1) Unsorted:

Algorithm:

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

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

Step3: Use for loop for adding the elements into 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 a range from '0' to the total no of elements to searched the elements from the list.

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

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

Unsorted program:


```
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 a number: 3
Element found at: 1

Enter a number: 10
No Element found.
```

#Sorted program:

```
s=list(input("Enter the numbers:"))
s.sort()
print(s)
a=int(input("Enter the No to be searched:"))
for i in range(len(s)):
    if(a==s[i]):
        print("Number Found in position:",i)
        break
else:
    print("No not found")
```

Output:

```
Enter the numbers: 2,5,8,9,1
[1,2,5,8,9]
Enter the No to be searched: 5
Number found in position 2

Enter the numbers: 2,5,9,3,1
[1,2,3,5,9]
Enter the No to be searched: 7
No not found.
```


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 the given algorithm.

2) Sorted linear search:

Sorting means arranging the elements in increasing or decreasing order.

Algorithm:

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

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

Step 3: Use for loop for using append() method to add the elements in the list.

Step 4: Use sort() method to sort the accepted element and assign in increasing order the list then print the list.

Step 5: Use if statement to give the range in which element is found in given range then display "Element not found".

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 case time complexity. It is a force approach. On the other hand incase 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.

Step 6: Then use the else statement if element is not found in range then satisfy the given condition.

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

Step 8: Use if loop that the elements in the list is equal to the element accepted from user.

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

Step 10: use another if loop to print that the element is not found if the element which is accepter from user is not their in the list.

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

29/11/19

———X———

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 a[m]:
            print("no found")
            break
        else:
            if s<a[m]:
                last=m-1
            else:
                first=m+1
```


Practical no: 2

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

Theory:

Binary search

Binary search is also known as the half interval search, logarithmic, search or binary chop is a search algorithm that finds an 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 to need to search entire list in linear search, which is time consuming. This can be avoided by using Binary Fashion search.

Algorithm:

Step1: Create Empty list and 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.

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

Step 5: Use if loop to give the range in which element is found in given range then display a message "Elements not found".

Step 6: Then use the else statement, if statement is not found in range then satisfy the below condition.

Step 7: Accept an argument ep key of the element that element has to be sorted.

Step 8: 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.

Step 9: Use for loop & assign the given range.

Step 10: If statement in list and still the element to be searched is not found then find the middle element (mi).

Step 11: Else if the item to be searched is still less than the middle term then

Initialize last(h) = mid(m) - 1
Else
Initialize first(l) = mid(m) - 1

Step 12: Repeat till you found the element stick the input to output of above algorithm.

Practical no: 3

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 simpliest 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:

Step 1: Bubble sort algorithm ~~stands~~ start for by comparing the first two elements of an array and swapping if necessary.

Step 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

Step 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.

Bubble sort:

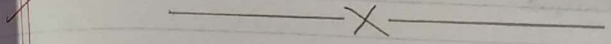
```
s = list(input("Enter list of nos:"))
for i in range(len(s)-1):
    for b in range(len(s)-1-i):
        if s[b] > s[b+1]:
            d = s[b]
            s[b] = s[b+1]
            s[b+1] = d
print(s)
```

Output:

```
Enter list of nos: 5,78,57,90,1000,15,1
[1,5,15,57,78,90,1000]
```

Step 5 : There are 'n' elements to be sorted then the process mentioned above should be repeated n-1 to get the required ~~sort~~ result.

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



CP

Practical no: 4

Aim: Implement quick sort to sort the given list.

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

Algorithm:

Step 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

Step 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.

Step 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.

44

```
# Quick sort :
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 not done:
        while l<=r and alist[l]<=pivot:
            l = l+1
        while alist[r]>=pivot and r>=1:
            r = r-1
        if r<1:
            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
x = int(input("Enter range for list:"))
alist = []
for b in range(0,x):
    b = int(input("Enter element:"))
```

```
    alist.append(b)
    n=len(alist)
quick(alist)
print(alist)
```

Output:

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


Step 4: we begin by incrementing leftmark until we locate a value that is greater that the P.V. we then decrement rightmark until we 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 eventual spilt point.

Step 5: At the point where rightmark becomes less than leftmark we stop. The position of rightmark is now the spilt point.

Step 6: The pivot value can be exchanged with the contents of spilt point and P.V (pivot value) is now in place.

Step 7: In addition, all the items to left of spilt point are less than PV & all the items to the right of spilt point are greater than PV. The list can now be divided at spilt point & quick sort can be involved recursively on the two halves.

Step 8: The quick sort function involves a recursive function, quicksorthelper.

Step 9: quicksorthelper begins with same base as the merge sort.

Step 10: If length of the list is less than 0 equal, one it is already sorted.

Step 11: If it is greater, then it can be ~~partitioned~~ partitioned and recursively sorted.

Step 12: The partition function implements the process described earlier.

Step 13: Display and stick the coding and output of above algorithm.

———X———

MM
20/12/19

Stack:

```
print("Akhil Pillai")
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 Empty")
        else:
            k=self.l[self.tos]
            print("data", k)
            self.l[self.tos]=0
            self.tos=self.tos-1
s=stack()
    def peek(self):
        if self.tos<0:
            print("stack empty")
        else:
            a=self.l[self.tos]
            print("data=", a)
s=stack()
```


Practical no: 5

Aim: Implementation of stack using python list.

Theory: A stack is a linear data structure that can be represented as in the real world in the form of a physical stack or a pile. The elements in the stack are added or removed from only 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 array as well as linked list. Stack has three basic operations of push, pop & peek. The operations of adding & removing the elements is known as 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.

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

Step 4: use if statement to give the condition that if length of given list is greater than the range

of list then print stack is full.

Step 5: Or Else print statement as insert the element into the stack and initialize the value.

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 topmost position.

Step 8: First condition checks whether the no of elements are zero while the second case whether top is assigned any value. If top is not assigned any value, then we can sure that stack is empty.

Step 9: Assign the element values in push method to add and print the given value is poped or not.

Step 10: Attach the input and output of above algorithm.

—————X—————

Output:

```
Akhil Pillai
>>> S.push(10)
>>> S.push(20)
>>> S.push(30)
>>> S.push(40)
>>> S.push(50)
>>> S.push(60)
stack is full
```

S.pop()
S.peek()

03/01/2020

#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: 6

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 ~~format~~ front. In simple terms, a queue can be described as a data structure based on first in first out (FIFO) principle.

- Queue() - Create a new empty queue.
- Enqueue() - Insert an element at the rear of the queue & 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. If dequeue operation cannot remove element if the queue is empty.

Algorithm:

Step 1: Define a class queue and assign global variables then define init (method) with a argument in init(), assign or initialize the value 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 statement that length is equal to res, then queue is full or else ~~queue~~ insert the ~~queue~~ elements in ~~que~~ empty list or display that queue element added successfully and increment by 1.

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

Step 5: Now call the Queue() 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.

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 elemented inserted', 7)
>>> b.enqueue(8)
('Queue element inserted', 8)
>>> b.enqueue(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]
```

10/01/24

#postfix:

```
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))
        elif k[i] == '/':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b)/int(a))
    return stack.pop()
s = "6 9 * +"
r = evaluate(s)
print("The value is:", r)
```


Practical no: 7

Aim: Program on Evaluation of given string by using stack in python Environment i.e postfix

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:

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 and print it.

Step 4: use for loop to assign the range of string then give condition using if statement.

Step 5: 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'.

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 'n'.

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 & input of above algorithm.

———————X———————

output:
('The value is:', 62)

17/01/2

linked list:

```
class node:
    global data
    global next
    def __init__(self, item):
        self.data = item
        self.next = None
class linkedlist:
    global s
    def __init__(self):
        self.s = None
    def addL(self, item):
        newnode = node(item)
        if self.s == None:
            self.s = newnode
        else:
            head = self.s
            while head.next != None:
                head = head.next
            head.next = newnode
    def addB(self, item):
        newnode = node(item)
        if self.s == None:
            self.s = newnode
        else:
            newnode.next = self.s
            self.s = newnode
    def display(self):
        if self.s == None:
            print("List is empty")
        else:
            head = self.s
```


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, 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 task so linked list is used to solving this type of problems.

Algorithm:

Step 1: Transversing of a linked list means visiting 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 us. The first node of the linked list. The first node of the linked list in turn is referred by head pointer of the linked list.

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

Step 4: 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 list only.

Step 5: We should not use the head pointer to traverse the entire 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~~ changes which we cannot revert back.

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 some unwanted changes to the 1st node, we will use a temporary ~~node~~ 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 datatype of the temporary node should also be Node.


```
while head = self.s
    while head.next != None:
        print(head.data)
        head = head.next
    print(head.data)
```

```
q = linked list()
```

output:

```
>>> q.addB(10)
>>> q.addB(20)
>>> q.addB(30)
>>> q.addB(40)
>>> q.addL(50)
>>> q.addL(60)
>>> q.addL(70)
>>> q.addL(80)
>>> q.display()
40
30
20
10
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= h.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 ~~refe~~ to find terminating condition for 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 list 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 linked list self.s = None.

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

Step 15 : Attach the coding and output of above algorithm.

24/01/2020

#Binary Tree:

```
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 leftside successfully at", h.val)
                        break
                    else:
                        h = h.left
                else:
                    if h.right == None:
                        h.right = p
                        print(p.val, "Node is added to rightside successfully", h.val)
```


Practical no: 9

Aim: Program based on Binary search Tree by implementing Inorder, Preorder & Postorder.

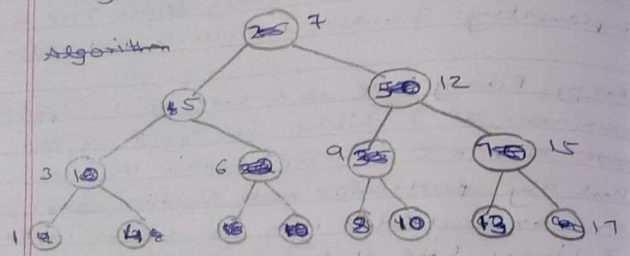
Theory: Binary Tree is a tree which supports maximum of 2 childsen for any node ~~can have either 0 or 1 or 2~~ within the tree. Thus any particular node can have either 0 or 1 or 2 childsen ~~there~~ it is another identity of binary tree that is ordered such that one child is identified as left child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

Inorder: 1) Transverse the left subtree. The left subtree inturn might have left and right subtrees.
2) Visit the root node.
3) Transverse the right subtree and repeat it.

Preorder: 1) Visit the root node.
2) Transverse the left subtree. The left subtree in inturn might have left & right subtree.
3) Transverse the right subtree repeat it.

Postorder: 1) Transverse the left subtree. The left subtree inturn might have left and right subtrees.
2) Transverse the right subtrees.
3) Visit the root node.

Binary search Tree.



Algorithm:

Step 1: Define class node and define init() method with 2 argument. Initialize 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 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 root is none then use else statement for if node is less than the main node then put or arrange that in leftside.

```
            break
        else:
            h = h.right

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

step 5 : Use while loop for checking the node if 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 right side.

step 7 : After this left subtree and right subtree. repeat this method to arrange the node according to binary search Tree.

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

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

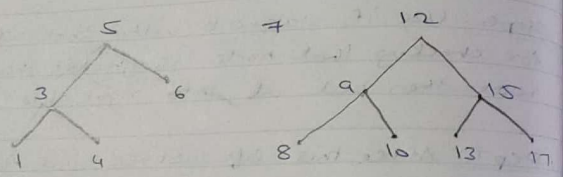
step 10 : For Preorder we have to give condition in else that first root, left & then right node.

step 11 : For Postorder In else part assign left then right and then go for root node.

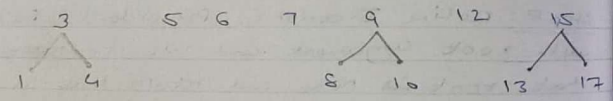
step 12 : Display the output & input of above algorithm.

Inorder : (LVR)

Step 1:



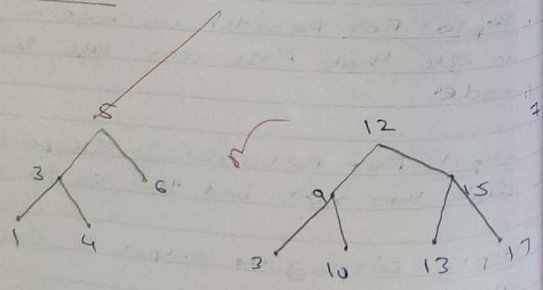
Step 2:



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

Post Preorder : (VLR)

Step 1:



Output:

```
>>> t = BST()
>>> t.add(7)
('Root is added successfully', 7)
>>> t.add(5)
('Root is added successfully', 5)
>>> t.add(12)
('Root is added successfully,' 12)
>>> t.add(3)
('Root is added successfully', 3)
>>> t.add(6)
('Root is added successfully', 6)
>>> t.add(9)
('Root is added successfully,' 9)
>>> t.add(15)
('Root is added successfully', 15)
>>> t.add(1)
('Root is added successfully', 1)
>>> t.add(4)
('Root is added successfully,' 4)
>>> t.add(8)
('Root is added successfully,' 8)
>>> t.add(10)
('Root is added successfully', 10)
>>> t.add(13)
('Root is added successfully', 13)
>>> t.add(17)
('Root is added successfully', 17)
```

>>> Inorder(t.root)

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

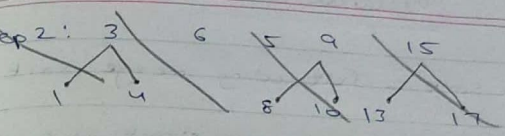
>>> Preorder(t.root)

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

>>> Postorder(t.root)

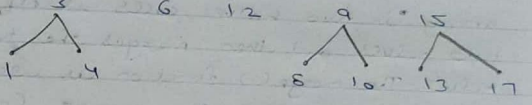
1 4 3 6 5 8 10 9 13 17 15 12 7

Step 2: 3 6 5 9 15 12 17



Step 2:

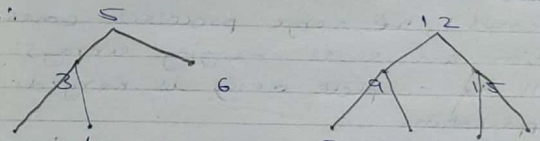
7 5 3 6 12 9 15



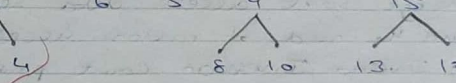
Step 3: 7 5 3 1 4 6 12 9 8 10 15 13 17.

- Postorder: (LRV)

Step 1:



Step 2: 3 6 5 9 15 12 7



Step 3:

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

Practical no:10.

Aim: To sort a list using merge sort.

Theory: Like Quicksort, Mergesort is a divide and conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges 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 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 comes into action and starts merging arrays back till the complete array is merged.

Applications:

- Merge sort is useful for sorting linked list in O(n log n) tion.
Merge sort access data sequentially and the need of random access is low
2. Inversion Inversion count problem.
3. Used in External sorting
Merge sort is more efficient than quick sort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular where sequentially accessed data structure are very common.

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

Outputs:

('Random List:', [27, 89, 70, 55, 62, 99, 45, 14, 10])

('In Mergesort list:', [10, 14, 27, 45, 55, 62, 70, 89, 99])

practical no: 11

Aim: To demonstrate the use of circular ~~array~~ 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 option, it again starts from the first position after the last, hence making the queue behave like a circular queue, structure. In case of a 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.

```
class Queue():
    global r
    global f
    def __init__(self):
        self.r=0
        self.f=0
        self.l=[0,0,0,0,0,0,0]
    def add(self,data):
        n=len(self.l)
        if(self.r<n-1):
            self.l[self.f]=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:
            s=self.f
            self.f=0
```

```
        if (self.f< self.r):
            self.l[self.f]=0
            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(10)
    data added:10
>>> q.add(20)
    data added: 20
>>> q.add(30)
    data added: 30
>>> q.l
    [10, 20, 30, 0, 0, 0]
>>> q.remove()
    [0, 20, 30, 0, 0, 0]
```

14/02/2020

Initially the head and the tail pointer will be pointing to the same location this would be 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:

Below we have some common real-world examples where circular queue are used:

1. Computer controlled Traffic signal system uses circular queue
2. CPU scheduling and memory management.

———————X———————