Aim: Implement binary search to find an item in the list.

Theory:

Linear Search

Linear search is one of the simplest searching algorithm in which targeted item is sequentially matched with each item in the list.

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

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

1] Unsorted:

Algorithm:

(i) Create an empty list and assign it to a variable.
(ii) Accept the total no. of elements to be inserted into the list from the user, say 'n'.
(iii) Use for loop for adding the element into the list.
(iv) Print the new list.
(v) Accept an element from the user that to be searched in the list.
(vi) Use for loop in a range from '0' to the total no. of elements to search the element from the list.
(vii) Use if loop that the elements in the list is equal to the element accepted from the user
(viii) If the element found then print the statement that the element found else along with the elements position.
(ix) Use another if loop to print that the element is not found, if the element which is from user is not there in the list.
(x) Draw the output of given algorithm.

Code:

```
a = [4, 3, 5, 8, 9, 2]
s = int(input("Enter the number to be searched:"))
for i in range(len(a)):
    if(s == a[i]):
        print("Number found at position", i)
        break
if(s != a[i]):
    print("No number found")
```

034

Output:

Enter number to be searched: 9
Number found at position 4

Enter number to be searched: 10
No number found.

Code:

```
a = list(input("Enter the number:"))
a.sort()
print a
s = int(input("Enter number to be searched:"))
for i in range(len(a)):
 if (s==a[i]):
     print("Number found at position", i)
     break
 if (s!=a[i]):
     print("No number found")
```

Output:

```
Enter the number: 2,4,6,8,10,12,14
[2,4,6,8,10,12,14]
Enter number to be searched: 10
Number found at position 4.

Enter the number: 2,4,6,8,10,12,14
[2,4,6,8,10,12,14]
Enter number to be searched: 16
No number found.
```

② Sorted Linear Search:

Sorting means to arrange the element in increasing or decreasing order.

Algorithm:

(i) Create empty list & assign it to a variable.
(ii) Accept total no. of elements to be inserted into the list from user, say 'n'
(iii) Use for loop for using append() method to add the elements in the list
(iv) Use sort() method to sort the accepted element and assign in increasing order the list, then print the list.
(v) Use If statement to give the range in which element is not found in given range, then display "Element not found" Then use else statement, if element is not found in range, then satisfy the given condition.
(vi) Use for loop in range from 0 to the total no. of elements to be searched, before doing this accept a search no. from user using input statement.
(vii) Use if loop that the elements in the lists is equal to the element accepted from user.

(ix) If the element is found then print the statement that the element is found along with the element position.

(x) 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.

(xi) Attach the input & output of above algorithm.

Source codes

```
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 (" number found")
                break
            else:
                if s<a [m]
                    last= m-1
                else:
                    first= m+1
```

Practical 2

Aim: Implement Binary Search to find user 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're 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 fusion search.

Algorithm:

1. Create empty list and assign it to a variable.
2. Using input method, accept the range of given list.
3. Use for loop, odd elements in list using append() method.
4. Use sort() method to sort the accepted element & assign it in increasing ordered list. print the list after sorting.

5. use if loop to give the range in which element is present in given range, then display a message "Element not found".

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

7. Accept one argument as key of the element that element has to be searched.

8. Initialise first to 0 & last to last element of the list as array is starting from 0. hence it is initialized 1 less than the total count.

9. use for loop & assign the given range.

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

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

12. Repeat till you found the element. stick the input & output of above algorithm

Source code:

```
a=list(input("Enter the numbers:"))
for p in range(len(a)-1):
    for c in range(len(a)-1-p):
        if a[c]>a[c+1]:
            t=a[c]
            a[c]=a[c+1]
            a[c+1]=t
print(a)
```

Output:

```
Enter the numbers: 50,1000,659,478,5000,400
[50, 400, 478, 659, 1000, 5000]
```

Practical-3

Bubble sort

Aim: Implementation of bubble sort program on given list.

Theory: Bubblesort 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 elements in ascending or descending order by comparing two adjacent elements at a time.

Algorithm:

1. Bubblesort 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 i we need to swap the element
3. If the element is smaller than second, then we do not swap the element

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

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

6. Write the output of input of above algorithm of bubble sort stepwise.

Source 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 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)
```

Practical-4

Quick sort

Aim: Implement Quick sort to sort the given list

Theory: The quicksort is a recursive algorithm based on divide and conquer technique.

Algorithm:

1. Quicksort 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.

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 with respect to pivot value while also converging on split point.

4. We begin by incrementing leftmark until we locate a value that is greater than is greater than the P.V. we then decrement rightmark until we find value that is less than the pivot value. At this point we have almost covered two items that are out of place with respect to eventual split point.
5. At the point where rightmark become less
8. 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 place.
7. In addition, all the items to left of split point are less than PV & all the items to the right at split point are greater than PV. 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. Quicksorthelper begins with same base thing as the merge sort.
10. If length of the list is 1 or less than equal, or, it is already sorted.
11. If it is greater, then it can be partitioned & recursively sorted.

```
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]
```

20/12/19

a. the partition function implement the process described earlier.

b. Display and stick the coding 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 a physical stack or pile. The elements in the stack are added or removed only from one position i.e the topmost position. Thus the stack works in LIFO (Last In First Out) principle as the elements that were inserted last will be removed first. A stack can be implemented using an array as well as linked list. It has three basic operation : push, pop, peek. The operations of adding & removing the elements is known as Push & Pop.

Algorithm:

1. Create a class stack with instance variable items.
2. Define the init method with self argument and initialize the initial value and then initialize to an empty list.

Source code:

C44

```
print("Vivek Tiwari")
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]
            self.tos = sdf.tos - 1
s = stack()
    def peek(self):
        if self.tos < 0:
            print("stack Empty")
        else:
            a = self.l[self.tos]
        print("data:", a)
```

Output:

Vivek Tiwari

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

03/04/2021

8 Defing methods push & pop under the class stack.

14 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.

5 Or Else print statement as insert the element into the stack & initialize the value.

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

: 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.

19 Assign the element values in push method to an and print the given value is popped, not.

1 First condition checks whether the no. of elements are zero, while the second case, whether top is assigned any value.

LK	5	12
Pop		10

If tos is not assigned any value, then you can be sure that stack is empty.

10. Attach the input & output of above algorithm.

Source code:

```
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 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-6

Aim: Implementing a Queue using Python list.

Theory: Queue is a linear data structure which has 2 references, front & 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 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()**: Returns 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. A dequeue operation cannot remove element if the queue is empty.

- **Algorithm:**

Step 1: Define a class Queue & assign global variables. Then define init() method with self argument in init(), assign or initialize the init value with the help of self argument.

Step 2: Define a empty list & define enQueue() method with 2 argument, assign the length of empty list.

Step 3: use if statement that length is equal to rear then Queue is full or else insert the element in empty list or display that Queue element added successfully & 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 delete the element from front side & increment it by 1.

Step 5: Now call the Queue() function & give the element that has to be added in the empty list by using enQueue() & print the list after adding & 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 element 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]
```

Source 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 = "5 8 3 - +"
r = evaluate(s)
print("The Evaluated value is:", r)
```

Practical-7

Aim: Program on Evaluation of given string by using stack in Python environment ie 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:

1. Define evaluate as function, then create an 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 assign the range of string then 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 an integer & push the value onto the 'p'.

6. 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.

7. Perform the arithmetic operation. Push the result back on the 'p'.

8. When the input expression has been completely processed, the result is on the Stack. Pop the 'p' & return the value.

9. Print the result of string after the evaluation of Postfix.

10. Attach output & input of above algorithm.

output:

>> Evaluated value is : 10

CSU

17/01/2024

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

24/01/2022

Practical-8

C51

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 contingous. The individual element of the linked list called a Node. Node comprises of 2 parts (1) Data (2) Next Data stores all the information wrt the element foreg- 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 solve these problems.

Algorithm:

1. Traversing of a linked list means visiting 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 transversed using the node which is referred by the head pointer of the linked list.

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.

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 revert back

6. We may lose reference to the 1st node in our linked list & 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 transverse the entire linked list.

7. We will use this temporary node as a copy of the node. We are currently transversing. Since we are making temporary node a copy of current one, the datatype of the temporary node should also be Node.

q

linkedlist()

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
```

24/01/2020

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.

9. But the 1st node is referred by current. So we can transverse to second one as n = n.next.

10. Similarly, we can transverse rest of data nodes in the linked list using same method by while loop.

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

12. The last node in the linkedlist is referred by the tail of the linked list. Since the last node doesn't have any next node, the value in the next field of the last node is None.

13. So we can refer the last node of the linkedlist as self.s = None

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

15. Attach the coding & output of above algorithm.

Practical-9

Aim: Programs based on Binary search tree by implementing Inorder, Preorder & Postorder, Transversal.

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.

Inorder: (i) Transverse the left subtree. The left subtree inturn might have left & right subtrees.
(ii) Visit the root node.
(iii) Transverse the right subtree & repeat it.

Preorder: (i) Visit the root node.
(ii) Transverse the left subtree. The left subtree inturn might have left & right subtree.
(iii) Transverse the right subtree & repeat it.

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

Source code:

```
class node:
    def __init__(self, value):
        self.left = None
        self.val = value
        self.right = None

class BST:
    def __init__(self):
        self.root = None
    def add(self, value):
        p = node(value)
        if self.root == None:
            self.root = p
            print("Root is added 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)
                        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:
        print(root.val)
        Postorde(root.left)
        Postorder(root.right)
```

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 added Successfully:', 9)
>>> t.add(15)
("Root is added Successfully:", 15)
```

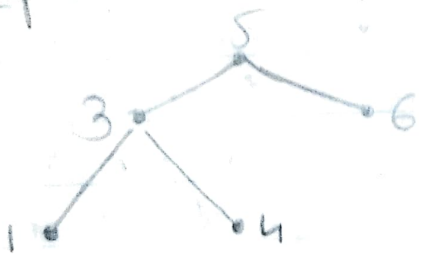
Algorithm:

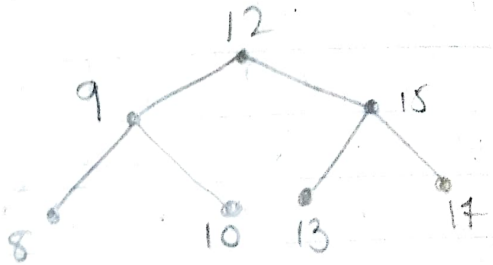
① Define class node & define init() method with 2 arguments. Initialize the value in this method.
② Again, Define a class BST that is Binary Search Tree with init() method with self argument and assign the root is None.
③ Define add() method for adding the node Define a variable p that p = node(value)
④ use If conditional statement for checking the condition that root is none, then use the else statement for if node is less than the main node then put or arrange that in left side.
⑤ use while loop for checking the node is less than or greater than the main node and break the loop if it is not satisfying.
⑥ use if statement within that else statement for checking that node is greater than, main root, then put it into rightside.
⑦ After this, leftsubtree & rightsubtree, repeat this method to arrange the node according to Binary Search Tree.
⑧ Define Inorder(), Preorder() & Postorder() with root argument & use If statement that root is none & return that in all.
⑨ In Inorder, else statement used for giving that condition first left, root & then right node.

(10) For Preorder, we've to give condition in else that first root, left and then right node
(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.

* Inorder: (LVR)

Step1:





Step2

Step3

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

```
>>> t.add(1)
("Root 1) added successfully:", 1)
>>> t.add(4)
("Root 17 added successfully:", 4)
>>> t.add(8)
("Root 1> added successfully:", 8)
>>> t.add(10)
("Root 1> added successfully:", 10)
>>> t.add(13)
("Root 1> added successfully:", 13)
>>> t.add(17)
("Root 1) added successfully:", 17)
```

Inorder (t.root)

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

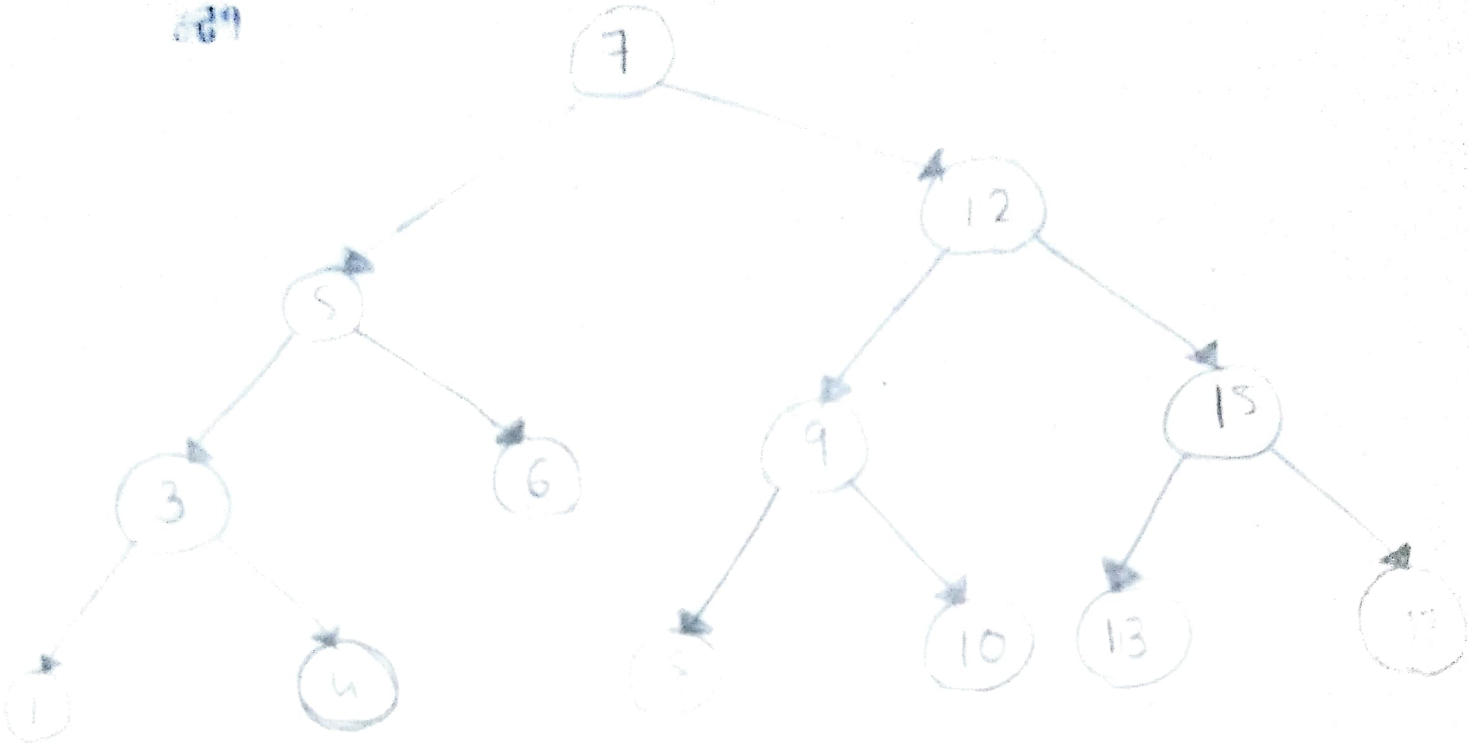
Postorder (t.root)

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

Preorder (t.root)

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

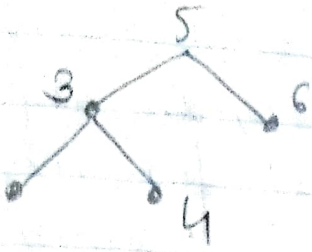
* Binary Search Tree

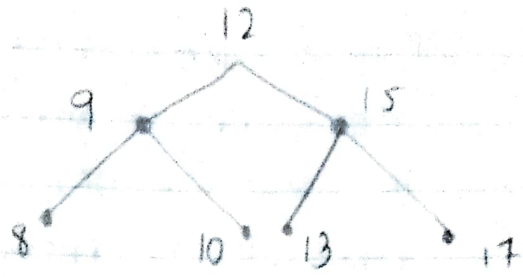


Preorder: (VLR)

Step 1:

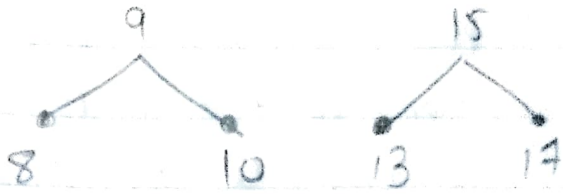
7





Step 2:

7 5 3 6 12

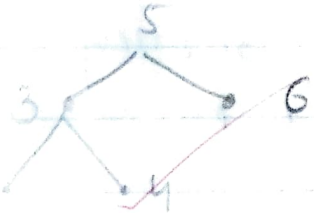


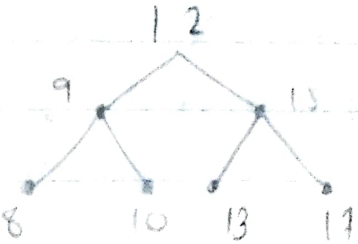
Step 3

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

Postorder: (LRV)

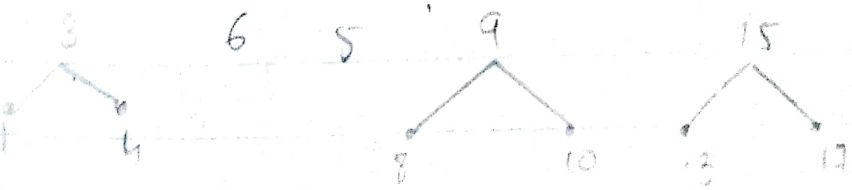
Step 1





7

Step 2



Step 3

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

:89

Practical - 10

Aim: To sort a list using Mergesort.

Theory: Like Quicksort, Mergesort is 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] & arr[m+1..r] are sorted sub-arrays into one. The array is recursively divided into two halves till the size becomes into action and starts merging arrays back till complete array is merged.

Applications:

Mergesort is useful for sorting linked lists in O(nlogn) time.

Mergesort is useful for sorting access data sequentially & the need of random access is low.

2. Inversion count problem.
3. Used in External Sorting

Source 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("\n mergesorted list:",arr)
```

Output:

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

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

mergesort 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 where sequentially accessed data structure are very common.

Practical-4

Aim: To demonstrate use of circular queue.

Theory: In a linear queue, once queue is completely full, it 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. After we dequeue any element to remove it from the queue, we're 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 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 a circular queue, head pointer will always point front of the queue & tail pointer will always point end of the que

Source 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:
            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):
            print (self.l[self.f])
            self.f = self.f+1
        else:
            print ("Queue is empty")
            self.f = 5

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]
```

14/02/20

Initially, the head & tail pointer will be pointing to the same location. This would mean 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've some real-world examples.

1. Computer Controlled traffic signal system.
2. CPU scheduling & memory management.