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 item is sequentially matched with each item in the list.

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

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

SE...

```
#unsorted
a=[4,3,5,8,9,2]
s=int(input("Enter a number:"))
for i in range(len(a)):
    if(s==a[i]):
        print("Element found at:", i)
        break
    if(s!=a[i]):
        print("No element found")
```

output:

Enter any list of number 18, 22, 11, 17, 63
Enter a number to be searched: 17
Number searched is found at position: 3

1] Unsorted:.
Algorithm:

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

Step2:. Accept the n total no. of elements to be inserted into the list from the user say n.

Step3:. Use for loop for adding the elements to the list

Step4:. Print the new list.

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

Step6:. use for loop in rauge from '0' to the total no. of elements to search the elements from the list.

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

Step8:. If the element is found then print the statement that the element is found along with the elements position.

Step 9:- Use another if loop to print that the element is not found if the element which is accepted from user is not there in the list.

Step 10:- Draw the output of given algorithm.

sorted

```
S = list(input("Enter the number:"))
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 bounded in position",i)
        break
    else:
        print("no. not bound")
```

Output:

Enter the number: 22,11,36,63,54

a = [11,22,36,54,63]

Enter a number to be searched: 36

Number is bounded at position 2

2] Sorted linear search:
Sorting means to arrange the element in increasing or decreasing order.

* Algorithm:-

step1: create empty list and assign it to a variable.

step2: Accept total no. of elements to be inserted into the list from user, say 'n'.

step3: use for loop for using append() method to add the elements in the list.

step4: use sort() method to sort the accepted element and assign in increasing order the list then print the list.

step5: use if statement to give the range in which element is found in given rauge then display "element not found".

step6: then use ~~else~~ else statement. if ~~state~~ element is not found in range then satisfy the given condition.

Step 7:- Use for loop in range from 0 to the no. of elements to be searched before d this accept an search no from user usi input statement.

Step 8:- Use if loop that the elements in the l equal to the element accepted from

Step 9:- If the element is found then print statement that the element is foun along with the element position.

Step 10:- Use another if loop to print that th element is not found if the elem which is accepted from user is not their in the list.

Step 11:- Attach the input and output of above algorithm.

Practical 2

Aim:- Implement binary search to find an searched number in the list.

Theory:-

Binary search is also known as half interval search algorithm that find the position of the target value within a position of a target value within a position of sorted array. If you are looking for the number which is at the end of the list than you need to search entire list in linear search which is time communing. This can be avoided by using Binary fashion search.

Algorithm:-

step1:- create empty list and assign it to a variable

step2:- using input method accept the rauge of given list.

step3:- Use for loop, add elemut in list using append() methods.

step4:- Use sort() methods to sort, the accept elmuts and assigning in increasing order list point the list after sorting.

Step5:- Use if loop to give the range in which element is found in given range then disp a message element not found.

Step6:- Then use else statement if element is not found in range then satisfy the below condition.

Step7:- Accept an argument and key of the element that element has to be searched.

Step8:- Initialize first to 0 and last element of the list as array is starting from zero hence it is initialised 1 less, then total count.

Step9:- Use for loop and assign the given range

Step10:- If statement in list and still the element to be searched is not found then the middle element (m).

Step11:- Else input the item to the searched is still less than the middle less than Initialize last (h): mid(m) - 1

Else.

Initialize first (l): mid(m) - 1

Step11:- Repeat still you found the element the input & output

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

Output:

```
>>> enter a range = 4
enter a number: 2
[2]
enter a number: 1
[1,2]
enter a number = 4
[1,2,4]
```

#Bubble sort

```
s=list(input("enter list of nos:"))
for i in range(len(s)-1)
    for b in range(len(s)-1-i):
        if s[b]>s[b+1]:
            d=s[b]
            s[b]=s[b+1]
            s[b+1]=d
    print(s)
```

output:-

enter list of nos:- 1,2,5,9,7,80,34,1017

[1,2,5,34,80,97,1017]

Practical 3

3. Bubble sort

Aim: Implementation of bubble sort program on given list.

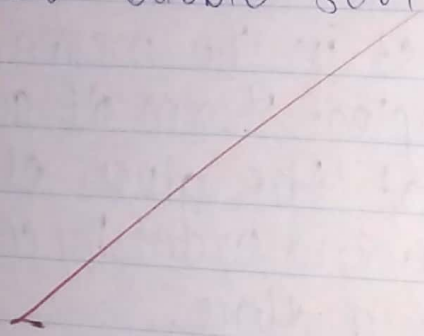
Theory: Bubble sort is based on the ideas of repetitary comparing of adjacent element and then swapping their position. If they exist in the increasing order. This is the simplest form of sorting available in this we sort the given element in ascending or decreasing order by comparing two adjacent element of a time.

Algorithm:

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

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

(6) Stick the output and input of above algo of bubble sort responsible.



```
def quick(alist):
    help(afirst, 0, len(alast)-1)
def help(alist, first, last):
    if first < last:
        split = poat(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[2]<=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
```

4. Quick sort

Aim:- Implement quick sort to sort the given list.

Theory:- The quick sort is a Recursive algorithm the based on the divide and conque technique.

Algorithm:-

Step①:- Click sort first elements a value, which is calle pivot value, first element sense as over first pivot value, since we know that first will eventually and up as last in that list.

Step②:- The partition on process will happen next. It will find the split point and at the same time mov other items to the appropriate side of the list, either less than or greater then pivot value.

Step③:- Parts timing begin by locating two position markers. lets call them left marks and right marks at the beginning and end remaining items in the list. The goal of the partition process is to move items that are on carry lists with respec to print value while also convergion of split point.

Step4:. we begin by increasing left marks until we locate a value that is greater than the p.v. we then decrement right mark until find value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to eventure split point.

Step5:. At the point where rightmarks becomes less than leftmarks, we stop. The position of right marks is now the split point.

Step6:. In addition all the terms to left of split are less than p.v. and all item to the right of split point are greater than p.v. The list can now be divided at split point and quick sorted can be involved necessarily can be two halves.

Step7:. The quickest function involves a recursive function on quicksorthper.

Step8:. Quicksorthelper begin with same base as the merge sort.

Step9:. If length of the list is less than 0 and equal use it already sorted.

```
x = int(input("enter arrauge from list:"))
alist = []
for b in rauge(0, x):
    b = int(input("enter element:"))
    alist.append(b)
    n = len(alist)
quick(alist)
print(alist)
```

output:

enter rauge fromlist 5
Enter element 4
Enter element 3
Enter element 2
Enter element 1
Enter element 8
[1, 2, 3, 4, 8]

23/12/22

step10:. If it is greater, then it can be partition and recursively sorted.

step11:. the position function implement the process described earlier.

step12:. Display and stick the coding with output of above algorithm.

practical no. 5

* 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 i.e. the topmost position. Thus, the stack works on 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 elements is known as push and pop.

Algorithm:-

Step 1:- create a class instance 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.

code:-

```
print("Abhinay vishwakarma")
class stack
    global tos
    def __init__(self):
        self.l=[0,0,0,0,0]
        self.tos=-1
    def push(self,data):
        n=len(self.l)
        if self.tos=n-1:
            print("stack is full")
        else:
            self.tos=self.tos+1
        self.l[self.tos]=data
    def pop(self):
        if self.tos<0:
            print("stack is empty")
        else:
            k=self.l[self.tos]
            print("Data=",k)
            self.l[self.tos]=0
            self.tos=self.tos-1
    def peek(self):
        if self.tos<0:
            print("stack empty")
        else:
            s=self.l[self.tos]
s=stack()
print("data=",s)
```

Output:

```
Abhinay Vishwakarma
Data =50
>>> s.d
  [10,20,30,40,50]
>>>s.pop()
>>>s.pop()
>>> s.d
  [10,20,30,0,0]
>>>s.pop()
>>>s.pop()
>>>s.pop()
>>> s.d
[0,0,0,0,0]
>>> s.push(10)
>>>s.push(20)
>>> s.push(30)
>>>s.push(40)
>>> s.push(50)
  [10, 20, 30, 40, 50]
```

Step3:: Define methods 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 then print stack is full.

Step5:: Or Else print statement as insert the else element ~~statement~~ into stack and initialize the values

Step6:: Push method used to insert the element but pop method used to delete element from the stack.

Step7:: If in pop method, value less than 1 then return the stack is empty or else delete the element from stack at topmost position.

Step9:: ~~Assign~~ the element values in push method to an and print the given value is poped "not".

Step10:: Attach the input and output of above algorithm

Step8:: First condition checks whether the no. of elements are zero while the second case whether tos is assigned a value. If top is assigned not any value then value can be sure that stack is empty.

03/01/22

Practical 6

Aim:- Implementing a Queue using python list.

Theory:- Queue is a linear data structure which has two 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 into inserted after rear and element of queue is deleted which is at front. In simple terms, a queue can be described 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(): Returns the element which was at the front the front is moved to the successive element. A deque operation cannot remove element of queue is empty.

```
class Queue:
    global r
    global f
    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 ("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()
```

output:.

```
>>> b.enqueue(1)
('inserted:', 1)
>>> b.enqueue(5)
('inserted:', 5)
>>> b.enqueue(7)
('inserted:', 8)
```

Algorithm:-

step1:- Define a class queue and assign global variables then define

Practical no:7

Evaluation of postfix expression

Aim:- program on Evaluation of given string by using stack in python environment i.e. postfix.

Theory:- The postfix is free of any paranthesis further we took care of the priorities of the operatoors in the program a given postfix expression can easily be evaluate using stacks. Reading the expression is always from left to Right in position.

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:- calculate the length of string and print it

step4:- use for loop to assign the Range of string then give condition using if statement.

#code –

```
def evaluate(s):
    k = s.split()
    n = len(k)
    stack = []
    for in in range(n):
        if k[i] is digit():
            stack.append(int(k[i]))
        elif k[i] == '+':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(a) + int(b))
        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 69' + '
r = evaluate(s)
print("the evaluated value is", r)
print("fiesto")
```

output →

('the evaluated is', 62)

fiesto

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 's'.

step6: If the token is an operator *, /, +, -, ^ it will need two operands pop the 's' twice the first pop is second operand and the second pop is the first operand.

step7: perform the arithmetic operations, push the result basic on the 'm'.

step8: when the input expression has been completely processed the result is on the stack. pop the 's' and return the value.

step9: print the result of string after the evaluation of postfix

step10: Attach output and input of above algorithm.

Practical 8

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

Theory: A linked list is a linear data structure which stores the elements in a node continue. The individual element of the linked list called a Node. Node comprises of two parts ① Data ② Next. Data stores all the information w.r.t the element for example roll no; name, addres etc. whereas next refers to the next node in case of larger list, if we add/remove any element from the list, all the elements of list has to adjust itself every time we add it very tedious task so linked list in used to solving this type of problems.

Algorithm:

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

Step 2: The entire linked list means can be accessed with the first node of the linked list.

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

```
class node:
    global data
    global next
    def ---- init ---- (self, item):
            self.data = item
            self.next = none

class linked list
    global s
    def --- init --- (self)
            self.s = None
    def addL(self, item):
            newnode = node(item)
            if self.s = None
                self.s = newnode
            else:
                head = self.s
            while head.next != None
                head = head.next
    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)
```

Output:

```
>>> start.addL(50)
>>>start.addL(60)
>>> start.addL(70)
>>>start.addL(80)
>>> start.addB(40)
>>> start.addB(30)
>>>start.addB(20)
>>>start.display()
>>> 20
    30
    40
    50
    60
    70
    80
>>>print("Abhinay vishwakarma")
Abhinay vishwakarma
```

Step4:- Now that we know that we can traverse the entire linked list using the head pointer. we should only use it to refer the first node of the list only.

Step5:- we should not use the head pointer to traverse the entired linked list because the head pointer is our only reference to the 1st node in the linked list modifying the reference of the head pointer can lead to changes which we cannot revert back.

Step6:- we may lose the reference of the 1st node in our linked list and hence most of our linked list so in order to avoid making some unwanted changes to the 1st node, we will use temporary node to traverse the entire linked list.

Step7:- we will use temporary node as a copy of the node we are currently traversing. since we are making temporary node a copy of current node the datatype of temporary node should also be node.

Step8:- 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 1st node.

step9:. But first node is refered by current, so we can traverse the second node as b=h.next.

step10:. Similarily we can traverse rest of node in the linked list using same method by while loop.

step11:. our concern is to find terminating condition by the while loop.

step12:. The last node in the linked list is referred by the tail of linked list, since the last node of linked does not have any next node.

step13:. so we can refer to the last node of linked list self.s=None.

step14:. we have to see now how to start traversing the linkedlist show to identify the last node of linked list.

Practical 9

Aim: Program based on Binary search tree by implementing Inorder, preorders & postorder traversal.

Theory: Binary tree is a tree which supports maximum of two children for any node with 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 as one child is identified as left child and other as Right child.

Inorder: (I) Traverse the left subtree the left subtree again might have left and Right subtree.
(II) Visit root node.
(III) Traverse the right subtree and repeat it

Preorder: (I) Visit the root node.
(II) Traverse the left subtree the left subtree in turn might left & Right subtrees.
(III) Traverse the Right subtree.

Algorithm:-
~~Step 1:~~
Postorder: (1) Traverse the left subtree. the left subtree in turn might have left and Right subtrees.
(2) Traverse the right subtrees
(3) Visit the root node.

Algorithm:

Step1:- Define class node and define init() method with two argument. Initialize the value in this method.

Step2:- Again define a class BST that is binary search tree with init() method with self argument and assign the root is none.

Step3:- Define add() method for adding the node define a variable p that p=node(value)

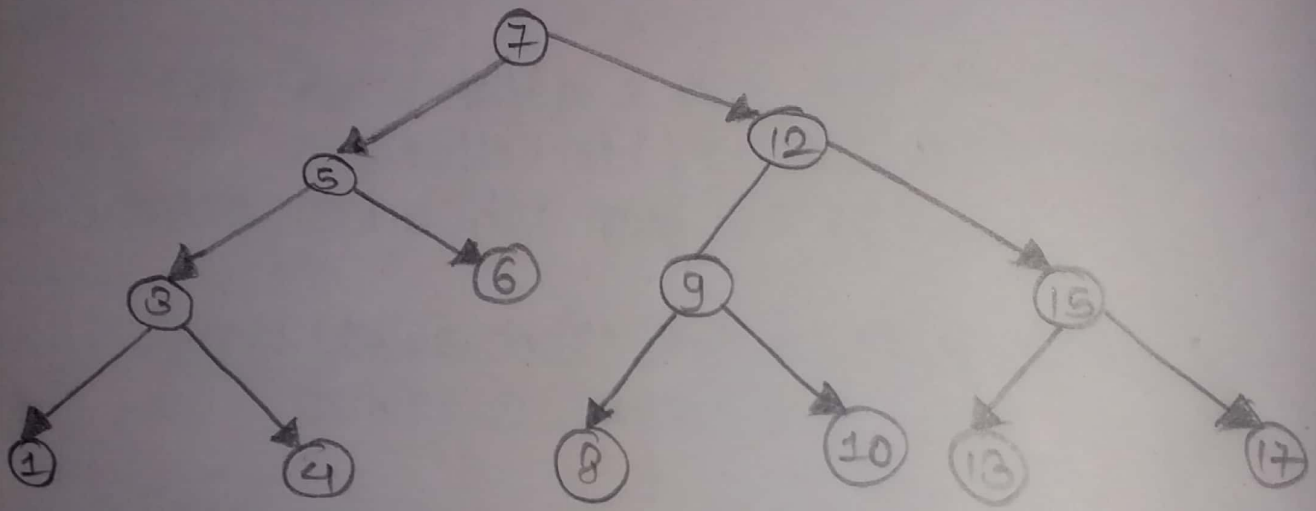
Step4:- use if statement for checking the condition that root is none then use else statement for if node is less than the main node then put or arrange that in leftside.

Step5:- 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.

Step6:- use if statement within that else statement for checking that node is greater than main root then put it into rightside.

Step7:- After this, leftsubtree and rightsubtree repeat this method to arrange the node according to binary search tree.

* Binary Search Tree:-



step8:- Define inorder(), preorder() and postorder() with root argument and use if statement that root is none and return that in all.

step9:- In Inorder, else statement used for giving that condition first left, root and then right node

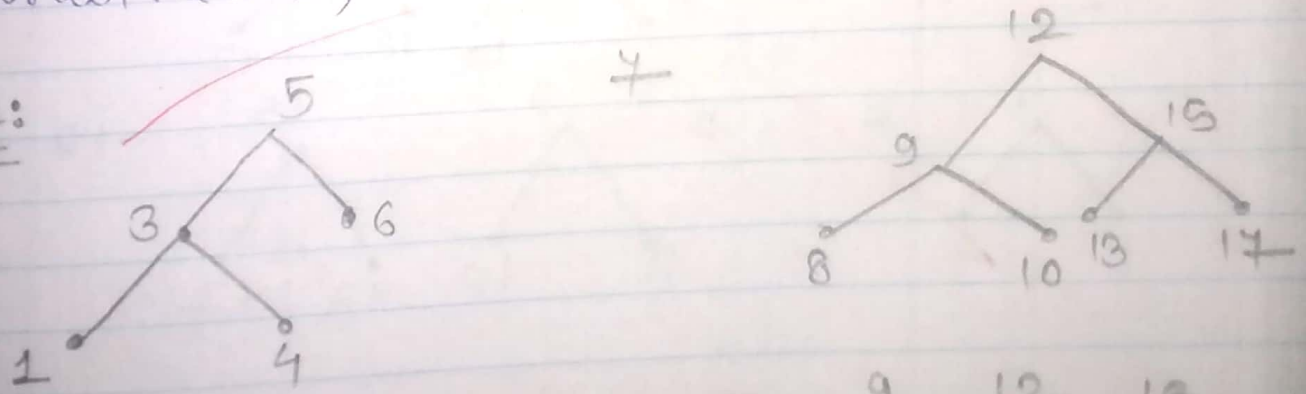
step10:- for preorder, we have to give condition in else that first root, left and then right node.

step11:- for postorder, In else part, assign left then right and then go for root node.

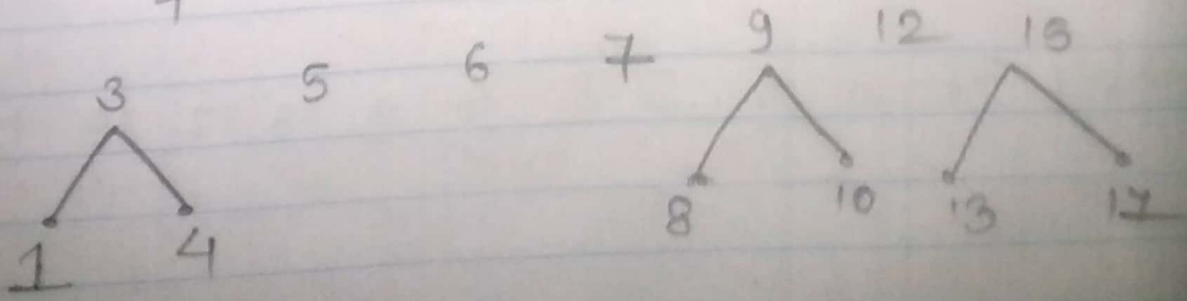
step12:- Display the output and input of above algorithm.

*Inorder: (LVR)

step1:



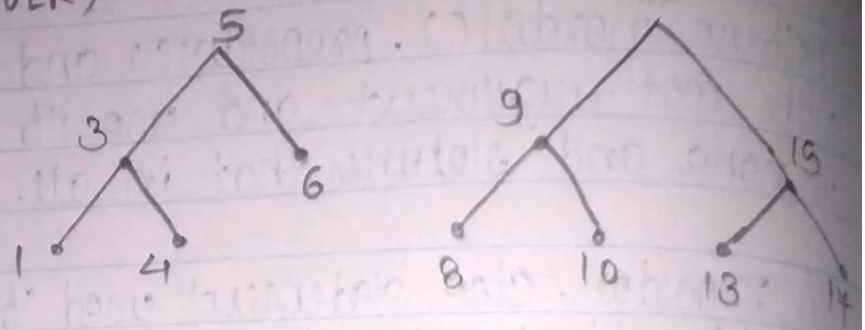
step2:



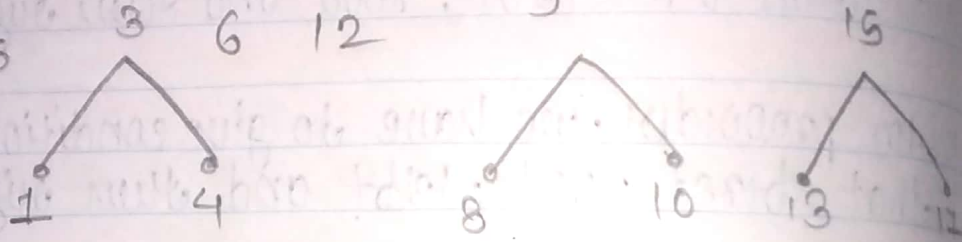
step3: 1 3 4 5 6 7 8 9 10 12 13 15 17

pre-order (VLR)

step 1 7



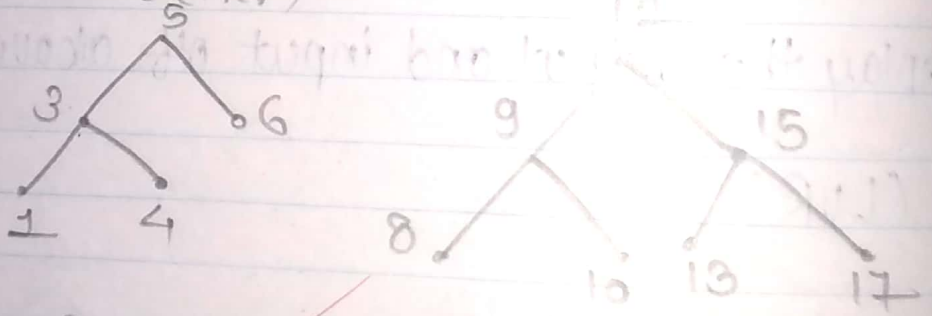
step2 7 5 3 6 12 9 15



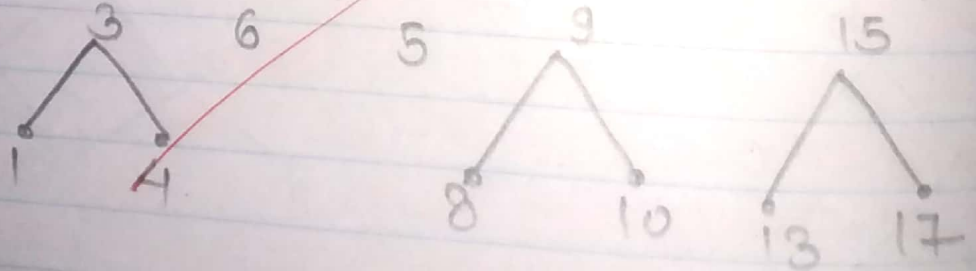
step3: 7 5 3 1 4 6 12 9 8 10 15 13 17

- postorder: (LRV)

step1:



step2:



step3: 8 1 4 3 6 5 8 10 9 13 17 15 12

```
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
arr = [27, 89, 70, 55, 62, 99, 45, 14, 10]
print("RANDOM List:", arr)
mergesort(arr)
print("\n mergesorted List:", arr)
```

Output:

```
RANDOM LIST: [27, 89, 70, 55, 62, 99, 45, 14, 10]
MERGESORTED LIST: [10, 14, 27, 45, 55, 62, 70, 89, 99]
```

Practical 10

Aim: To sort a list using merge sort.

Theory: Like quicksort merge sort is a Divide and conquer algorithm. It divides input array in two 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] and arr[m+1...r] are sorted and merges the two sort sub-arrays into one. The array is recursive divided into two halves till size becomes 1. Once the size becomes 1, the merge process comes into action and starts merging arrays back into the complete array is merged.

Application:

Merge sort is used for sorting linked lists in O(nlogn) time merge sort access data sequentially and the need of random access is low.

2] Inversion count problem

3] used in external sorting

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

Practical 11

Aim: To demonstrate the use of circular queue.

Theory: In a linear queue, once the queue is completely full, it's 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, we are actually moving the front of the queue forward, thereby reducing the overall size. of the queue and we cannot insert new elements, because the rear pointer 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 structure which follows the principle of FIFO but instead of ending the queue at the last position, it again starts from the first position after the last. hence making the queue behave like a circular data structure. In case of a circular queue head pointer will always point to front of the queue and tail will always point to the end of the queue. Initially, the head and the tail pointer will be pointing to the same location, this would mean that the queue is empty.

```
class queue:
    global r1
    global f
    def __init__(self):
            self.r1=0
            self.r1=0
            self.1=[0,0,0,0,0,0]

    def add(self,data):
            n=len(self.1)
            if (self.r<n-1):
                self.l[self.r1]=data
                print("data added:",data)
                self:r1=self.r1+1
            else:
                s=self.r1
                self.r1=0
                if(self.r1<self.f):
                    self.1[self.r1]=data
                    self.r1=self.r1+1
            else:
                self.r1=s
                print("queue is full")

    def remove(self):
            n=len(self.1)
            if (self.f<=n-1):
                print ("data removed.", self.1)
                self:1[self.f]=0   [self.f])
                self.F = self.f+1
```

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

Output:

```
>>> q=Queue()
>>> q.add(44)
('Data added:', 44)
>>> q.add(57)
('Data removed:', 57)
>>> q.remove()
('data removed:', 44)
```

Newdata is always added to the location pointed by the tail pointer, and once the data is added tail pointer is increased to point to the next available location Applications. Below are have some comman real-world examples where circular queue is used:

1] computer controlled traffic signal system uses circular queue.

2] CPU scheduling memory management.

m
28/01/