☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member's Signature
1.	Implement linear search for to find an item in the list	35	29/11/19	
2.	Implement binary search to sort and find an item in the list	37	6/12/19	
3.	Implementation of bubble sort program on given list	39	6/12/19	
4.	Implementation of quick sort to sort the list.	41	20/12/19	
5.	Implementation of stacks using python list	48	3/1/20	
6.	Implementing a queue using list	49	10/1/20	
7.	Evaluation of Postfix. Exp.	51	17/1/20	
8.	Implementation of single linked list	44	24/1/20	

☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member Signature
	~~practical~~			
9.	Inorder preorder, Postorder	53		
10.	Merge sort	56	14/2/2020	
11	Circular queue	58.	14/2/2020	

Practical - 1

Aim:- Independent linear search to find an item in the list

Theory:-

linear search

linear search is one of the simple list 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 examining the middle term.

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

Unsorted:-

Step1: Create an empty list & origin it to a variable

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

Step3:- Use for loop for adding the elements into list.

Unsorted program :-

```
a=list(input ("Enter any list of number"))
b= int (input("Enter a number to be searched"))
for i in range (len(a)):
    if (b==a[i]):
        print "Number searched is found at
                position", i
        break
if(b!=a[i]):
    print" Number not found"
```

Output :-

Enter any list of number 18,22,11,17,63

Enter a number to be searched:17

Number searched is found at position 3

Step 4 :- find the new list

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

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

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

Step 8 :- if the element is found then print the statement that the element is found along with the elements position.

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

Step 10 :- Draw the output of the given algorithm

2] Sorted :-

Step 1 :- Create empty list & assign it to a variable

Step 2 :- Accept total no. of element to be inserted into the elements in the list.

```
a=list(input("Enter any list of numbers:"))
a.sort()
print a
b=int(input("Enter a number to be searched:"))
for i in range(len(a)):
    if(b==a[i]):
        print "Number is found at position",i
        break
if(b!=a[i]):
    print "Number not found"
```

output

Enter any list of number 22, 11, 36, 63, 54

a = [11, 22, 36, 54, 63]

Enter a number to be searched: 36

Number is found at position 2

24/01/2020

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

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

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

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

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

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

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

Step 10:- Use another if loop to point that the element is not found if the element which is accepted from the user is not in the list.

Practical-2

* Aim:- Implement binary to find a search key no in the list.

* Theory:-

Binary search

Binary search is also known as half interval search logarithmic search or binary search 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 search

* Algorithm

Step 1:- Create Empty list & Assign it to a variable

Step 2:- Using input method accept the range of given list

Step 3:- Use for loop, add elements in list using append() method

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

Source code:-

```
#Binary search
a=list(input("Enter the list to be sorted:"))
b=int(input("Enter the number to be searched:"))
a.sort()
print(a)
l=0
h=len(a)-1
while(h>=l):
    mid=(l+h)/2
    if[b< a[l]or b>a[h]):
        print "Number not found"
        break
    elif(b==a[mid]):
        print "Number is found at position,mid
        break
    elif(b<=a[mid]):
        h=mid-1
    elif(b>=a[mid]):
        l=mid+1
```

24/01/2022

Output

Enter the list to be searched: 38, 77, 66, 55, 44, 33

Enter the number to be searched = 33

[33, 44, 55, 66, 77, 88]

Number is found at position 6

Enter the list to be sorted: 99, 77, 66, 44

Enter the number to be searched: 55

[44, 66, 77, 99]

Number not found

Step 6:- Use if loop to give the range in which element is found is given their display a message".

Step 7:- Then use else statement, if statement is in range then satisfy the below condn

Step 7:- Accept an argument & key of the element that element user to searched

Step 8:- Initializing first to 0 & last to element of list as array its starting 0 since it is initialized 1 less than the total count.

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

Step 10:- if statement in list & full the element to the searched is not found then find the middle element (m)

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

Initialize last (m) = mid (m) - 1

Else

Initialize first (m) = mid (m) + 1

* Repeat till you found the element state the input & output of above algo.

Practical No. 3

Aim:- Implementation of bubble sort program on a given list

Theory:-

Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist wrong order. This is the simplest form of sorting available. In this we sort the given element in accending or decending order by comparing two adjacent elements at a time.

Algorithm:-

Step 1: Assign an empty list to a variable.

Step 2: Assign the range of the list by the user

Step 3: Use for loop with range equal to the accepted value. Add elements in list by using append method.

Step 4: Use for loop with range equal to one less the length of list. Use another for loop inside the previous loop statement with range one less than the length of list minus the argument passed by previous loop.

```
x = int (input ("Enter the range of the list:"))
alist = [ ]
for i in range (0,x):
    b = int (input ("Enter element:"))
    alist.append (b)
for parzer in range (len (alist )-1):
    for comparison in range (len (alist)-1-parzer):
        if (alist [comparison] > alist [comparison+1]
            t = A [comparison]
            a.list [comparison] = son [comparison +1]
            alist [comparison+1] = t
print (alist)
```

Output :-

Enter the range of the list: 5

Enter element: 5

Enter element: 9

Enter element: 3

Enter element: 1

Enter element: 4

(1, 3, 4, 5, 9)

22/01/20

Step5:- Use if conditional statement to check whether the element is greater than next adjacent element

Step6:- if the condition is true, swap the two elements by using a temporary variable

Step7:- Print the updated list.

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 & conquer technique

Algorithm:

Step 1: Quick sort first selects a value which is called pivot value. first element serves as our first pivot value. Since we first know that it will last the list.

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

Step 3: Partitioning begin by locating two position markers, let's call them leftmark and right, at the beginning and end of remaining items in the list. The goal of the partition process is to move the elements that are on the wrong side with respect to pivot value

```
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 list[t]<=pivot:
            l = l+1
        while alist[r]>=pivot and r>=1
            r = r-1
    if r<l:
        done = True
    else
        t = alist[l]
        alist[l] = alist[r]
        alist[r] = t
x = int(input("Enter range of list"))
alist[]
for b in range(0, x)
    b = int(input("Enter element"))
    alist.append
n = len(alist)
```

Output :-

Enter the range of list: 4

Enter element: 7

Enter element: 1

Enter element: 9

Enter element: 4

[1, 4, 7, 9]

24/01/2021

Step 4:- We begin by incrementing leftmark until we locate a value that is greater than the P.V we then decrement rightmark until we find value that is less than the pivot value. At this point we have discovered two items that are out of place.

Step 5:- At the point where rightmarks becomes less than leftmarks we stop. The position of rightmarks is now the split value.

Step 6:- The pivot value can be exchanged with the contents of split & pivot value is now in place

Step 7:- In addition, all item to the left of split point are less than P.V and the items to the right of split point, is greater than PV. The list can now be divided at split point & quicksort can be invoked in two

Step 8:- The quicksort function invokes recursive function, quicksorthelper

24/01/22

merg = sort

Step 9:- Quicksorthelper begins with base case

Step 10:- if the length of the list is less than 0 or equal to 1, It is already sorted

Practical - 8

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

Theory:- A linked list is a linear data structure which stores the elements in a node in a linear fashion but necessarily contiguous. The individual element of the linked list is called a Node. Node comprises of 2 parts ① Data ② Next. Data stores all the information w.r.t the element for example roll no., address, etc. whereas next refers to the next node. In case of large list if we add/remove any element of list has to adjust itself everytime we add, it is very tedious task so linked list is used

Algorithm:

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

Step 2: The entire linked list means can be access using the first node of the linked list. The first node

```
class node:
    global data, next
    def __init__(self, item):
        self.data = item
        self.next = None
class linkedlist:
    global s
    def __init__(self):
        self.s = None
    def add L(self, item):
        new node = node(item)
        if self.s == None:
            self.s = newnode
        else:
            head = self.s
            while head.next != None:
                head.next = newnode
    def add B(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
```

```
q = linkedlist()
```

Output:-

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

90
80
70
60
40
30
20
10

of the linked list in turn is referred by Head pointer of the linked list

Step3:- Thus the entire linked list can be traversed using the node which is referred by head pointer

Step4:- Now that we know we can traverse the entire linked list using the head pointer. we should only use it to refer the first node of list only

Step5:- We should not use head pointer to traverse the entire linked list because the head pointer is our only reference to the 1st node in the linked list, modifi the reference of the head pointer can lead to changes which we cannot revert

Step6:- We may lose the reference to the 1st node in our linked list, and hence part of our linked list. So in order to avoid making some unwanted chang to the node We will use a temporary to traverse the entire linked list

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

Step 8: Now that the current is referring to the first node, if we want to access 2nd node of list we can refer it as the next node of the 1st node.

Step 9:- But the 1st node is referred by current. So we can traverse to 2nd node as h=h

Step 10:- Similarly we can traverse rest of node the linked list using same method by while

Step 11:- Our concern now is to find terminating con for the while loop

Step 12:- The last node in the linked list is referred by the tail of linked list. Since the last node of linked list does not any next node.

Step 13:- So we can refer the last node of list as reff.s = None.

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

Step 15:- Attach the coding as input 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 a pile. The elements in the stack are added or removed only from one position- the topmost position. Thus, the stack works on LIFO principle as the elements 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, return the topmost element end of the stack.

* Algorithm:-

Step1:- Create a class stack with instance variable items.

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

Step3:- Define the methods push & pop under the class stack.

Code


```
print "
class stack:
    global tos
    def __int__(self):
        self.l = [0,0,0,0,0]
        self.top = -1
    def push(self,data):
        n=len(self.l)
        if self.top=n-1:
            print "stack is full"
        else:
            self.tos = self.tos+1
        self.l[self.tos]=data
    def pop(self):
        if self.tos<0
            print "Stack is empty"
        else:
            k = self.l[self.tos]
            print "Data = ",k
        self.l[self.tos]=0
        self.tos = self.tos-1
    def peek(self):
        if self.tos<0 =
            print "stack is empty"
        else:
            a = self.l[self.tos]
        print "Data = ",a
```

S.A

Output :-

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

Step 4:- Use the if statement to give the condn that if length of given list is greater than the range of list then print stack is full.

Step 5:- Or else print statement ~~or~~ insert the element into the stack & initialize the values

Step 6:- Push method used to insert the element but pop method used to delete the element from the stack.

Step 7:- if in pop method value is less than 1 then return the stack is empty or else delete the element from stack

Step 8:- first condition checks wether the no. of elements are zero while the second case wheather its assigned any value. if not not then we are sure stack is empty

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

Step 10:- Attach the input & output of above algorithm.

Practical-6

* Aim:- Implementing a Queue using list.

* Theory:- Queue is a linear data structure which has 2 reference front & rear. Implementing a queue using python is the simplest as the python lists 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 & element of queue is deleted which is at front.

* Queue():- Creates a new empty.

* Enqueue():- Insert an element at the rear of the queue and similar to that of the 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 & assign global variables then define init() method

Code:-

```
class Queue:
    global f, r, a
    def __init__(self):
        self.f = 0
        self.r = 0
        self.a = [0,0,0]
    def enqueue(self, value):
        self.n = len(self.a)
        if ((self.r == self.n):
            print("Queue is full")
        else:
            self.a(self.r) = value
            self.r += 1
            print("insert =", 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()
```

Output:-

```
lub.enqueue(2)
Insert = 2
>>> b.enqueue(4)
Insert = 4
>>> b.enqueue(7)
Insert = 7
>>> b-q
[2, 4, 7]
>>> b-dequeue()
Queue element deleted : 2
>>> b-dequeue()
Queue element deleted : 4
>>> b-dequeue()
Queue element deleted : 7
>>> b-dequeue()
Queue is empty
>>> b-q
[0, 0, 0]
```

```
>>> b-enqueue(8)
Queue is full
```

argument init() assign or initialize with the help of self argument

Step 2:- Define an empty list & define an 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 added successfully & increment by 1

Step 4:- Define dequeue() with self argument order 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 & using that delete the element from left side & increment 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 deletion & display the list after deleting the element from the list.

Practical-7

* Aim:- Program on evaluation of given string by using stack in python Environment in Postfix

* Theory:- The postfix expression in free of any parenthesis further further use took care of the priorities of the operators in the prog. A given postfix expression can easily be evaluated using stacks. Reading the expression is always from left to right in Postfix

* Algorithm

Step 1:- Define evaluate as function then create a empty stack called in python

2- Convert the string to a list by using string word' split

3- Calculate the length of string & print it.

4- Use for loop to assign the range of string then give cond^n using if stmt.

```
#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))
        elif k[i] == '/':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) / int(a))
    return stack.pop()
```

```
s = "2 5 5 + *"
r = evaluate(s)
print "The evaluated value is = ", r
```

*output :-

The evaluated value is = 20.

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 entering after evaluation of postfix

10- Attach out & input of above algorithm

Practical-9

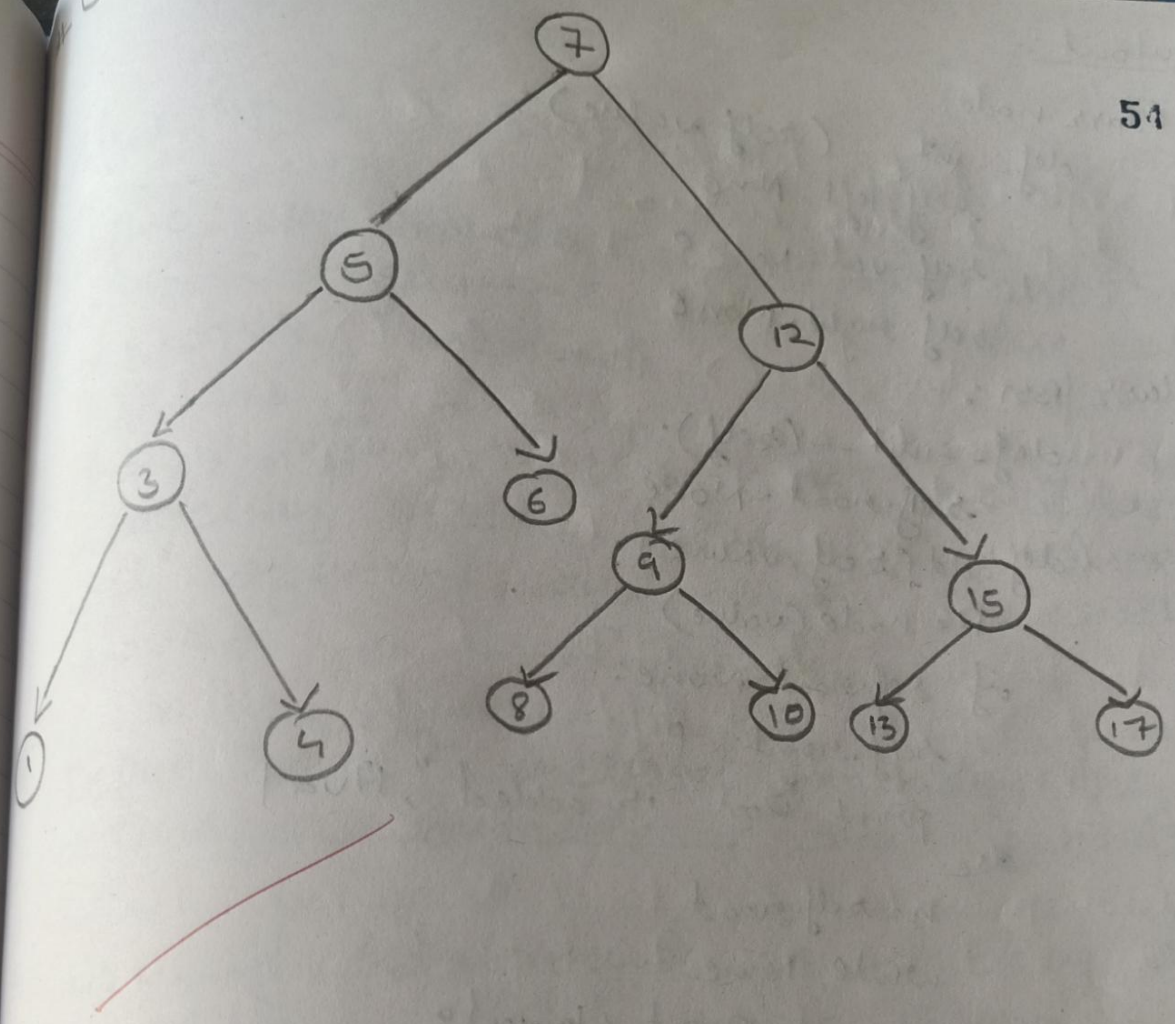
- Aim:- Program based on Binary search tree by implementing Inorder, Preorder & postorder, Transversal,

- Theory: Binary tree is a tree which supports max 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 intern might have left and right subtrees.
ii) Visit the root node.
iii) Transverse the right subtree & repeat it.

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

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


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

Output :-

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

Algorithm :-

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

2. Again, define a class BST i.e Binary searching 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 statement for checking the condition that root is none then use else statement for if node is less than the main node then put ~~an~~ arrange that in leftside.

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

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

7: After this left subtree & right subtree repeat this method to arrange the node according to binary tree

8: Define Inorder(), Preorder() & postorder with root arrangement and use if statement that root is none & return that in all.

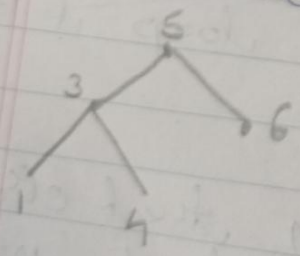
9: In Inorder, the statement used for giving that condn first ~~left~~ root, left & then right root node

10 For preorder we have to give condn in the that first root, left & right node

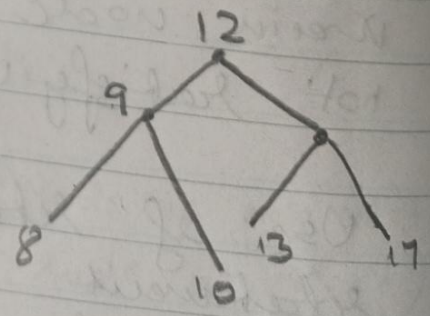
11 for postorder, In the part, arrange left then right & then go for root node

12 Display the output & input of above algorithm

Inorder (LVR



7



```
def Inorder(root):
    if root == None:
        return
    else:
        Inorder(root.left)
        print(root.val)
        Inorder(root.right)
def pre-order(root):
    if root == None:
        return
    else:
        print(root.val)
        Preorder(root.left)
        Preorder(root.right)
def Portorder(root):
    if root == None:
        return
    else
        portorder(root.left)
        Portorder(root.right)
        print(root.val)
t = BST()
```

Aim :- To sort a list using merge sort

Theory :- Like Quick-sort, mergesort is a divide & conquer algorithm. It divides input array in two halves and then merges the two sorted halves. The merge() function is used for merging two halves.

The merge(arr, l, m, r) is key process that assumes that arr[l ... m] and arr[m+1 ... r] are sorted and merges the two sorted sub-arrays into one. The array is recursively divided in two halves till the size becomes 1, the merge processes comes into action & starts merging arrays back till the complete array is merged.

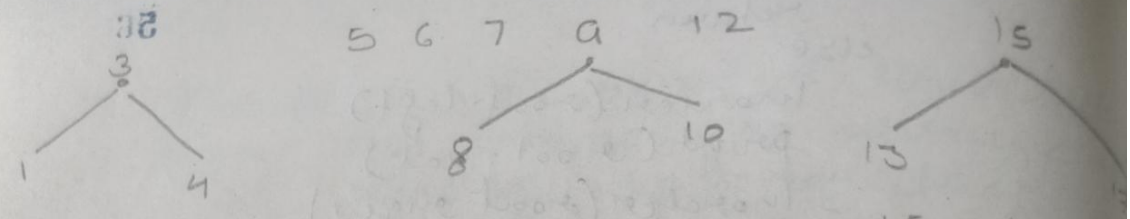
1. Merge sort is useful for sorting linked lists in O(nlogn) time.

~~Linked lists~~

- Merge sort accesses data sequentially and the need need of random access is low.

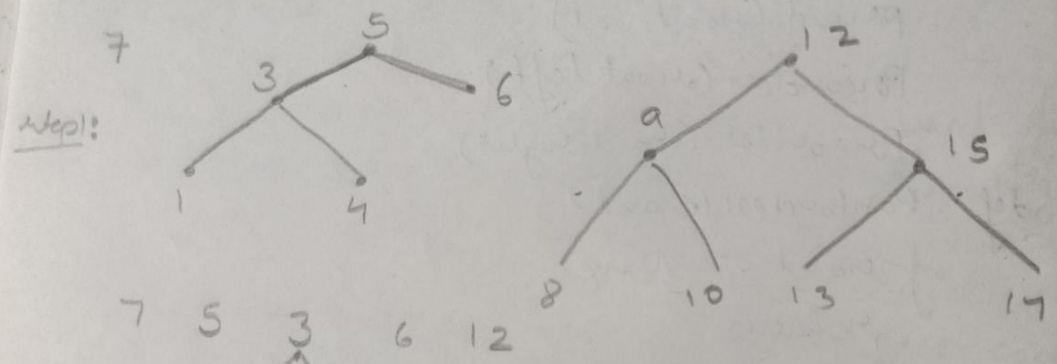
2. Inversion count Problem

3. Used in external sorting



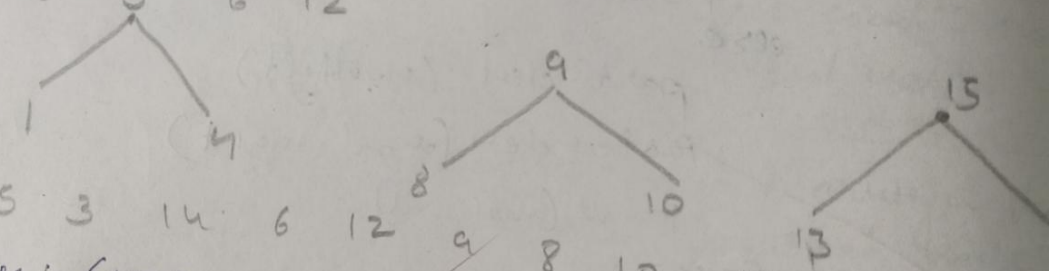
1 3 4 5 6 7 8 9 10 12 13 15 17

* Preorder :- (VLR)



Step1:

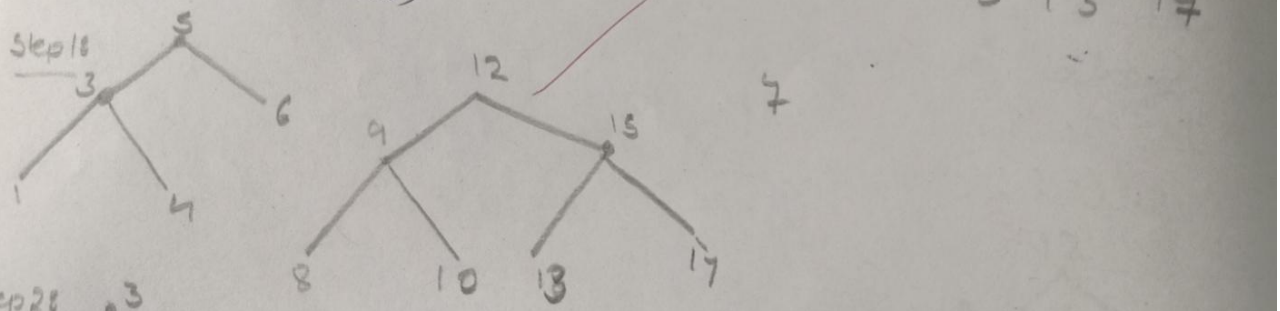
7 5 3 6 12



Step2:

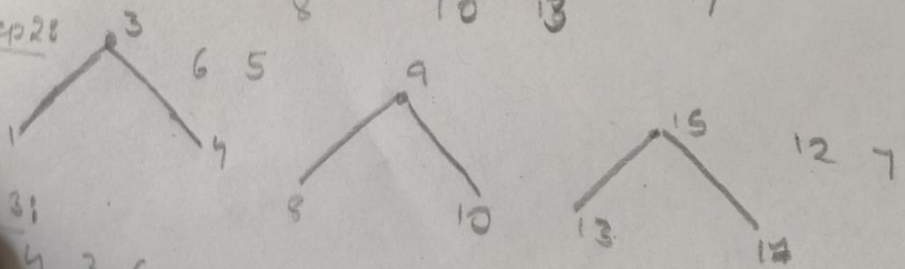
Step3: 7 5 3 1 4 6 12 9 8 10 15 13 17

* Postorder :- (LRV)



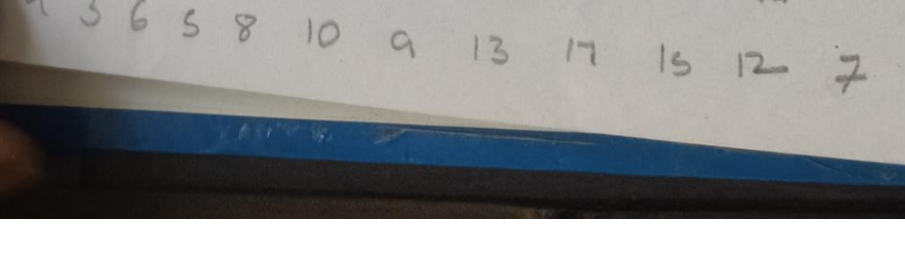
Step1:

7



Step2:

6 5



Step3:

12 7

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

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially and is thus popular in sequentially accessed data structures are very common.

```
def mergeSort(arr):
    if len(arr)>1:
        mid = len(arr)//2
        lefthalf = arr[:mid]
        right half = arr[mid:]
        mergeSort(lefthalf)
        mergeSort(right half)
        i=j=k=0
        while i<len(lefthalf) and j<len(right half):
            if lefthalf[i]<right half[j]:
                arr[k] = lefthalf[i]
                i = i+1
            else:
                arr[k]=right half[j]
                j=j+1
                k=k+1
            while i< len(lefthalf):
                arr[k] = lefthalf[i]
                i=i+1
                k=k+1
            while j<len(right half):
                arr[k]=right half[j]
                j=j+1
                k=k+1
```

Output:-

```
>>> arr= [14,2,2020]
>>> mergeSort(arr)
>>> mergeSort(arr)
>>> print(arr)
[2,14,2020]
```

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 de queue the queue to remove some of the elements until queue is reset, no new elements can be inserted. When we deque any element to remove it from the queue, we are actually moving the front of the queue forward, thereby reducing the overall size of the queue. And we cannot insert new elements, because the rear pointer is still at the end of the queue. The only way is to reset the linear queue for a fresh start.

Circular queue is also a linear data structure, which follows the principle of FIFO, but instead of ending the queue at the last position, it again starts from the first position after the last hence making queue behave like a circular data structure. In case of a circular [illegible] pointer will always poi[illegible]

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

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 queues are used:-

1. Computer controlled Traffic signal system uses circular queue
2. CPU scheduling & Memory Management.

```
>>> q.add(12)
Data added: 12
>>> q.add(4)
Data added: 4
>>> q.add(23)
Data added: 23
>>> q.add(1)
Data added: 1
>>> q.add(45)
Data added: 45
>>> q.add(5)
Queue is full
>>> q.l
[12, 4, 23, 1, 45, 0]
>>> q.remove()
Data removed: 12
>>> q.remove()
Data removed: 4
>>> q.remove()
Data removed: 23
>>> q.remove()
Data removed: 1
>>> q.remove()
Data removed: 45
```

```
>>>q.remove()
0
```

14/02/202