Output.

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

```
20
30
40
50
60
70
80
```

24/01/25

Step4: Now that we know that we can transverse the entire linked list using the head pointer, we shoud only use it to refer the first node of list only.

Step5 : 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 charges which we ~~convert~~ cannot revert back.

Step 6: We may lose the referace to the 1st node in our linked list, and hence most of our linked list. So in order to avoid making some unwanted charges to the 1st node, we will use a temporary node to transverse the entire linked list.

Step 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 node the data type of the temporary node should also be Node.

Aim: Implement Linear Search to find an Item in the list

Theory:

Linear Search

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

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

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

1) Unsorted:

Algorithm:

Step1: Create an empty list and assign it to a vairable.
Step2: Accept the total no. of elements to be inserted into the list from the user say 'n'
Step3: Use for loop for adding the elements into the list.
Step 4: Print the new list.
Step 5: Accept an element from the user that to be searched in the list.
Step6: Use for loop in a ronge from '0' to the total no. of elements to search the element from the list.
Step 7: Use if loop that the elements in the list is equal to the element accepted from user
Step8: If the element is found then print the statement that the element is found along with the elements position
Step 9: Use another if loop to print that the element is not found if the element which is accepter from user is not their in the list.
Step10: Draw the output of given algorithm

Unsorted.


```
s= int(input(" Enter the required number"))
a=[10,12, 9, 14, 17, 9 ]
for i in range (len (a)):
    if (a[i] == s):
        Print " required number found in position",i
        break

if(s!= a[i]):
    print " The required number not found"
```

Output

```
Enter the required number 4
required number found in position 2
```

86 Sorted

```
s=list(input("Enter the element:"))
s.sort()
print s
a=int(input("Enter the number to be searched"))
for i in range(len(s)):
        if(a==s[i]):
                print "number found! in the position",i
                break
else:
    print "number not found"
```

Output

```
Enter the element: 9,6,7,12,5,8
[5,6,7,8,9,12]

Enter the number to be searched: 8

number found! in position 3
```

176

Sorted:

Algorithm

Step 1: Create Empty list and assign it to a vairable accept total no. of elements to be inserted into the list from user, say 'n'

Step 2 :- Use for loop for using append() method to add the elements in the list. Use sort() method to sort the accepted element and assign in increasing order the list, then print the list

Step 3 :- Use If statement to give the range in which element is found in given range then display "Element not found".

Step 4 :- Then use else statement, if ~~sto~~ element is not found in range then satisfy the given cond?

Step 5 :- Use for loop in range from 0 to the total no. of elements to be searched before doing this accept on search no from user using Input statement

Step 6 :- Use if loop that the elements in the list is equal to the element accepted from user. If the element is found then print the statement that the element is found along with the element position.

Step 7 :- Use another if loop to print that the element is not found if the element which is accepter from user is not their in the list

Step 8 :- Attach the input and output of above algorithm.

29/11/19

PRACTICAL - 2

Aim Implement Binary Search to find or 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 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

Step 1: Create Empty list and assign it to a vairable. Using Input method accept the range of given list. Use for loop, add elements in list using append () method.

Step 2: Use sort() method to sort the accepted element and assign it in increasing ordered list print the list after sorting

02 Source Code

```
a=list (input("Enter list:"))
a.sort()
c=len(a)
s=int(input("Enter Search no:"))
if(s>a[c-1] or s<a[0]):
        print("not in a list")
else:
    first, last=0, c-1
    for i in range(0,c):
        m=int((first+last)/2)
        print("found at", m)
        if s==a[m]:
            print("number found")
            break
        else:
            if s<a[m]:
                last=m-1
            else:
                first=m+1
```

Output

20/12/19

```
Enter list: 8, 5, 9, 2, 1
[1, 2, 5, 8, 9]
Enter Search no: 9
(found at, 2)
```

Step 3: Use If loop to give the range in which element is found in given range then display a ~~ran~~ message "Element not found"

Step 4:- Then use else statement, if statement is not found in range then satisfy the below condition. Accept an argument and key of the element that element has to be searched

Step 5:- Initialize first to 0 and last to last element of the list. as array is starting from 0 hence it's initialized 1 less than the total count. Use for loop and assing the given range.

Step 6:- If statement in list and still the element to be searched is not found then find the middle element(m).

Step 7:- Repeat till you found the element stick the input and output of above algorithm.

PRACTICAL - 3

Aim :- Implementation of Bubble sort program on given list.

Theory : Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and they exist the wrong order.

This is the simplest form of sorting available In this we sort the given elements in ascending or descending order by comparing two adjecent elements at a time.

(i) Bubble sort algorithm start by comparing the first two element of an array and swapping if necessary

(ii) If we wont to sort the element of array in ascending order then first element is greater than second then I we need to swap the element.

(iii) If the element is smaller than second then we do not swap the element.

(iv) Again second and thirs elements are compared and swapped if it is necessary and this process go on until last and second last element is swopped and swapped.

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

print(a)
```

Output

```
Enter list: 65, 75, 30, 90, 2, 11
[2, 11, 30, 65, 75, 90]
```

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

6) Stick the output and input of above algorithm of bubble sort stepwise.

PRACTICAL-4

Aim :- Implement Quick sort to sort the given list

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

Algorithm.

1. Quick sort first selects a value, which is called pivot value, first element serve as our first pivot value. Since we know that first will eventually end up as last in that list

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 - lets call them left mark and right mark at the beginning and end of remaining items in the list. The goal of the partition process is to move items that are on wrong side with respect to pivot value while also converging on the split point.

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

```
def quick(alist):
    help(alist, 0, len(alist)-1)
def help(alist, first, last):
    if first < last:
        split = past(alist, first, last)
        help(alist, firs, split-1)
        help(alist, split+1, last)
def past(alist, first, last):
    print(alist[first])
    l = first+1
    r = last
    done = False
    while not done:
        while l<=r and alist[l]<=pivot:
            l = l+1
        while alist[r]>=pivot and r>=1:
            r = r-1
        if r<l:
            done = True
        else:
            t = alist[l]
            alist[l] = alist[r]
            alist[r] = t
    t = dlist[first]
    dlist[first] = alist[r]
    alist[r] = t
    return r
x = input("Enter range: ")
dlist = []
```

```
for b in range (0, x):
        b = input ("Enter element")
        alist. append (b)
        n = list (alist)
quick (alist)
print (alist)
```

Output.

```
Enter range ~~of~~ 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

than the pivot value. At this point we have discovered two items that are out of place with respect to eventual split point.

(5) At the point where right mark 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 (pivot value) is now in place.

(7) In addition all the items to left of split povit are less than PV and all the items to the right of split point are greater than PV. The list can now be divided at split point and quick sort can be invoked recursively on the two halves.

(8) The quicksort function invokes a recursive function, quicksorthelper.

(9) quick sorthelper, begins with some base called as merge sort

(10) If length of the list is less than or equal to ten, it is sorted.

(11) If it is greater, then it can be partitined and recursively sorted.

(12) The partition function implements the process described earlier.

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

Algorithm :

1) Create a class stack with instance vairable items

2) define the init method with self argument and initialize the initial value and the initialize to on empty list.

```
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 empty")
        else:
            a = self.i[self.tos]
            print("data =", a)

s = stack()
```

Output

```
>>> S.push(20)
>>> S.i
  [20, 0, 0, 0, 0]
>>> S.push(40)
>>> S.i
  [20, 40, 0, 0, 0]
>>> S.pop()
>>> Data = 40
>>> S.i
  [20, 0, 0, 0, 0]
>>> S.push(10)
>>> S.push(20)
>>> S.push(30)
>>> S.push(40)
>>> S.push(50)
>>> S.i
  [10, 20, 30, 40, 50]
>>> S.peek()
  Stack empty
```

S.peek()
O/P

03 01 02

3) Define methods push and 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 and initialize the value.

6) Posh 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 topmost position.

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

9) Assign the element values in push method and print the given values is poped or not

10) Attach the input and output of above algorithm.

PRACTICAL - 6

Aim: Implementing a Queue using Python list.

Theory: Queue is a linea data structure which has 2 references frond and rear Implementing a queue using Python list is the simplest as the python list provides inbuilt functions to perform the specified operations of the queue. It is based on the principal that o new element is inserted after rear and Element is inserted after rear and element of queue is deleted. which is at front. In simple terms, a queue can be described as o 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 (): 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 element.

```
class queue:
    global r
    global f
    global a
    def __init__(self):
        self.r=0
        self.f=0
        self.a=[0,0,0,0,0,0]
    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==self.n):
            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

```
>>> 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 element inserted :', 6)
>>> b.enqueue(7)
('queue element inserted :' 7)
>>> b.enqueue(8)
('queue element inserted :' 8)
>>> b.enqueue(9)
  queue is full
>>> b.dequeue()
  ('queue element deleted', 3)
>>> 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, 0]
```

```
-->>>Print(b.a)
[3, 4, 5, 6, 7, 8]
```

rear
front

Algorithm

Step1 : Define a class Queue and assign global vairables then define init() method with self argument in init() , assign or initialize the initial value with the help of self argument.

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

Step3 Use If statement that length is equal to zero then Queue is full or else insert the element in empty list or display that Queue element added successfully and increment by 1.

Step4 Define deQueue() with self-argument. Under this use if statment 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

Step5 Now call the Queue() function and give the element that has to be added in the empty list by using enQuence() and print the list after adding and some for deleting and display the list after deleting the element from the list.

Practical -7

Aim: Postfix Implementation

Theory:

The postfix expression is free of any parenthese. Further we took care of the priorities of the operators in the program. A given postfix expression can easily be evaluated using stacks. Reading the expression is always from left to right in Postfix.

Algorithm.

Step1: define evaluate as function then create a empty stack in Python.

Step2: Convert the string to a list by using the string method 'split'.

Step3: Calculate the length of String and print it.

Step4: Use for loop to assign the range of string then give condition using .If statement.

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

```
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] == '/':~~
        else:
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) / int(a))
    return stack.pop()
s = "5 8 3 - +"
r = evaluate(s)
Print("Evaluated value is:", r)
```

Output

>>> Evaluated value is: 10

17/01/2020

Step6: If the token is an operato *, /, +, -, ^ it will need two operands. Pop the 'p' twice. the first pop is second operand and the second pop is the first operand.

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

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

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

Step10 Attach output and input of above Algorithm.

Aim Implementation of Single linked list by adding the nodes from last node position.

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

Algorithm:

Step1: Transversing of a linked list means visiting all nodes in the linked list in order to perform some operation on them.

Step2: The entire linked list ~~mean~~ can be accessed use 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.

Step3: Thus, the entire linked list can be transversed using the node which is referred by the head pointer of the linked list.

52

```
class node:
    global date
    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:
        new node =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)

Start=linkedlist()
```

```
def delete(self):
    if self.s == None:
        print("list is empty")
    else:
        head = self.s
    while True:
        if head.next != None:
            d = head
            head = head.next
        else:
            d.next = None
            break.
```

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

Step4: Now that we know that we can transverse the entire linked list using the head pointer, we shoud only use it to refer the first node of list only.

Step5 : 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 charges which we ~~convert~~ cannot revert back.

Step 6: We may lose the referace to the 1st node in our linked list, and hence most of our linked list. So in order to avoid making some unwanted charges to the 1st node, we will use a temporary node to transverse the entire linked list.

Step 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 node the data type of the temporary node should also be Node.

Output.

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

```
20
30
40
50
60
70
80
```

24/01/25

Step8: 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.

Step9: But the 1st node is referred by current. So we can transverse to 2nd nodes as h = h.next

Step10: Similarly, we can transverse rest of nodes in the linked list using same method by while loop.

Step11: Our concern now is to find terminating condition for the while loop.

Step12: The last node in the linked list is referred by the tail of linked list. Since the last node of linked does not have any next node, the value in the next field of the last node is None.

Step 13: So we can refer the last node of linked list self.s = None.

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

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

Practical-9

Aim: Program based on Binary Search tree by Implementing Inorder, Preoder and Postorder transversal.

Theory: Binary Tree is a tree with support maximum of 2 children from 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 is ordered such that one child is identified as left child and other as right child.

→ Inorder (i) Transverse the left subtree, The left subtree might have left and right subtrees.
(ii) Visit the root node.
(iii) Transverse the right subtree and repeat it.

→ Preorder: (i) Visit the root node
(ii) Traverse the left subtree. The left subtree in turn might have left and right subtree.
(iii) Traverse the right subtree repeat it.

→ Post order: (i) Traverse the left subtree. the left subtree in turn might have left and right subtrees.
(ii) Traverse the right subtrees.
(iii) Visit the root node.

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

class BST:
    def __init__(self):
        self.root = None
    def add(self, value):
        p = node(value)
        if self.root == None:
            self.root = p
            print("Root is added successfully", p.val)
        else:
            h = self.root
            while True:
                if p.val < h.val:
                    if h.left == None:
                        h.left = p
                        print(p.val, "Node is added to
                            leftside successfully at",
                                                h.val)
                        break
                    else:
                        h = h.left
                else:
                    if h.right == None:
                        h.right = p
                        print(p.val, "Node is successfully
                            added to rightside 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:
        Postorder (root.left)
        Postorder (root.right)
        print (root.val)
t = BST ()
```

Output

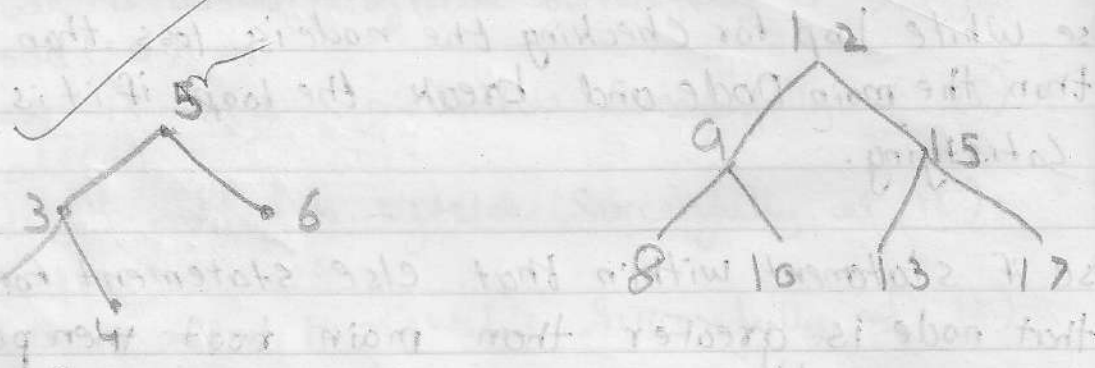
```
>>> t.add (25)
('Root is added Successfully', 25)
>>> t.add (15)
(15, 'Node is added to leftside Successfully at', 25)
>>> t.add (50)
(50, 'Node is added to rightside Successfully at', 25)
>>> t.add (10)
(10, 'Node is added to leftside Successfully at', 15)
>>> t.add (22)
(22, 'Node is added to rightside Successfully at', 15)
>>> t.add (35)
(35, 'Node is added to leftside Successfully at', 50)
>>> t.add (70)
(70, 'Node is added to rightside Successfully at', 50)
>>> t.add (4)
(4, 'Node is added to leftside Successfully at', 10)
```

• Algorithm :

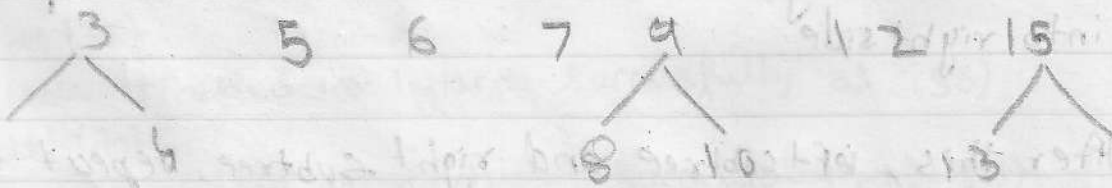
- Define class node and define init() method with 2 argument. Initialize the value in this method.
- Again, Define a class BST that is Binary Search Tree with init() method with self argument and assign the root as None.
- Define add() method for adding the node. Define a variable p that p = node(value)
- Use If Statement for checking the condition that root is none then use else statement for ; if node is less than the main node then put or arrange that in leftside.
- Use while loop for checking the node is less than or greater than the main node and break the loop if it is not satisfying.
- Use if statement within that else statement for checking that node is greater than main root then pot it into right side.
- After this, left subtree and right subtree, repeat this method to arrange the node according to Binary Search Tree

- Define Inorder() . Preorder() and Postorder() with root argument and use if statement that root is none and return that in all.
- In Inorder, else statement used for giving that condition first left, root and then right node
- For Preorder, We have to give condition in else that first root, left and the right node.
- For Postorder, In else part, assign left then right and then go for root node
- Display the output and input of the Above Algorithm.
- Inorder: (LVR)

Step1



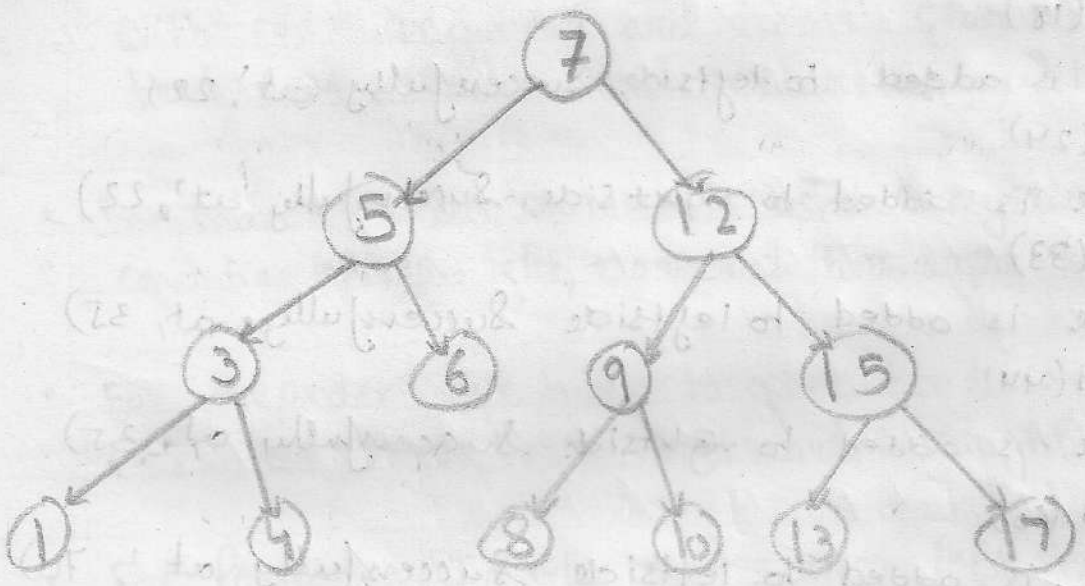
Step2



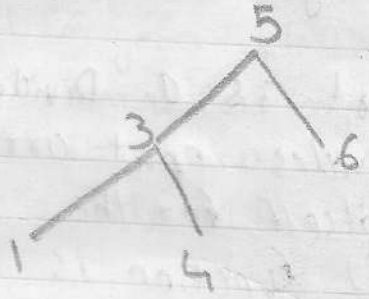
Step3 1 3 4 5 6 7 8 9 10 11 12 13 15 17

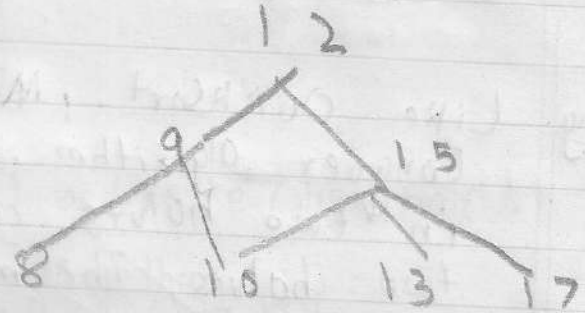
```
>>> t.add(12)
(12, 'Node is added to rightside Successfully at', 10)
>>> t.add(18)
(18, 'Node is added to leftside Successfully at', 22)
>>> t.add(24)
(24, 'Node is added to rightside Successfully at', 22)
>>> t.add(33)
(33, 'Node is added to leftside Successfully at', 35)
>>> t.add(44)
(44, 'Node is added to rightside Successfully at', 35)
>>> t.add(60)
(60, 'Node is added to leftside Successfully at', 70)
>>> t.add(90)
(90, 'Node is added to rightside Successfully at', 70)
```

* Binary Search Tree



Preorder: (VLR)



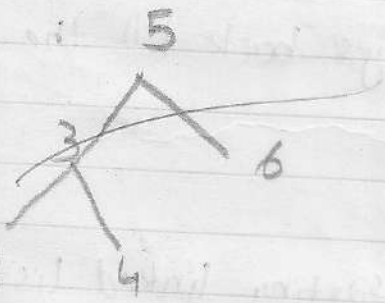


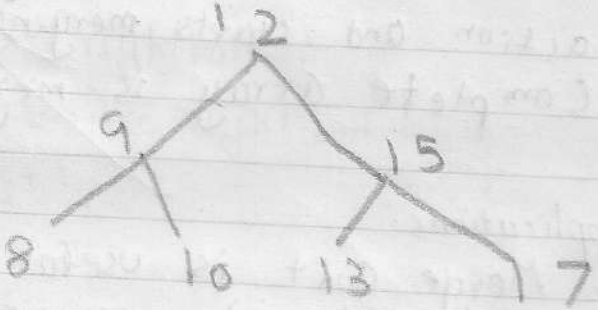
7 5 3 6 12

9 15

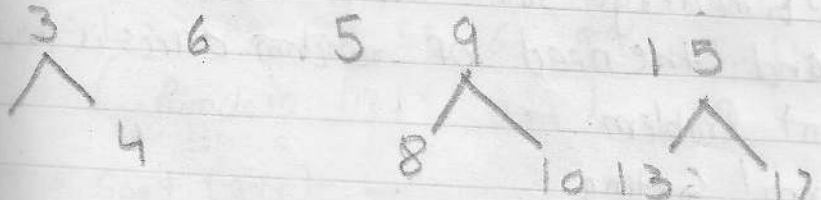
7 5 3 1 4 6 12 9 8 10 15 13 17

Postorder: (LRV)





3 6 5 9 15 12 7



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

Practical 10

Aim To sort a list using Merge sort

Theory Like QuickSort, MergeSort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself, for the two halves, The merge() function is used for merging two halves.

~~That~~ 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 and starts merging arrays back till the complete array is merged.

Applications:

1. Merge Sort is useful for sorting linked lists in O(n log n) time. Merge Sort access data subsequently and the need of random access is low
2. Inversion Count Problem
3. Used in External Sorting.

Merge Sort is more efficient than quick Sort - for some types of lists if the data to be sorted can be

```
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("\nMERGESORT LIST :", arr)
```

Output

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

MERGESORT LIST : [10, 14, 27, 45, 55, 62, 70, 89, 99]

only efficiently accessed sequentially and thus is popular where sequentially accessed data structure are very common.

Practical - 11

Aim : To demonstrate the use of circular queue

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

When we dequeue any element to remove it from the queue, we are actually moving the front ~~fro~~ of the queue forward, thereby reducing the overall size of the queue. And we cannot insert new elements, because the rear pointer is still at the end of the queue. The only way is to reset the linear queue for a fresh start.

Circular Queue is also a linear data structure, which follows the principle of FIFO, but instead of ending the queue at the last option, it again starts from the first position after the last

```
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=s

q=Queue()
```

Output

```
>>> q.add(100)
Data added: 100
>>> q.add(200)
Data added: 200
>>> q.add(300)
Data added: (300)
>>> q.remove()
Data removed: 100

>>q.l
[0, 200, 300, 0, 0, 0]
```

Initially, the head and the tail will be pointing to the same locality 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 to tail pointer is incremented to put the next available location.

Application

1) Computer controlled Traffic Signaling System uses Circular queue.

2) CPU scheduling and Memory Management

PTB
14/02/2020