give condition in else that first root, left and run right node;

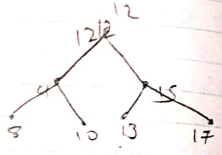
step: for postorder In else part assign left then right and run right node

step: display the output and input of above Algorithm.

Inorder: (LVR)

step:





step:

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

```
def postorder (root):
    if root == None:
        return
    else:
        postorder (root.left)
        print (root.val)
        postorder (root.right)
```


t: BST()

output:

```
>>> t.add(25)
Root is added successfully 25
>>> t.add(15)
15 node is added to ~~right~~ left successfully at 25
>>> t.add(50)
50 node is added to right successfully at 25
>>> t.add(10)
10 node is added to left side successfully at 15.
>>> t.add(22)
22 node is added to right side successfully at 15.
>>> t.add(35)
35 node is added to left side successfully at 50.
>>> t.add(70)
70 node is added to right side successfully at 50.
>>> t.add(4)
4 node is added to left side successfully at 10
>>> t.add(12)
4 node is added to right side successfully at 10
>>> t.add(18)
18 node is added left side successfully at 22
```

```
        else:
            h=self.root
        while True:
            if p.val < h.val:
                if h.left == None:
                h.left = p
                print(p.val, "node is added to leftside success
at", h.val)    else:
                break
                h = h.left

            else:
                if h.right == None:
                h.right = p
                print(p.val, "Node is added to rightside success
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:
            Preorder(root.left)
            print(root.val)
            Preorder(root.right)
```

Algorithm:

57

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

Step 2: Again Define a class BST that is Binary search tree with init() method with self argument and assign the root is none.

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

Step 4: use If statement for checking the condition that if node is less than the main node then put or arrange that in leftside

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

Step 6: use if statement within that else statement for checking that node is greater than main root then put it into rightside

Step 7: Apply this left subtree and right subtree repeat this method to arrange the node according to Binary search tree

Practical - a

Aim: Program based on binary search tree by implementing Inorder preorder & postorder Traversal

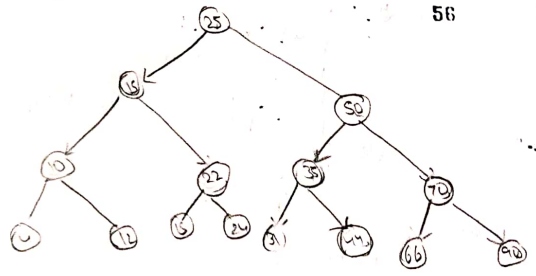
Theory: Binary tree is a tree which supports maximum of 2 children for any node within the tree. Thus any particular node can have either 0 or 1 or 2 children there is another identity of binary tree that it is ordered such that one child is identified as left child and other as right child.

-> Inorder: i) Traverse the left subtree. The left subtree might have left and right subtrees.
ii) visit the root node
iii) Traverse the right subtree and repeat it.

-> Preorder: i) visit the root node
ii) Traverse the left subtree. The left subtree might have left and right subtree
iii) Traverse the right subtree repeat it

-> Postorder: i) Traverse the left subtree. The left subtree might have left and right subtrees.
ii) Traverse the right subtrees
iii) visit the root node

* Binary search tree



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 sucessfully", p.val)
```

output

>>> start.addL(50)
>>> start.addL(60)
>>> start.addL(70)
>>> start.addL(80)
>>> start.addB(40)
>>> start.addB(30)
>>> start.addB(20)
>>> start.addB(10)
>>> start.display()

20
30
40
50
60
70
80

step 8: Now that current is referring to the first node, if we want to access 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 in the linked list is referred by linked does not have any next node. the value in the next field of the last node is None

step 13: So we can refer to last node of linked list self.s = None

step 14: We have to now see how to start transverse the linked list. & how to identify whether we reached the last node of linked list or not

step 15: Attach the coding or input and output of above algorithm

Step 5: we should not use the head pointer to traverse the entired linked list because the head pointer can lead to changes which we cannot revert back

Step 6: we may lose the refference to the 1st node in our linked list and hence most of our unwanted changes to the 1st node, we will use a temporary node to traverse the entire linked list.

Step 7: we will use this temporary node as a copy of the node we are currently traversing since we are making temporary node a copy of current node the data type of the temporary node should also be Node

```
def add(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 add(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)
s1 = linkedlist()
```


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

Theory: A linked list is a linear data structure which store the elements in a node in a linear fashion but no necessarily contigous. The individual element of the linked list called a node. Node consists of 2 parts ① data ② next. If we add / remove any element from the list, all the element of list has to adjust itself every time we add it is very tedious task so linked list is used to solving this type of problems

Algorithm:

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

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

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

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

Step 6: If 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 output and input of above algorithm

Out put

>>> Evaluated value is: 10

```
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 = '5 8 3 - +'
r = evaluate(s)
print("evaluated value is:", r)
```


Aim: postfix Implementation

Theory

The postfix expression is free of any parentheses further we took care of the priorities of the operators in the program. A given postfix expression can easily be evaluated using stack. Reading the expression is always from left to right in Postfix

Algorithm

Step1: define evaluate as function then create a empty stack in python

Step2: convert the string to a list by using the string method 'split'.

Step3: Calculated the length of string and print it.

Step4: use for loop to assign the range of string then give condition using. If statement

Step5: 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'.

```
>>> q.enqueue(1)
('element inserted ...', 1)
>>> q.enqueue(2)
('element inserted ...', 2)
>>> q.enqueue(3)
('element inserted ...', 3)
>>> q.enqueue(4)
  queue is full
>>> q.l
[1, 2, 3]
>>> q.dequeue()
  1
element deleted ..
>>> q.dequeue()
  2
element deleted....
>>> q.dequeue()
  3
element deleted...
>>> q.dequeue()
  queue is empty
>>> q.l
[0, 0, 0]
```

10/9/24

No.

Algorithm

Define a class queue and assign global variable then define init() method with self argument in init() assign or initialize the initial value with the help of self argument

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

Use a statement that length is equal to rear then queue is full or else insert the element in empty list or display that queue element added success fully and increment by 1

Define dequeue() with self argument under this use if statement that front is equal the length of the 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

Now call the enqueue() function and give the element that has to be added in the empty list by using enqueue() and print the list after adding and same for deleting and display the list after deleting the element from the list

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


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 principal that a new element is inserted after rear and element of queue is deleted which is at front. In simple terms, a queue can be described as a data structure based on first in first out FIFO principle

- Queue(): creates a new empty queue
- Enqueue(): Insert an element at the rear of the queue and similar to that of insertion of linked using tail.
- Dequeue(): Return 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

c) use if statement to give the condition that range of list then print stack is full

d) or else print statement as insert the element into the stack and initilize the value

e) push method used to insert the element but pop method used to delete the element from the stack

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

g) This condition checks whether the no of element are zero while the second case when top is assigned we can be sure that stack is empty.

h) Assign the element value in push method to and print the given value as poped not

i) Write the input and output of above algorithm.

output

Name Sangwan

Data : 56

```
>>> s1
[10, 20, 30, 40, 50]
>>> s.pop()
>>> s.pop()
>>> s.l
[10, 20, 30, 40, 50]
>>> s.pop()
>>> s.pop()
>>> s.pop()
>>> s.l
[0, 0, 0, 0, 0]
>>> s.push(10)
>>> s.push(20)
>>> s.push(30)
>>> s.push(40)
>>> s.push(50)
>>> s.l
[10, 20, 30, 40, 50]
```

s.peek()

?

A.

```
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 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()
def peek(self):
if self.tos < 0:
    print("stack empty")
else:
    a = self.i[self.tos]
    print("data:", a)
s = stack()
```

Practical - 5 47

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 stack are added or removed only from one position i.e. the topmost position. Thus the stack work on the LIFO (LAST IN first OUT) principle as the element that was inserted lastest has the free basic operation push pop, peek, The operation of adding and removing the element is known as push & pop

Algorithm

1) create a class stack with instance variable item.
2) define the init method with self argument and initialize the inirial value and the initialization on empty list
3) define method push and pop under the class stack

```
for b in range(0,5):
    b=input("Enter element")
    alist.append(b)
    n:list(alist)
quick(alist)
print(alist)
```

output

```
Enter range for list 5
Enter element 4
Enter element 3
Enter element 2
Enter element 1
Enter elment 8
[1,2,3,4,8]
```

d) At the point where right marker becomes less than leftmost we replace position of rightmost & new split point

e) The pivot value can be exchanged with the content of split point and PV (pivot value) is now in place

f) In addition all the items to left of split point are less than pivot and all the items to the right of split point are greater than PV. The right half and quick sort can be invoked recursively on the two halves

g) The quicksort function invokes a recursive funct quicksort

h) Mergesort begin with some base called as mergesort

i) If length of the list is less than or equal to one, it is

j) If it's greater than it can be partitioned and recursively sorted

k) The partition function implements the process described earlier

l) Display and study the coding and output of above algorithm

```
def sort(alist):
    quickSort(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 not done:
        while l<=r and alist[l]<=pivot:
            l = l+1
        while alist[r]>=pivot and r>=l:
            r = r-1
        if r<l:
            done = True
        else
            t = alist[l]
            alist[l] = alist[r]
            alist[r] = t
    t = alist[first]
    alist[first] = alist[r]
    alist[r] = t
    return r
print("Enter range:")
```

Practical - 4 45

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

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 the list.

② The partition process will happen next. It will find out less than or greater than pivot value

③ partitioning begins by locating two position markers. lets call them left mark and right mark at the beginning of the remaining items, which also converging to the split point

④ We begin by incrementing until we locate a value that is greater the PV we then decrement rightmark until we find value that is less than PV. These items are out of place with respect to eventual split point

5A

5) Here are n element to be sorted thus the process mentioned above should be repeated n-1 to get the required result

6) Stick the output and input of above algorithm of bubble sort step wise

```
a=list(input("Enter list:"))
for i in range(0,len(a)-1):
    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

Enter list: 65, 75, 30, 90, 2, 11

[2, 11, 30, 65, 75, 90]

Aim : Implementation of Bubble sort program on given list.

Theory: Bubble sort is based on the idea of repeatedly comparing pairs of adjacent element and they exist in wrong order. This is the simplest form of sorting available. In this we sort the given elements in ascending or decending order by comparing two adjecent element at a time

1. Bubble sort algorithm start by comparing the first two element of an array and swapping if necessary
2. If we want the element of array in ascending order then first element is greater than second then we need to swap the element
3. If the element is smaller than second then we do not swap the element
4. Again second and third element are compared go on until last and second last element swapped and swapped

Step 3: use if loop to give the range in which element is found in given range then displays message "Element not found"

Step 4. Then use else statement. if statement is not found in range then satisfy the below condition element that element has to be searched

Step 5. Initialize first to 0 and last to last element of the list as array by starting from total count use for loop and assign the given range.

Step 6. If statement in list and still the element to be searched is not found then find the middle element (m)

Step 7: Repeat till you found the element which is input and output of above algorithm

Practical : 2

Aim: Implement Binary search to find an searched no. in a list.

Theory

Binary search

Binary Search is also known as half interval search, logarithmic search or binary chop is a search algorithm that finds the position of a target value within a sorted 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 and assign it to a variable list. use for loop, add element in list using append() method.

Step 2: use sort() method to sort the accepted element and assign it in increasing ordered list print the list after sorting

step 6: use if loop that the element in list is equal to the element accepted the statement that the element is ~~no~~ found along with the element position.

step 7: - 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 8: Attach the input and output of above algorithm

Sorted

```
s = list (input ("Enter the element"))
s.sort ()
print s
a = int (input ("Enter the number to be searched"))
for i in range (len (s)):
    if (a == s[i]):
        print "number found ! in the position", i,
        break

else:
    print "number not found"
```

output

Enter the element : 9, 6, 7, 12, 5, 8

[5, 6, 7, 8, 9, 12]

Enter the number to be searched = 8

number found ! in position 3

2) sorted

step 1: create empty list and assign it to a variable accept total no of element to be inserted into the list from user, say 'n'

step 2:- use for loop for using append () method to add the element in the list. use sort () method to sort the accepted element and assign in increasing order the list, then print the list

step 3: use If statement to give the range in which "Element not found".

step 4: Then use else statement if element is not found in range then satisfy the given condn

step 5:- use for loop in range from 0 to the total no. of elements to be searched before doing this accept on search no from user using Input statement

7. unsorted

Algorithm

Step1: create an empty list and assign it to a variable

Step2: Accept the total no of element to be inserted into the list from the user e.g n

Step3 use for loop for adding n elements into the list

Step4: print the new list

Step5. Accept the element from the user that to be searched in the list

Step6. use for loop in a range from 0 to the total no of elements to search the element from the list

Step7. use if loop that the element in the list is equal to the element accepted from user.

Step8. If the element is found then print the statement with the element position

Step9: use another if loop to print that the element is not found if the element isn't in the list

Step10. Draw the output of given algorithm

unsorted

```
s=int(input("Enter the required number"))
a=[10,12,9,14,17,9]
for i in range(len(a)):
    if[a[i]==s]:
        print "record number found in position",i
        break
if (s!=a[:]):
    print "The required number not found"
```

output

Enter the required number 9
required number found in position 2

Practical no:-1

Aim: Implement linear search to find an Item in the list

Theory:

Linear Search

Linear Search is one of the simpliest searching algorithm in which targeted searching algorithm in which each item in the list

It is the worst searching algorithm with worst case time complexity It is a force approach and other hand in case of list in sequence. A binary search is used which will start by examining the middle term.

Linear search is a technique to compare each

★★ INDEX ★★

No.	Title	Page No.	Date	Staff Member's Signature
1)	Implement linear search to find an Item in the list		29/11/19	
2.	Implement Binary search to find on searched no in the list		5/12/19	mr 10/01/20
3	Implementation of Bubble shot on a list		20/12/19	
4.	Implement quick sort the given list		20/12/19	
5.	Implementation of Stack using Python list	9	3/1/20	mr 03/01/2020
6.	Implementation of a queue using python		10/1/20	mr 10/01/2020
7.	Evaluation of post fix		17/1/20	mr 07/02/2020
8.	Singular Linked list		24/1/20	
9.	programbased on binary search tree by implenating Inorder, preorder & postorder Traversal		7/2/20	

Remarks	Staff Member's Signature
DBMS	04/10/17
DS	