PRACTICAL No: 1

LINEAR SEARCH

Sorted data

Algorithm

Step 1: Store the list of element in a variable and perform sort method on the variable.

Step 2: Take input from the user for the number to be searched

Step 3: Use for loop to check for the number and use if conditional statement to check if the element is same as the input number and if found print the position of the element. If not found then print the respective output.

08. Code

```
a=[4, 3, 5, 8, 9, 2]
a.sort()
S= int(input("Enter the element to be searched:"))
for i in range(len(a)):
        if (S==a[i]):
            print("Element found at ", i)
            break
if(S!=a[i]):
    print("Element not found")
```

Output

```
>>>
Enter the element to be searched: 5
Element found at 3
>>>
Enter the element to be searched: 0
Element Not found.
```

18.

Unsorted

Algorithm

Step 1:- Take 2 input from the user first of list of element and then the number of to be searched.

Step 2:- Use for to check for the number use if conditional statement to check if the element is same as the input number and if found print the position of the element

Code

```
a=list(input("Enter the list of elements:"))
print("list of element are:", a)
s=int(input("Enter the element to be searched:"))
for i in range(len(a)):
    if(s==i):
        print("Element found at", i)
        break
if(s!=a[i]):
    print("Element not found")
```

Output

```
>>>
Enter the list of element: 3,5,7,9,0
('list of element are:', [3,5,7,9,0])
Enter the element to be ~~four~~ searched: 5
('Element found at', 1)
>>>
Enter the list of element: 5,8,9,0,4
('list of element are:', [5,8,9,0,4])
Enter the element to be searched: 2
Element not found.
```

29/11/19

PRACTICAL NO. 2
BINARY SEARCH

AIM:- Implement Binary Search to find an searched no in the list.

THEORY:- Binary search is also known as Half interval search, logarithmic search or binary chop it 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:-

Step1: Take an input of element from the user as list datatype.

Step2:- Sort the element entered by user by sort() and take input of element to be searched.

Step3:- Use if conditional statement to check if the number is in the list if not then display the output as element not found.

i.e.

Step 4:- In else block use the logic to check if the number is present in the list of element.

Step 5:- print the output as element found.

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: 3,9,2
Enter the number to be searched: 0
Element not found
```

PRACTICAL NO:- 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:-

Step1:- Declare an empty list and then take an input of the ~~no~~ number of element.

Step2:- Use the for loop and then take input of the number of element in the list.

Step3:- Use the for loop an if the element is smaller than second then we do not swap the element.

Step4: 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.

Step5: There are n elements to be sorted then the the process mentioned above should be repeated n-1 to get the required result.

Step6: Stick the output of and input of above algorithm of bubble sort elsewise.

18. Code

```
a=[]
x = int(input("Enter the number of element: "))
for i in range(0, x):
    i = int(input("Enter element:"))
    a.append(i)
    print(a)
for j in range(0, len(a)):
    for k in range(len(a)-1):
        if a[j] < a[k]:
            tmp = a[k]
            a[k] = a[j]
            a[j] = tmp
print("Element after sort are: ",a)
```

```
>>> Enter the number of element: 5
Enter element: 1
[1]
Enter element: 7
[1, 7]
Enter element: 5
[1, 7, 5]
Enter element: 6
[1, 7, 5, 6]
Enter element: 3
[1, 7, 5, 6, 3]
Element after sort are: [1, 3, 5, 6, 7]
```

PRACTICAL NO: 4

QUICK SORT

AIM:- Implement Quick sort to sort the given list.

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

ALGORITHM:-

Stp1:- 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.

Stp2:- 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.

Stp3:- Partitioning begins by locating two position markers lets call them leftmark and 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.

Step4: We begin by increasing leftmark until we locate a value that is greater than the p.v. We then decrement ~~rightmark~~ rightmark until we find value that is less than the p.v. At this point we have discovered two items that are out of place with respect to eventual split point.

Step5: At the point where rightmark becomes less than leftmark we stop. The position of rightmark is now the split point.

Step6: The p.v can be exchanged with the content of split point and p.v is now in place.

Step7: In addition, all the items to left of split point are less than p.v and all the items to the right of split point are greater than p.v. The list can now be divided at split point and quick sort can be invoked recursively on the two halves.

Step8: The quicksort function invokes a recursive function, quicksorthelper.

Step9: quicksorthelper begins with same base as the merge sort.

Step10: If length of the list is less than 0 or equal one list is already sorted.

Step11: If it is greater than it can be partitioned and recursively sorted.

Step12: The partition function, implement the process described earlier.

Step13: Display and stick the coding and output of above algorithm.

Code

```
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)
```

```
>>>
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]
```

21/12/19

PRACTICAL No:- 5

AIM:- Implementation of stack using Python

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:-

Step1: Create a class Stack with instance variable items

Step2: Define the init method with self argument and initialize the initial value and then initialize to an empty list.

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 element into the stack and initialize the value.

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

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

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

Step9: Assign the element values in push method to add and print the value is

Step10: Attach the input and output of above algorithm

Code

```
# Stack
class Stack:
    global tos
    def __init__(self):
        self.l = [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
            print(self.l)
    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
            print(self.l)
    def peek(self):
        if self.tos < 0:
            print("Stack is empty")
```

```
        else:
            a = self.l[self.tos]
            print "Data = ", a
s = stack()
```

Output

```
>> s.push(10)
   [10, 0, 0, 0]
>> s.push(20)
   [10, 20, 0, 0]
>>> s.push(30)
   [10, 20, 30, 0]
>>> s.push(40)
   [10, 20, 30, 40]
>>> s.push(50)
   stack is full
>>> s.peek()
   Data = 40
>>> s.pop()
   Data = 40
   [10, 20, 30, 0]
>>> s.pop()
   Data = 30
   [10, 20, 0, 0]
>> s.pop()
   Data = 20
   [10, 0, 0, 0]
>>> s.pop()
   Data = 10
   [0, 0, 0, 0]
>>> s.pop()
   stack is empty
>>> s.peek()
   stack is empty
```

03/01/2022

PRACTICAL NO: 6

AIM:- Implementing a Queue Using Python list.

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

Queue(): create 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 dequeue operation cannot remove element if the queue is empty.

ALGORITHM:

Step1: Define a class Queue and assign global variables then define init() method with self argument, assign or initialize the init value with the

help of self argument.

Step2: Define a empty list and define enqueue() method with 2 argument, assign the length of empty list.

Step3: Use if statement that length is equal to rear then queue is full or else insert the element in empty list or display the queue element added successfully and increment by 1.

Step4: Define dequeue() with self argument under this use if statement the element is less then the length then is given that front is at zero delete the front element and else display that the queue is empty.

Step5: Now call the queue() function and give the element that has to be added by using enqueue() and print the list after adding and same for deleting and display the list after deleting the element.

Code

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

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

10/01/22

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 stacks. Reading the expression is always from left to right in Postfix.

ALGORITHM:

Step1: Define evaluate as function then create a empty stack.

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 given condition using if statement.

Step5: Scan the token list from left to right. If token is an operand convert it from a string to an integer and push the value onto the 'p'.

Step6: 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.

Step7: Perform the arithmetic operation. Push the result back on the 'm'.

Step8: When the input expression has been completely processed the result is on the stack. Pop the 'p' and return the value.

Step9: Print the result of string after the evaluation of Postfix.

Step10: Attach output and input of above algorithm.

Code

```
def evaluate(s):
    k = s.split()
    n = len(k)
    stack = []
    for i in range(n):
        if k[i].isdigit():
            stack.append(int(k[i]))
        elif k[i] == '+':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) + int(a))
        elif k[i] == '-':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) - int(a))
        elif k[i] == '*':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) * int(a))
        else:
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) / int(a))
    return stack.pop()
s = "8 6 9 * +"
r = evaluate(s)
print "The evaluated value is:", r
```

Output

>>> The evaluated value is: 62

17/01/24

PRACTICAL NO: 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 not necessarily continuous. The individual element of the linked list called a Node. Node comprises of 2 parts 1) Data 2) Next. Data stores all the information with respect to the element for example roll no name, address, etc whereas next refers to next node. In case of larger list, if we add/remove any element from the list, all the elements of the 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:-

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

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

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

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.

5. 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 to traverse the entire linked list.

6. 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

7. 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.

Code

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

8. But the 1st node is referred by current. So we can traverse to 2nd nodes as h = h.next.

9. Similary we can traverse rest of nodes in the linked list using same method by while loop.

10. Our concern now is to find terminating condition for the while loop.

11. 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.

12. So we can refer the last node of linked list as self.s = None

13. We have to now see how to start traversing the linked list and how to identify whether we have reached the last node of linked list or not.

14. Attach the coding or input and output of above algorithm.

Output

```
>>> S.addL(50)
>>> S.addL(60)
>>> S.addL(70)
>>> S.addL(80)
>>> S.addB(40)
>>> S.addB(30)
>>> S.addB(20)
>>> S.display()
>>
20
30
40
50
60
70
80
```

24/01/22

PRACTICAL No:- 9

BINARY SEARCH TREE

AIM:- Program based on binary search tree by implementing Inorder, Preorder and 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 either 0 or 1 or 2 children. There is another identity of binary tree that it is ordered such that one child is identified as left child and other as right child.

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

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

Postorder:- i) Traverse the left subtree. The left subtree intern might have left and right subtree.
ii) Traverse the right subtree.
iii) Visit the root node.

Algorithm:

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

Again, Define a class BST that is Binary Search Tree with init() with self argument and assign the root is None.

Define add() for adding the node. Define Initialize variable P that has node value.

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.

Use while loop for checking the node is less than or greater than the main node and break the if loop if it is not satisfied.

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

After this left subtree and right subtree, repeat this method to arrange the node according to binary Search Tree.

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

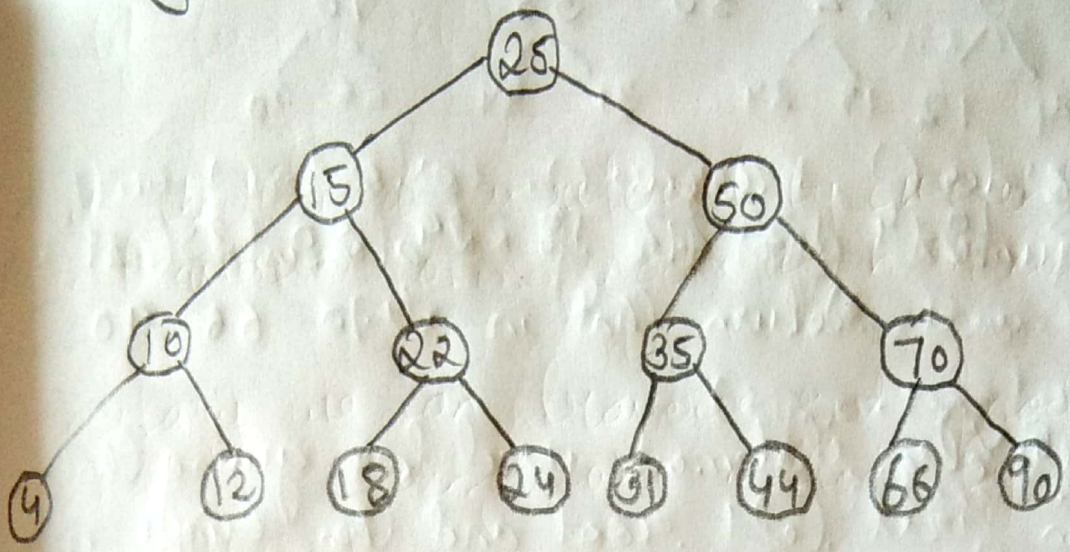
9:- In inorder else statement used for giving that condition first left, root and then right node.

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

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

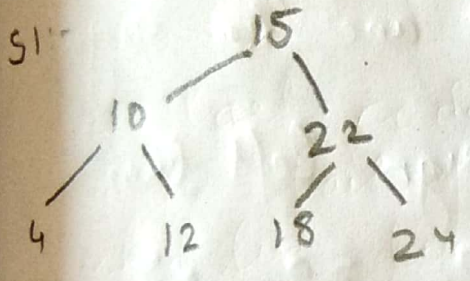
12:- Display the output and input of above algorithm.

Binary Search Tree

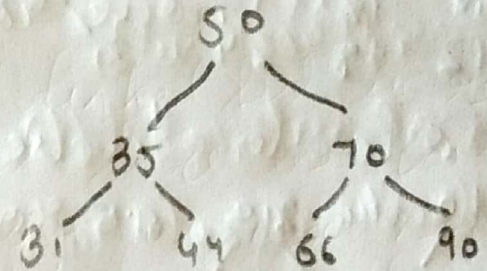


Inorder: (LVR)

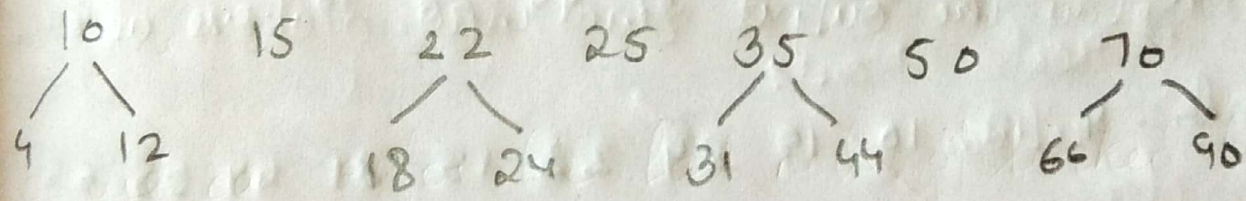
S1:



25



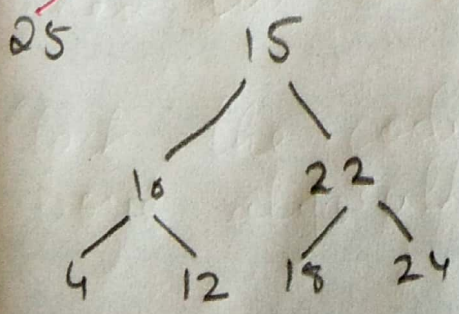
S2:

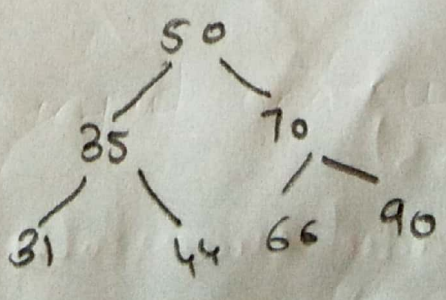


Inorder: 4 10 12 15 18 22 24 25 31 35 44 50 66 70 90

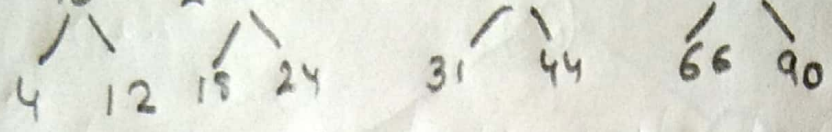
Preorder: (VLR)

S1:





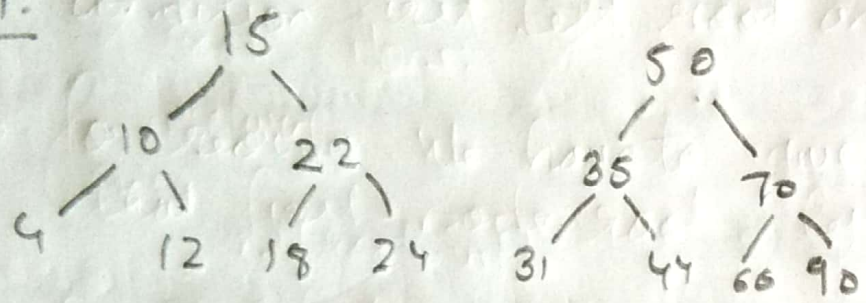
S2:. 25 15 10 22 50 35 70



Preorder 25 15 10 4 12 22 18 24 50 35 31 44 70 66 90

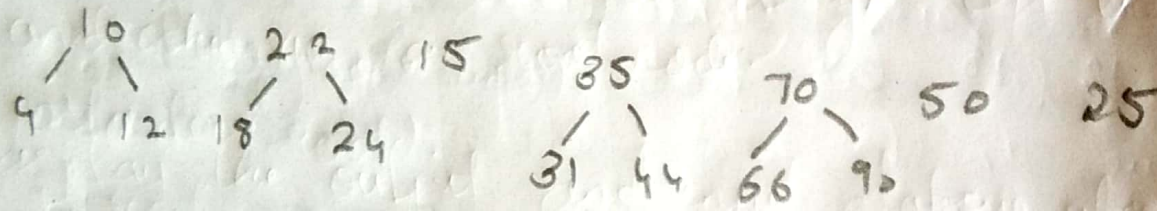
Postorder (LRV)

S1:



25

S2:



50 25

Postorder:. 4 12 10 18 24 22 15 31 44 35 66 90 70 50 25

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

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

t = BST()
```

Output:

```
>>> t.add(25)
Root is added successfully 25
>>> t.add(15)
15 node is is added to left side at 25
```

```
>>> t.add(50)
50 added to rightside at 25
>>> t.add(10)
10 Node is added to leftside at 15
>>> t.add(22)
22 Node is added to rightside at 15
>>> t.add(35)
35 Node is added to leftside at 50
>>> t.add(70)
70 Node is added to rightside at 50
>>> t.add(4)
4 Node is added to leftside at 10
>>> t.add(12)
12 Node is added to rightside at 10
>>> t.add(18)
18 Node is added to leftside at 22
>>> t.add(24)
24 Node is added to rightside at 22
>>> t.add(31)
31 Node is added to leftside at 35
>>> t.add(44)
44 Node is added to rightside at 35
>>> t.add(66)
66 Node is added to leftside at 70
>>> t.add(90)
90 Node is added to rightside at 70
```

>>> inorder(t.root)

4
10
12
15
18
22
24
25
31
35
44
50
66
70
90

> preorder(t.root)

25
15
10
4
12
22
18
24
50
35
31
44
70
66
90

> postorder(t.root)

4
12
10
18
24
22
15
31
44
35
66
90
70
50
25

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 halfs.

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-array into one. The array is recursively divided in two half till the size becomes 1.

Once the size becomes 1, the merge process comes into action and starts merging back the array.

Applications:-

1. Merge Sort is useful for sorting linked list in O(n log n).
   Merge Sort accesses data sequentially and the need of ~~se~~ random access is low.
2. Inversion Count Problem
3. Used in External Sorting.

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

Code:

```
def mergesort(arr):
    if len(arr)>1:
        mid = len(arr)//2
        lefthalf = arr[:mid]
        righthalf = arr[mid:]
        mergesort(lefthalf)
        mergesort(righthalf)
        i = j = k = 0
        while i< len(lefthalf) and j < len(righthalf):
            if lefthalf[i]< righthalf[j]:
                arr[k] = lefthalf[i]
                i = i+1
            else:
                arr[k] = righthalf[j]
                j = j+1
            k = k+1
        while i < len(lefthalf):
            arr[k] = lefthalf[i]
            i = i+1
            k = k+1
        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("Mergesort list:", arr)
```

18

Output

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

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

PRACTICAL No: 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, 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 element because the rear pointer is still at the end of the queue. The only way is to reset the linear queue for a fresh start.

Circular queue is also a linear data 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 data structure. In case 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.

Code:

```
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:
            self.r = n
            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:
                print(self.l[self.f])
                self.f = self.f + 1
```

```
        else:
            print("Queue is empty")
            self.f=s

q = queue()

>>> q.add(11)
Data added: 11
>>>q.add(22)
Data added: 22
>>> q.add(33)
Data added: 33
>>> q.add(44)
Data added: 44
>>> q.add(55)
Data added: 55
>>> q.add(66)
Data added: 66
>>> q.add(77)
Queue is full
>>> q.remove()
>Data removed: 11
>>> q.remove()
Data removed: 22
>>> q.l
[0, 0, 33, 44, 55, 66]
```

MP
14-2-2020