Practical No. 1

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

Theory:

linear search

- linear search is one of the simplest search algorithm in which targetted searching algorithm in which each item in the list.

It is worst searching algorithm with worst case time complexity. It is a force approach on the other hand in case of an ordered list, instead of searching the list in the 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.

) Linear search :

Algorithm :

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

Step 2 : Accept the total no of elements to be inserted into the list from the user.

Step 3 : Use for loop for adding the elements in the list.

Step 4 : Print the new list.

Step 5 : Accept the element from the user that to be searched in the list.

Step 6 : Use for loop in a range from '0' to the total no of elements to search the element from the list.

Step 7 : Use if loop that the element in the list is equal to the element accepted from user.

Step 8 : If the element is found then print the statement that the element is found along with the element position.

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

Step 10 : Print the output of the

```
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 "required number found at position ";
        break.
if(s != a[i]):
    print "the required number not found"
```

Output

Enter the required number
required number found at position 2

SE

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:

Algorithm

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

Step 2: Use for loop for using append() method to add the elements 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 is found in given range then display "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:

Step 6: Use of loop that the elements in the list is equal to the element accepted from user. If the element is found then print the statement that the element is found along with the element position.

Step 7: Use another of 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.

code

```
a= list (input ("enter the list of elements"))
a= sort ( )
n= len(a)
s= int (input ("enter the number to be searched"))
if (s> a[n-1] or s< a [0]):
        print ("element not found")
else:
    f=0
    l=n-1
    for i in range (0, n):
        m= int ((f+l)/2))
        if s==a[m]:
            print ("the element is found at:", m)
            break
        else:
            if s< a[m]:
                l=m-1
            else:
                f= m+1
```

OUTPUT

>>> Enter the list of element : 2, 7, 8, 9
Enter the number to be searched: 8
(the element is found at: 2)

>>> Enter the list of elements : x, y, z

Practical - 2

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 chop is a search algorithm that finds the position of a target value within 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 and assign it to a variable using input method accept the range of given list. Use for loop, add elements 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 3: Use If loop to give the range of which element is found in given range then display a message "element not found"

Step 4: Then use else statement, if statement is not found in range then satisfy the below condition. Accept an a argument and key of the element that element has to be searched.

Step 5: Initialize first to 0 and last to last element of the list as array is starting from 0 hence it is initialized 1 less than the total count. Use for loop and assign the given range

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

Step 7: Repeat till you found the element or the input. and output of given algorithm

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

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 position if they exist in the wrong order. This is the simplest form of sorting available. In this we sort the given elements 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 we want to sort the elements 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.

Step (4): Again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped.

Step 5: Then all n elements to be sorted then the process mentioned above should be separated n-1 to get the required result.

Step 6: Stick the output input of above algorithm of bubble sort stepwise.

```
def quick (a list):
    help (a list, 0, len (a list)-1)
def help (a list, first, last):
    if first < last:
        split = part (a list, first, last)
        help (a list, first, split-1)
        help (a list, split+1, last)
def part (a list, first, last):
    pivot = a list [first]
    l = first+1
    r = last
    done = False
    while not done:
        while l <= r and a list [l] <= pivot:
            l = l+1
        while a list [r] >= pivot and r >= l:
            r = r-1
        if r < l:
            done = True
        else:
            t = a list [l]
            a list [l] = a list [r]
            a list [r] = t
    t = a list [first]
    a list [first] = a list [r]
```

QUICK SORT

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 the first, will eventually end up as last in that list.

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

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 the partition process is to move items that are on wrong side with respect to pivot value while also converging on the split point.

Step 4: We begin by incrementing leftmark until we locate a value that is greater than the P.V. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered the eventual split point

5: At the point when rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point.

6: The pivot value can be exchanged with the contents of split point and p.v/ pivot value is now in pla

7 In addition, all the items to left of split point are less than p.v & all the items to the right of split point are greater than p.v. The list can now be divided at split point & quick sort can be invoked recursively on the two halves

8: The quicksort function invokes a recursive function, quicksorthelper.

9: quicksort helper begins with same base case as the merge sort

10: If length of the list is less than or equal to one it is already sorted

```
n = input("enter range")
a = []
for b in range(0, n):
    b = input("enter element")
    a.append(b)
    a.sort(a)
print(a)
```

Output

```
Enter range for list 5
Enter element 6
Enter element 3
Enter element 2
Enter element 1
Enter element 8
[1, 2, 3, 6, 8]
```

20/12/19

11. If n is greater than n can be partitioned and recursively sorted

12. The partition function implements the process described earlier.

13. display and check the count and output of above algorithm

PRACTICAL-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 an 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 the LIFO (last in first out) principle as the element that was inserted last will be removed first. A stack can be implemented using an array as well as linked list. stack has three basic operations; push, pop, peek. The operations of adding & removing the elements is known as Push & Pop.

* Algorithm:

Step 1: Create a class stack with instance variables items

Step 2: Define the init method with self argument and initialize the initial value and then initialize to an empty list

>>> s.push(20)

>>> s.l

[20, 0, 0, 0, 0]

>>> s.pop()

Data = 20

>>> 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]

```
class stack :
    global top
    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:
            x = self.l[self.tos]
            print "Data =", x
            self.l[self.tos] = 0
            self.tos = self.tos - 1
```

step 3: Define methods push & pop under the class stack

step 4: Use of statement 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 statement as insert the element into the stack & 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 in pop method value is less than 1 then return the stack is empty or else delete the element from stack at top most position.

step 8: First condition check whether the no of elements are zero while the second case whether it is assigned any value. If top is not assigned any value then we can't say that stack is empty

22/6/22

PRACTICAL - 6

Aim :- Implementing a Queue using list

Theory :- Queue is a linear data structure which has 2 references front & rear. Implementing a queue using Python list is the simplest is the python list provides in built function to perform the specified operation of the queue It is based on the principal that a new element is inserted after rear & element of queue deleted which is at front. In simple terms a data 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 is moved to successive element * dequeue operation cannot remove the element in the queue is empty

```
roa:
class Queue:
    global f, r, a
    def __init__(self):
        self.f=0
        self.r=0
        self.a=[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()
```

8A.

Output

```
>>> b.enqueue(2)
    insert:2
>>> b.enqueue(4)
    insert:4
>>> b.enqueue(7)
    insert = 7
>>> b.q
  [2,4,7]
>>> b.dequeue()
    queue element deleted:2
>>> b.dequeue()
  queue element deleted:4
>>> b.dequeue()
    queue element deleted:7
>>> b.dequeue()
    Queue is empty
>>> b.q
  [0,0,0]
>>> b.enqueue(8)
    Queue is full.
```

Algorithm

Step 1: Define a class queue & assign global variables that defines init (i) method with self argument in init (i) assign & initialize the initial value with the help of self argument

Step 2:- Define an empty list & define enqueue () method with two argument. Assign the length of empty list

Step 3:- Use if statement that length is equal to rear than queue is full or else insert the element in empty list & display the queue element added successfully and increment by 1.

Step 4: Define dequeue () with self argument create rear, use statement that front is equal to length of list then display queue is empty or else. give that front is at zero and next that delete the element from front with & increment it by 1

Step 5: Now call the queue () function give the element that have to be added in the empty list by using enqueue () & print the list after adding and now for deleting & display the list after deleting the element from the list

PRACTICAL-7

Aim:- Program an environment of given ex by using stack in Python environ i.e Postfix

Theory: the postfix expression is free of any parantheis further we took care of the priorities of the operators in the program. A given postfix expression can easily be evaluated using stacks. Read the expression is always from left to right in postfix.

Algorithm:

1. Define evaluate as function then create a empty stack in Python.
2. convert the string to a list by using the string method split
3. Calculate the length of string & print it
4. Use for loop to access the range of string the give condition using if statement
5. Scan the token list from left to right. If token is an operand, convert it from a string to Integer & Push

```
def evaluate(s):
    t = s.split()
    n = len(t)
    stack = []
    for i in range(n):
        if t[i].isdigit():
            stack.append(int(t[i]))
        elif t[i] == 't':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) + int(a))
        elif t[i] == '-':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) - int(a))
        elif t[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 - *"
print("evaluated value is:", evaluate(s))
```

Output

Evaluated value is: 10

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

Perform the arithmetic operations. Push the result back on the 'm'

When the input expression has been completely processed the result is on the stack. Pop the p & return the value.

Print the result of stack after the evaluation of Postfix

Attach output & input of above algorithm

17/01/22

12 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 in a linear fashion. but no 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 larger list, all the elements if we add/remove any element from the list, all the elements of list has to adjust itself every time we add it is very tedious so linked list is used to solve this type of problems

* Algorithm:

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

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

Scanned with CamScanner

```
class node:
    def data, next

def __init__(self, item):
    self.data = item
    self.next = None

class linked list:
    globals
    def __init__(self):
        self.s = None
    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 addB(self, item):
        newnode = node(item)
        if self.s == None:
            self.s = newnode
        else:
            newnode.next = self.s
```

```
def display (self):
    head = self.1
    while head.next != None:
        print (head.data)
        head = head.next
    print (head.data)

n = linked list()
```

Output

```
>>> q.addL(40)
>>> q.addL(30)
>>> q.addL(20)
>>> q.addL(10)
>>> q.addB(60)
>>> q.addB(70)
>>> q.addB(80)
>>> q.addB(90)
>>> q.display()
```

the head pointer of the linked list

Step 3: Thus, the entire linked list can't be transversed using the node which is referred by the head pointer of the linked list

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

Step 5: We should not use the head pointer to transverse 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 easily track.

Step 6: We may lose the reference to the 1st node in our linked list, and hence most of our linked list so in order to avoid making same unwanted changes to the 1st node, we will a temporary node to transverse the entire linked list

Step 7: We will use this temporary node as a copy of the node we use currently transversing. Since we are making temporary node a copy of current node the data of the temporary node should also be

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 the 2nd nodes as n = n.next

Step 10: Similarly we can transverse rest of nodes in the linked list using same method by while loop

Step 11: Our concern now is to find terminating condition of the while loop

Step 12: The last node in the linked list is referred by the tail of linked list. Since the last node of linked it does not have any next node, the value in the next field of the last node is None.

Step 13: So we can refer the last node of the linked list self.next = None.

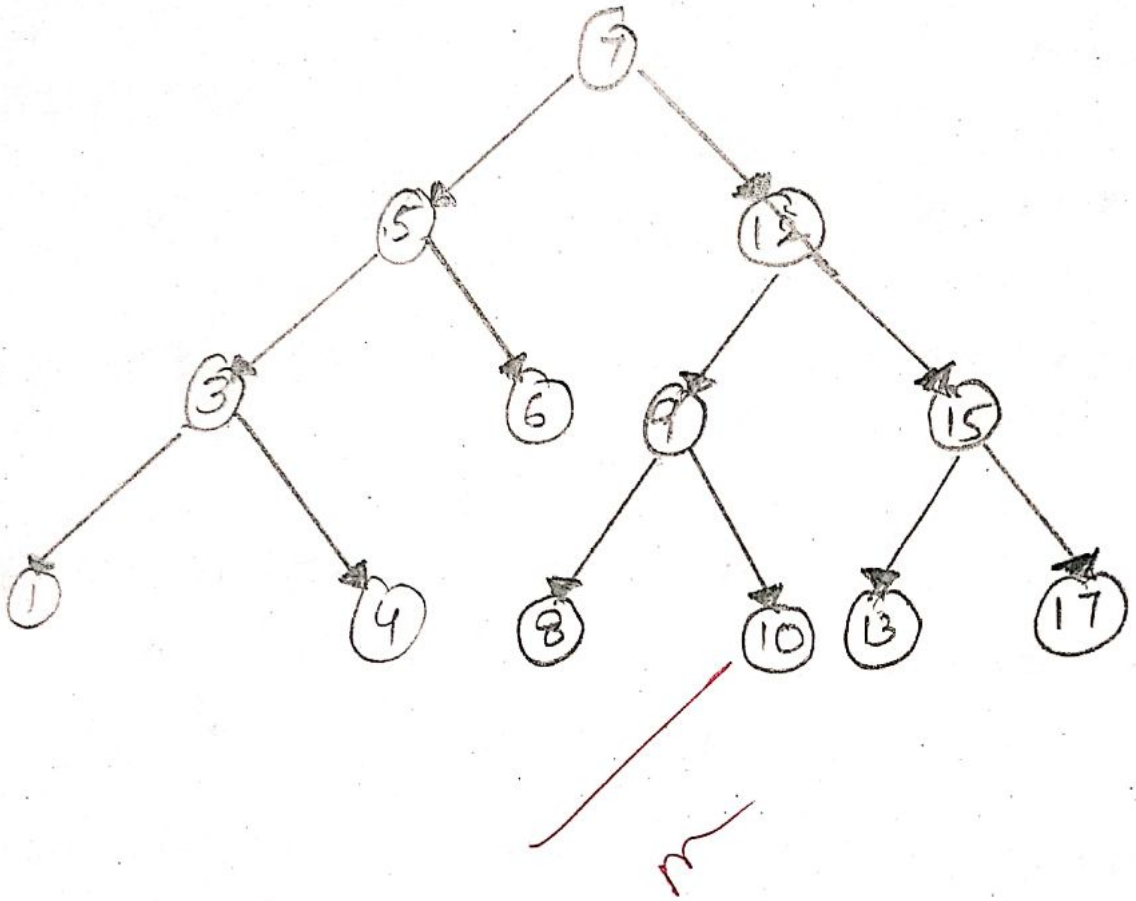
Step 14: We may have to now see how to start traversing the linked list & how to identify whether we have reached the last node of linked list or not.

Step 15: Finally the coding of input & output of the algorithm

90
80
70
60
40
30
20
10

24/07/2020

Binary search tree



PRACTICAL-9

• 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 & other as right child.

1) Inorder:- i) Traverse the left subtree, the left subtree inturn might have left and right subtrees
ii) Visit the root node
iii) Traverse the right subtree and repeat it.

2) Preorder: i) Visit the root node.
ii) Traverse the left subtree, the left subtree inturn might have left & right subtree
iii) Traverse the right subtree, repeat it.

3) Postorder: i) Traverse the left subtree. The left subtree inturn might have left & right subtrees
ii) Traverse the right subtree
iii) Visit the root node.

Algorithm

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

2. Again Define a class. Bst — that is binary search tree with init() method with self argument & assign the root is None

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

4. Use if statement for checking the condition the root is none then use else statement. If node is less than the main node then put an arrange that in left side.

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

6. Use if statement with then that else statement for checking that node is greater than main root then put it into

```
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
            while True:
                if p.val < h.val:
                    if h.left == None:
                        h.left = p
                        print(p.val, "Node is added to left side successfully at", h.val)
                        break
                    else:
                        h = h.left
                else:
                    if h.right == None:
                        h.right = p
                        print(p.val, "Node is added to right side successfully at", h.val)
                    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)
        Postorder(root.right)

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

output

```
t = BST()
t.add(7)
t.add(5)
t.add(12)
t.add(3)
t.add(6)
```

Binary search tree

Step 8: Define Inorder(), Preorder() and Postorder() with root argument and use if statement that root is none & return that as all.

Step 9: In Inorder, else statement used for giving that condition first left, root & then right node.

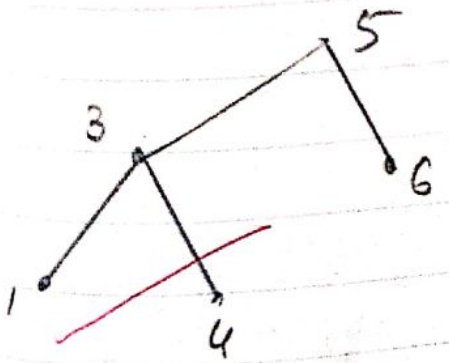
Step 10: For Preorder, we have to give condition in else that first root, left & then right node.

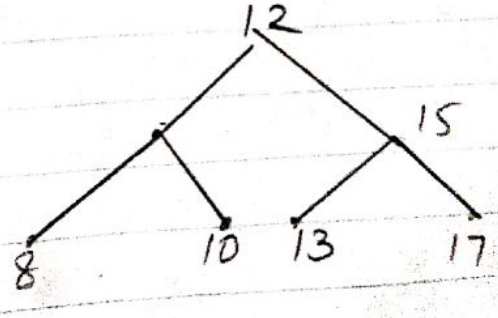
Step 11: For Postorder in else part, assign left then right and then go for the next node

Step 12: Display the output & input of above Algorithm

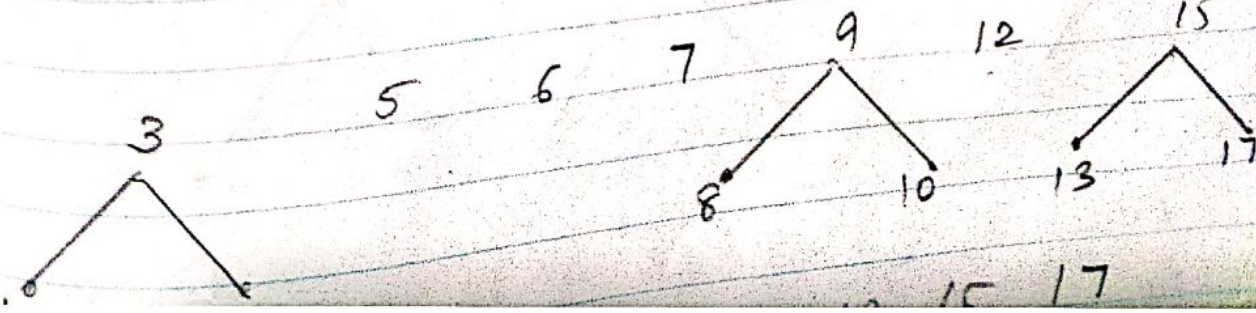
Inorder: (LVR)

Step 1:



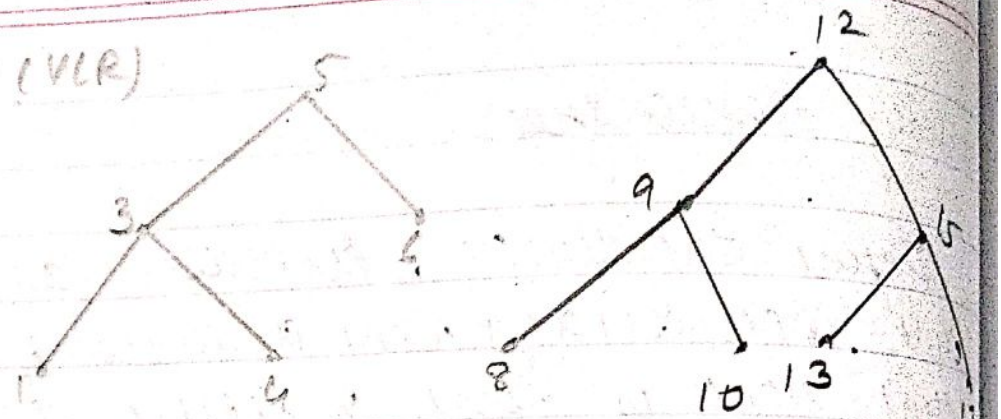


Step 2

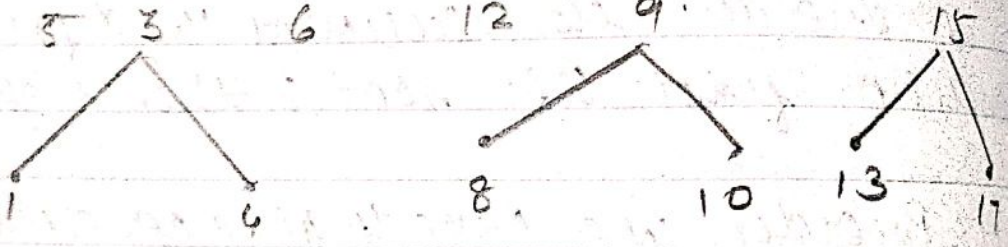


• Preorder : (VLR)

step 1:



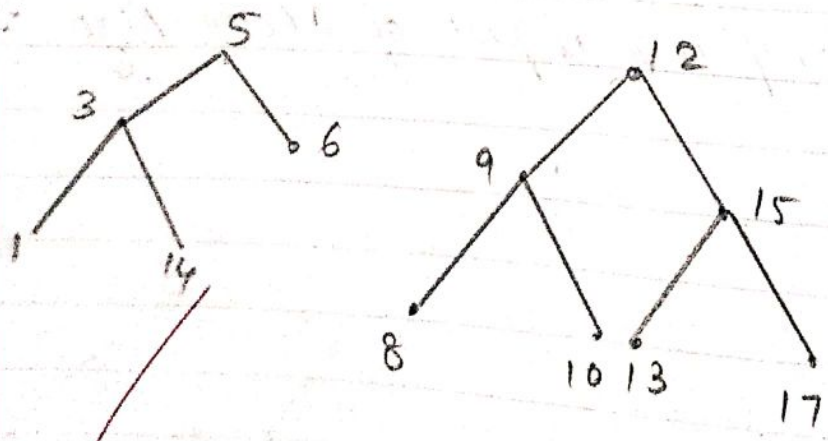
step 2: 7 5 3 6 12 9 15



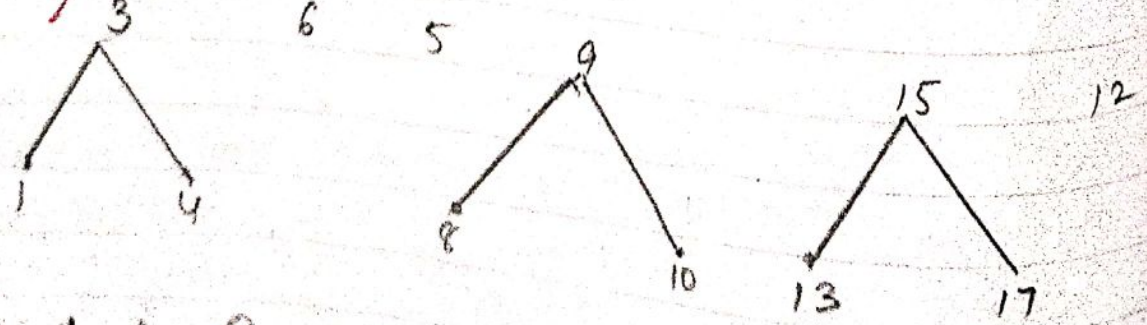
step 3: 7 5 3 1 4 6 12 9 8 10 15 13 17

• Postorder: (LRV)

step 1:



step 2



step 3: 1 4 3 6 5 8 10

```
t.add(9)
.add(5)
.add(1)
t.add(4)
t.add(8)
t.add(14)
t.add(13)
t.add(17)
int("\n Inorder form of Tree")
Inorder(t.root)
("\n Preorder form of Tree")
Preorder(t.root)
Print("\n Post Inorder form of Tree")
Postorder(t.root)
```

MV
07/02/2022

```
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)
```

PRACTICAL-10

AIM: To demonstrate the use of circular queue

Theory: In a linear queue, once the queue is completely full, it is not possible to insert more elements. Even if we dequeue the queue to remove some of the elements, until the queue is reset, no new 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 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 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. The end of circular queue, head pointer will always point to the front of the queue and tail pointer will always point to the end of the queue. Initially, the head & the tail points will point to the same location, this would mean that the queue is empty. New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point

to the next available location. Applications Bel
we have some common real world examples
where circular queues are used.

1. computer controlled traffic signal system
uses circular queues.
2. CPU scheduling and memory management.

```
if (self.f <= n-1):
print("Data removed:", self.l[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 = 1
```

```
q = Queue()
q.add(44)
q.add(55)
q.add(66)
q.add(77)
q.add(88)
q.add(99)
q.remove()
q.add(66)
q.d
```

data added: 88
queue is full
Data removed: 44

11a

```
def merge sort(arr):
    if len(arr)>1:
      mid = len(arr)//2
      lefthalf = arr[:mid]
      righthalf = arr[mid:]

    merge sort(lefthalf)
    merge sort(righthalf)

    i=j=k=0

    while(i<len(lefthalf) & 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

while j<len(righthalf):
    arr[k]=righthalf[j]
    j=j+1
    k=k+1

arr=[23, 9, 10, 55, 62, 99, 45, 14, 87
print("RANDOM LIST
```

28/02/2020.

PRACTICAL-11

AIM: To sort a list using Merge sort

Theory: like Quicksort, mergesort is a Divide & conquer algorithm. It divides input array in two halves, calls itself for the two halves & then merges the two sorted halves. The merge() function is used for merging two halves. The merge(a, p, q, m, r) is a process that assumes that a[p..m] and a[m+1..r] are sorted and merges the two sorted sub-arrays into one. The array is recursively divided in two halves till the size becomes 1. Once the size becomes 1 the merge processes come into action and starts merging arrays back till the complete array is merged

Applications:

1. merge sort is useful for sorting linked lists in O(nlogn) time. Merge sort accesses data subsequently 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 types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus is popular in languages like Lisp, where sequentially accessed data structures are very common.