☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member's Signature
1)	Linear search program: unsorted linear search and sorted linear search	31	29/11/19	29/11/19
2)	Implement binary search to find an item in an ordered list	33	6/12/19	
3)	Implementation of Bubble sort program on given list	35	20/12/19	01/12/19
4)	Implement quick sort to sort the given list	36	20/12/19	
5)	Implementation of stack	38.	3/01/2020	03/01/2020
6)	Implementing Que in python	40	10/01/20	10/01/2020
7)	evaluation of given string using stack	43	16/1/20	
8)	linked list	45	24/1/20	

★★ INDEX ★★

No.	Title	Page No.	Date	Staff Member's Signature
9)	Binary search, inorder, preorder & postorder	47	7/2/20	
10)	merge sort implementation	53	14/2/20	
11)	Circular queue implementation	57	14/2/20	

Practical - 1

Aim: Implement Linear search to find an item in a list.

Unsorted Linear search:

Algo:

1) assign a list in a variable.
2) Take input from a user in a variable.
3) Use for conditional statement till range of list assigned in step 1, to check wether the variable in step 2. is equal to element in list.
4) Print the position if found or else print not found.

Sorted Linear search:

Algo:

1) assign a list in a variable taken from user and sort it.
2) print the list and take input for number to be searched.
3) Use for conditional statement to iterate over the list and check wether the input variable is equal to element in a list.
4) Print the position if found or else print not found.

06

unsorted

```
a = [4,3,5,8,9,2]
b = int(input("Enter number:"))
for i in range(0,len(a)):
    if a[i]==b:
        print("found at:",i+1)
        break
    else:
        print("not found")
```

output:

```
>>> Enter number: 5
found at: 3
>>> Enter number: 0
not found.
```

sorted

```
a = list(input("enter list:"))
a.sort()
print a
b = int(input("enter search no:"))
for i in range(len(a)):
    if b==a[i]:
        print("found at:",i+1)
        break
else:
    print("not found")
```

output:

```
>>> enter list: 4,9,2,1,9
[1,2,3,4,9]
enter search no: 2
found at: 2.
```

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

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

Practical - 2

Aim: Implement Binary search to find a no. from a list.

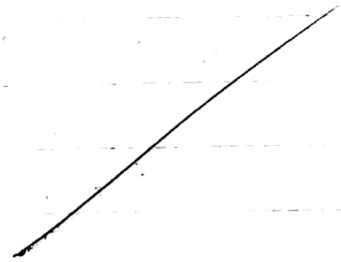
Theory:

Binary search is also known as half interval search, logarithmic search or binary chop is a search algorithm that finds the position of a target value within a sorted array. If you are looking for the number which is at the end of the list then you need to search entire list in linear search, which is time consuming. This can be avoided by using Binary fashion search.

Algorithm:

① Create Empty list and assign it to a variable.
② Using Input method, accept the range of given list.
③ Use for loop, add elements in list using append() method.
④ Use sort() method to sort the accepted element and assign it in increasing ordered list print the list after sorting.
⑤ Use If loop to give the range in which element is found 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 an orguement & key of the element that element has to be searched.
⑧ Initialize 0 and 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 and still the element to be searched is not found then find the middle element (m)
(11) If statement in list and still the element to be searched is not found then find the
(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(1) = mid(m)-1
(12) Repeat till you found the element stick the input & output of above algorithm.



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

output:

Enter list: 4,9,2,3,7

Enter search no: 3

number found at 4.

12.

code:

```
a = list(input("Enter the element:"))
for i in range(0,len(a)):
    for j in range(0,len(a)-1):
        if a[j] > a[j+1]:
            a[j],a[j+1] = a[j+1],a[j]
print(a)
```

output:

Enter the element: 4,5,9,1,2

[1,2,4,5,9]

Practical-3

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 given elements in according or descending order by comparing two adjacent elements at a time.

Algorithm:

1. Initialize a variable ~~wit~~ and take a input in it with list datatype.
2. ~~for~~ use for conditional statement to iterate over the element of list in step 1.
3. use ~~another~~ for conditional statement to iterate ~~the~~ the element and check wether that element is bigger than it's next element
4. If it is true than swap the elements
5. print the list

Practical-4

Aim: Implement Quick sort to sort the given list

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

Algorithm:

① Quick sort first select 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 then 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 partion process is to move items that are on wrong side with respect to pivot value which also converging on the split point.

④ We begin by incrementing leftmark until we locate a value that is greater than the P.V. we then decrement rightmark until we find 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 split point.

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>=l:
                r = r-1
            if r<l:
                done = True
            else:
                t = alist[l]
                alist[l] = alist[r]
                alist[r] = t
        t = alist[first]
        alist[first] = alist[r]
        alist[r] = t
        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 : 5
Enter element: 4
Enter element: 3
Enter element: 2
Enter element: 1
[1,2,3,4,5]
```

⑤ At this point where rightmark ~~bet~~ becomes less than leftmark, we stop. The position of rightmark is now the split point

⑥ The pivot value can be exchanged with the content of split point and p.v is now in place

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

⑧ The quicksort function invoker a recursive function, quicksort Keeper

⑨ Quicksort helper begins with same base as the merge sort

⑩ If length of the list is less than or equal to 1, it is already sorted.

⑪ If it is greater than, then it can be partitioned and recursively sorted

⑫ The partition function, implements the process described earlier

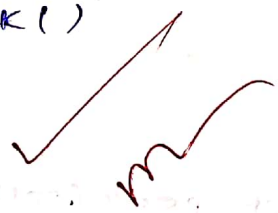
21/12/19

```
class stack:
    global tos
    def __init__ (self):
        self.i = [0,0,0,0,0]
        self.tos = -1
    def push (self,data):
        n = len (self.i)
        if self.tos = n-1 :
            print ("stack is full")
        else:
            self.tos = self.tos+1
            self.i [self.tos] = data
    def pop (self):
        if self.tos < 0:
            print ("stack is empty")
        else:
            K=self.i [self.tos]
            print ("data =", K)
            self.i [self.tos] = 0
            self.tos = self.tos -1
```

```
def peek (self):
    if self.tos < 0:
        print ("stack is
                empty")
    else:
        a = self.i [self.tos]
    print ("data =", a)
```

```
s = stack ()
>>> s.push (10)
>>> s.push (20)
>>> s.push (30)
>>> s.push (40)
>>> s.push (50)
>>> s.push (60)
stack is full
>>> s.i
[10,20,30,40,50]
```

```
>>> s.pop()
  data = 50
>>> s.pop()
  data = 40
>>> s.pop()
  data = 30
>>> s.pop()
  data = 20
>>> s.;
[10, 0, 0, 0, 0]
>>> s.peek()
    10
```



Practical - 5

Aim: Implementation of stack 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 principle as the elements that was inserted last will be removed first stack can be implemented using an array as well as linked list has three basic operation: push, pop, peek. The operation of adding and 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.

3) Define method to push & pop under the class stack.

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.

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

6) Push method used to insert the element but pop method used to delete the element from the stack.

7) If in pop method, value is less than 1, then return the stack is empty or else delete the element from stack at top most position.

8) Assign the element values in push method to an end print the given value is poped not

9) first condition checks whether the no. of elements are zero, while the second case, whether top assigned any value. if top is not assigned any value, then you can be sure that stack is empty.

10) Attach the input & output of above algorithm.

03/01/22

```
Class Queue:
    global r
    global f
    global a
    def __init__(self):
        self.r=0
        self.f=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 elemented 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-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 operation 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(): Creates a new empty queue.

- enqueue(): Insert an element at the rear of the queue and similar to that of insertion of linked using tail.

- Dequeue(): Return the element which was at the front the front is moved to the successive element. A dequeue operation cannot remove element if the queue is empty.

Algorithm:

1) Define a class Queue and assign global variables then define init() method with self arguement in init(), assign or initialize the initial value with the help of self arguement.

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

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 succesfully and increment by 1.

4) Define dequeue() with self-arguement 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.

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.

b. dequeue (1)

Queue element inserted : 1

b. enqueue (2)

Queue element inserted : 2

b. enqueue (3)

Queue element inserted : 3

b. enqueue (4)

Queue element inserted : 4

b. enqueue (5)

Queue element inserted : 5

b. enqueue (6)

Queue is full

10/01/2022

b. dequeue()

Queue element deleted : 1

b. dequeue()

Queue elemet deleted : 2

b. dequeue()

Queue element deleted : 3

b. dequeue()

Queue elemat deleted : 4

b. dequeue()

Queue elemet deleted : 5

b. dequeue()

Queue is empty.

S8

```
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 = " 3 5 7 * -"
r = evaluate(s)
print(" The evaluated value is :", r)
```

Output:

The evaluated value is : -32

practical - 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 parenthesis. 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.

Algo:

1) Define evaluate as function then create a empty stack in python.

2) Convert the string to a list by using the string method split

3) Calculate the length of string & print

4) Use for loop to assign the range of string then give condition using if statements

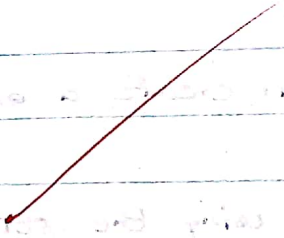
5) Scan the token list from left to right. If token is an operand convert it from a string to an integer & push the value onto the p.

6) If the token is an operator (*, /, +, -) It will need two operands. Pop the p twice. The first pop is second operand and the second pop is the first operand.

7) Perform the arithmetic operation and push the result back on the 'm'.

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

9) Print the result of string after the evaluation of postfix.



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

class linkedlist:
    globals 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.a
    def display(self):
        head = self.s
        while head.next != None
        print(head.data)
        head = head.next
        print(head.data)
```

Practical - 8

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

Theory: A Linked List is a linear data structure which stores the elements in a rock in a linear fashion but not 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 elements for e.g., 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.

Algo:

1) Transversing of a linked list means rising the nodes, in the linked list in order to perform same operation on them.

2) The entire linked list means can be accessed to use the first node of the linked list. The first node of the linked list from is reffered 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 entire linked list because the head pointer is our only reference to the 1st rank in the linked list, modifying the reference of the head pointer can lead to changes which we cannot revert back.

6) We will use this temporary note as a copy of the node we are currently transversing since we are making temporary node a copy of current node the datatype of the temporary node should also be note

7) Now that current is displaying 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.

8) But 1st node is referred by current node so we can transverse it to 2nd nodes as h = h.next.

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

10) Our concern now is to find terminating condition.

11) last node is re

output:

```
>>> start.addL(40)
>>> start.addL(20)
>>> start.addL(20)
>>> start.addB(10)
>>> start.addB(50)
>>> start.display()

50
10
10
40
```

1) 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.

2) So we can refer the last node of the linked list self.s = none.

3) we have to now see how to start transversing the linked list & how to identify wether we have reached the last node of linked list or not.

Practical - 9

Aim: Program 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 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.

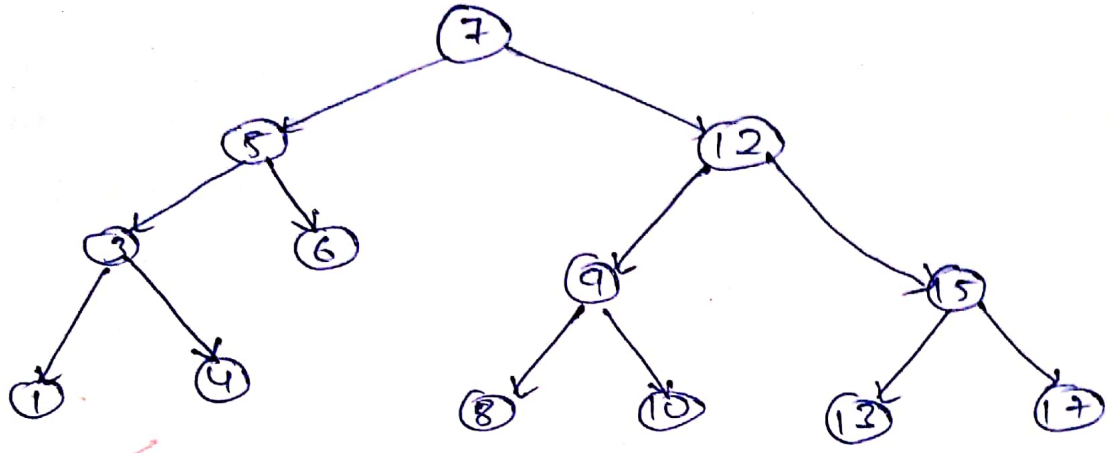
→ Preorder:

1) Visit the root node
2) Traverse the left subtree the left subtree inturn might have left and right subtree.
3) Traverse the right subtree, repeat it.

→ Postorder:

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

Binary search Tree



Algo:

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

2) Again, Define a class BST that is Binary search tree with init() method with self arguement and assign the root as None.

3) Define add() method for adding the node Define a variable p that p = node / value

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.

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 in to rightside.

7) After this, left subtree and right subtree, repeat this method to arrange the node according to Binary search Tree.

8) Define Inorder(), preorder() and postorder() with root arguement 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.

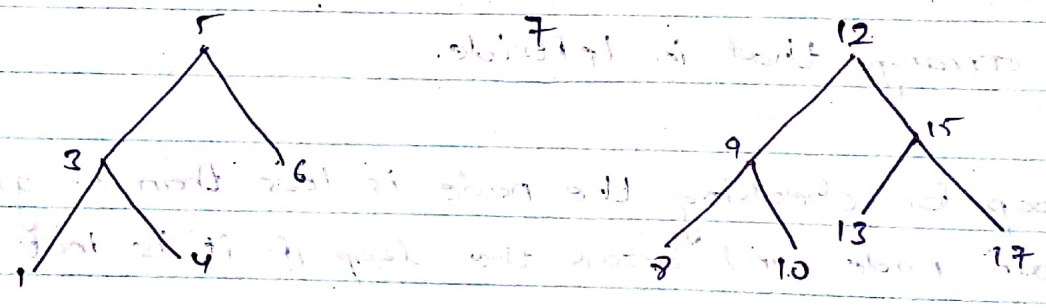
10) For preorder, we have to give condition in else that first root, left and then right node.

11) For postorder, in else part, assign left then right and then go for root node

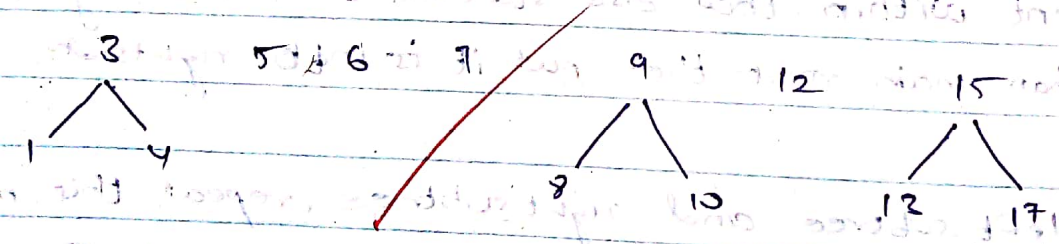
12) Display the output and input of above algorithm.

Inorder : (LVR)

1)



2)

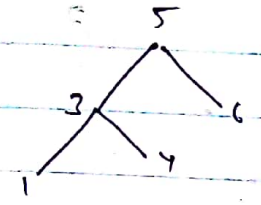


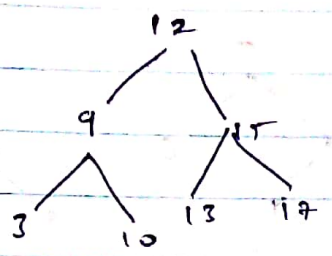
3) 1 3 4 5 6 7 8 9 10 12 13 15 17

```
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, success")
        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 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)
```

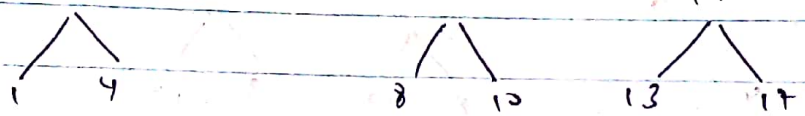
Preorder : (VLR)

1) 7





2) 7 5 3 1 12 9 15 4 8 10 13 17

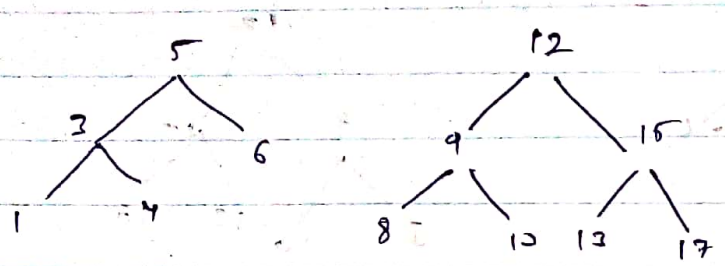


3) 7 5 3 1 4 6 12 9 8 10 15 13 17

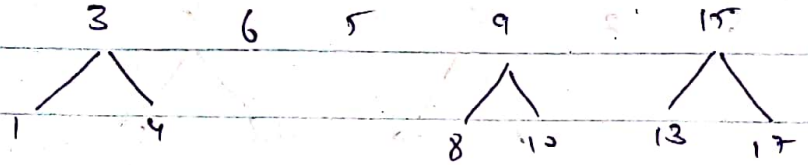
12

Postorder: (LRV)

1)



2)



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

```
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()
```

Outputs

```
>>> t.add(25)
root is added at 25
>>> t.add(15)
15, node is added to leftside at 25
>>> t.add(50)
50, node is 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(4)
4, node is added to leftside at 10
>>> t.add(12)
12 inode is added to rightside at 10
>>> t.add(35)
35, node is added to leftside at 50
>>> t.add(70)
70, node is added to rightside at 50
```

```
>>> t.add(21)
    21, node is added to leftside at 25
>>> t.add(44)
    44, node is added to rightside successfully at, 25
>>> t.add(66)
    66, node is added to leftside at 70
>>> t.add(90)
    90, node is added to rightside at 70

>>> postorder(t.root)
4       44      50
12      35      25
10      66
22      90
15      70
21

>>> preorder(t.root)
25      25      66
15      50      90
10      35
4       21
12      44
22      70

>>> Inorder(t.root)
4       35
10      44
12      50
15      66
22      70
25      90
31
```

Practical-10

Aim: To sort a list using merge sort

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] 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 become 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

Applications:

1. Merge sort is useful for sorting linked lists in O(n logn) tion. Merge sort accesses data sequentially and the need of random access is low.
2. Inversion count problem.
3. Used in External sorting.
4. Mergesort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially. and thus is 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 ("In Merge sorted List : ", arr)
```

Practical - 11

Aim: To demonstrate the use of circular queue.

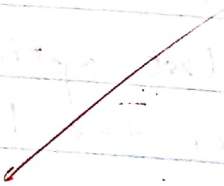
Theory: In a linear queue once the queue is completely full it is not possible to insert more elements. Even if we dequeue the queue to remove some of the elements until the queue is reset, no new elements can be inserted.

When we dequeue any element to remove it from the queue we are actually moving the front of the queue forward, thereby reducing the overall size of 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 data 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.

Initially, the head and the tail pointer will be pointing to the same location this would mean that the queue is empty. New data is always added to the location pointed by the tail pointer, and once the data is added tail pointer is incremented to point to the next available location applications.

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



```
class Queue:
    global r
    global f
    def __init__(self):
        self.r=0
        self.f=0
        self.I =[0,0,0,0,0,0]
    def add (self,data):
        n= len (self.I)
        if (self.r<n-1):
            self.I[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.I[self.r] =data
                self.r=self.r+1
            else:
                self.r=s
                print ("Queue is full")
    def remove (self):
        n = len (self.I)
        if (self.f<=n-1):
            print ("Data removed:", self.I[self.f])
            self .f= self.f+1
        else:
            s =self.f
            self.f =0
```

```
        if (self.f<self.r):
            print (self.I[self.f])
            self.f=self.f+1
        else:
            print ("Queue is empty")
            self.f=-1

q = Queue()
q.add(44)
q.add(85)
q.add(66)
q.add(77)
q.add(88)
q.add(99)
q.remove()
q.add(66)
```

data added : 88

Queue is full

Data removed : 44

MR
28/04/20