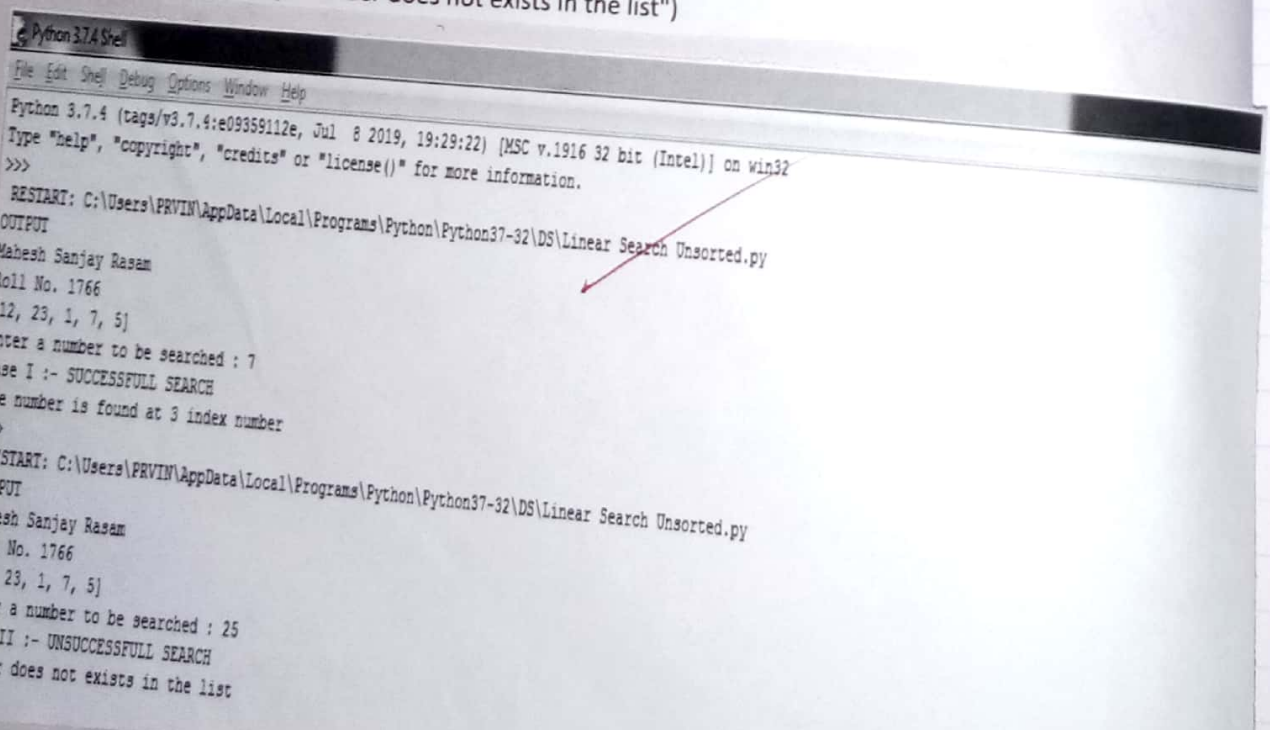
Linear Search (Unsorted)

Source Code :-

```
found=False
a=[12,23,1,7,5]
print("Mahesh Sanjay Rasam")
print("Roll No. 1766")
print(a)
search=int(input("Enter a number to be searched : "))
for x in range(len(a)):
  if(search==a[x]):
    print("Case I :- SUCCESSFULL SEARCH")
    print("The number is found at",x,"index number")
    found=True
    break
if(found==False):
  print("Case II :- UNSUCCESSFULL SEARCH")
  print("Number does not exists in the list")
```



Practical - 1

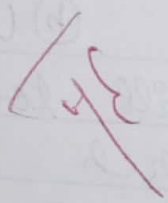
Linear Search (unsorted)

Aim: To search a number from the list using linear search (unsorted).

Theory: The process of identifying or finding a random elements from list is called searching.
There are 2 types of search
① Linear Search
② Binary Search
The linear search is further classified as :-
ⓐ SORTED ⓑ UNSORTED
Here, we will look on the unsorted Linear Search
Linear Search is also known as sequential search, is a process that checks for every element of list linearally or sequentially until the search is found.
When the elements of list are not arranged in ascending or descending order ie they are in random manner then the search will be called as unsorted linear search.

Unsorted linear Search

1. The elements of list will be in rand manner. that is in unsorted manner.
2. The best case will be at 0 index number.
3. The worst case or search will be at n index number.
4. The elements of list are searched one after other ie from 0 index to n inde until the search is found.



Linear Search (Sorted)

Source Code :-

```
found=False
a=[12,45,67,85,90]
print("Mahesh Sanjay Rasam")
print("Roll No. 1766")
print(a)
search=int(input("Enter a number to be searched : "))
if(search<a[0] or search>a[len(a)-1]):
  print("Case III")
  print("Number out of range")
else:
  for x in range(len(a)):
    if(search==a[x]):
      print("Case I :- SUCCESSFUL SEARCH")
      print("The number is found at",x,"index number")
      found=True
      break
if(found==False):
  print("Case II :- UNSUCCESSFUL SEARCH")
  print("Number not found in the list")
```

Practical - 2
Linear Search (Sorted)

Aim :- To search a number from the list using linear search.

Theory :-

A linear search or sequential search is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched. A linear search runs in at worst linear time and makes at most n comparisons, where n is the length of the list. If each element is equally likely to be searched, then linear search has an average case of n/2 comparisons, but the average case can be affected if the search probabilities for each element vary. The performance of linear search improves if the desired value is more likely to be near the beginning of the list than to its end. But the disadvantage of the linear search is that if there are 1000 elements than and if we want the last element it has to run 1000 times.

Sorted Linear Search

1. The elements of the list must be in sorted manner ie. must be in ascend or be in descending order.
2. The best case will be at 0 index number.
3. But, in sorted linear search firstly we check that the number which has to be searched must be greater than or eq to number present at 0 index and must be less than or equal to number present at n index.

Python 3.7.4 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Users\PRVIN\AppData\Local\Programs\Python\Python37-32\DS\Linear Search Sorted.py
OUTPUT
Mahesh Sanjay Rasam
Roll No. 1766
[12, 45, 67, 85, 90]
Enter a number to be searched : 85
Case I :- SUCCESSFUL SEARCH
The number is found at 3 index number
>>>
 RESTART: C:\Users\PRVIN\AppData\Local\Programs\Python\Python37-32\DS\Linear Search Sorted.py
OUTPUT
Mahesh Sanjay Rasam
Roll No. 1766
[12, 45, 67, 85, 90]
Enter a number to be searched : 50
Case II :- UNSUCCESSFUL SEARCH
Number not found in the list
>>>
 RESTART: C:\Users\PRVIN\AppData\Local\Programs\Python\Python37-32\DS\Linear Search Sorted.py
OUTPUT
Mahesh Sanjay Rasam
Roll No. 1766
[12, 45, 67, 85, 90]
Enter a number to be searched : 7
Case III
Number out of range
>>>
```

Binary Search

Source Code:-

```
a=[12,25,34,56,82]
print("Mahesh Sanjay Rasam")
print("Roll No. 1766")
print(a)
search=int(input("Enter a number to be searched : "))
lb=0
ub=len(a)-1
while(True):
  m=(lb+ub)//2
  if(lb>ub):
    print("Case-II :- UNSUCCESSFULL SEARCH")
    print("Number does not exist in the list")
    break
if(search==a[m]):
  print("Case-I :- SUCCESSFULL SEARCH")
  print("Number is found at",m,"index number")
  break
else:
  if(search<a[m]):
    ub=m-1
  else:
    lb=m+1
```

Practical - 3

Binary Search

Aim : To search a element from the list using Binary Search technique.

Theory :

A binary search also known as half-interval search is an algorithm used in programming languages to locate a specified value (key) within an list. For the search to be binary the list elements must be in sorted manner that is either in ascending or in descending manner.

At each step of the algorithm a comparision is made and the procedure branches into one of two directions.

Firstly in Binary search it compares with the middle element of the list if it is found it will print the index number or if not it will checks that the number to be searched is greater or less than middle element of the list. If less it will search or go left of the middle element or if greater it will search right element

of the middle element.

Binary Search

1. The list must be sorted either be in ascending or in descending manner
2. The best search or case will be len(list)//2.
3. If number is less than middle element it will search to the left of the list.
4. If number is greater than middle element it will search to the right of the list.

Python 3.7.4 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Users\PRVIN\AppData\Local\Programs\Python\Python37-32\DS\Binary Search.py
OUTPUT
Mahesh Sanjay Rasam
Roll No. 1766
[12, 25, 34, 56, 82]
Enter a number to be searched : 82
Case-I :- SUCCESSFULL SEARCH
Number is found at 4 index number
>>>
 RESTART: C:\Users\PRVIN\AppData\Local\Programs\Python\Python37-32\DS\Binary Search.py
OUTPUT
Mahesh Sanjay Rasam
Roll No. 1766
[12, 25, 34, 56, 82]
Enter a number to be searched : 5
Case-II :- UNSUCCESSFULL SEARCH
Number does not exist in the list
>>>
```

Stack

Source Code :-

```
print("Mahesh Rasam")

class stack:

  global tos


  def __init__(self):

    self.l=[0,0,0,0,0,0,0]

    self.tos=-1

  def push(self,data):

    n=len(self.l)

    if self.tos==n-1:

      print("Stack is full")

    else:

      self.tos=self.tos+1

      self.l[self.tos]=data

def pop(self):

  if self.tos<0:

    print("Stack is empty")

  else:

    k=self.l[self.tos]

    print("Data=",k)

    self.tos=self.tos-1

s=stack()

s.push(11)

s.push(22)

s.push(33)
```

Practical - 4

Stack

Aim :- To implement working of Stacks.

Theory :-

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First in Last Out). There are many real-life examples of a Stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e the plate which has been placed at the bottomost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO (Last In First Out) / FILO (First In Last Out) order.

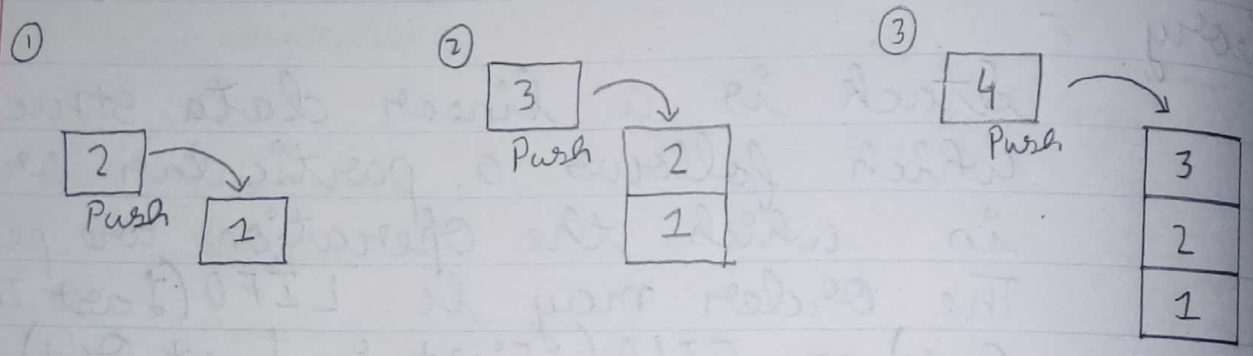
A stack is an abstract data type that serves as a collection of elements, with two principal operations:

(a) push, which adds an element to the collection.

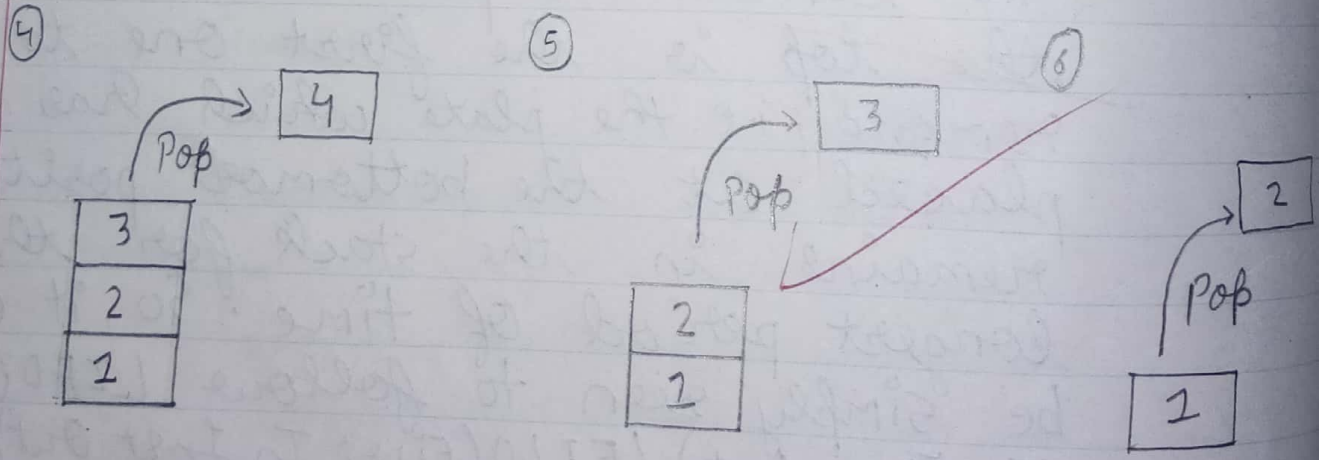
(b) pop, which removes the most

recently added element that was not yet removed

① Push operation



② Pop operation



Stack

Source Code :-

```
s.push(44)
s.push(55)
s.push(66)
s.push(77)
s.push(88)
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
s.pop()
```

Output :-

```
Mahesh Rasam
Stack is full
Data= 77
Data= 66
Data= 55
Data= 44
Data= 33
Data= 22
Data= 11
Stack is empty
```

Queue

Source Code:-

```
print("Mahesh Rasam")

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

      self.r=self.r+1

    else:

      print("Queue is full")

 def remove(self):

   n=len(self.l)

   if self.f<n-1:

     print(self.l[self.f])

     self.f=self.f+1

   else:

     print("Queue is empty")

Q=Queue()

Q.add(11)

Q.add(22)
```

Practical - 5

Queue

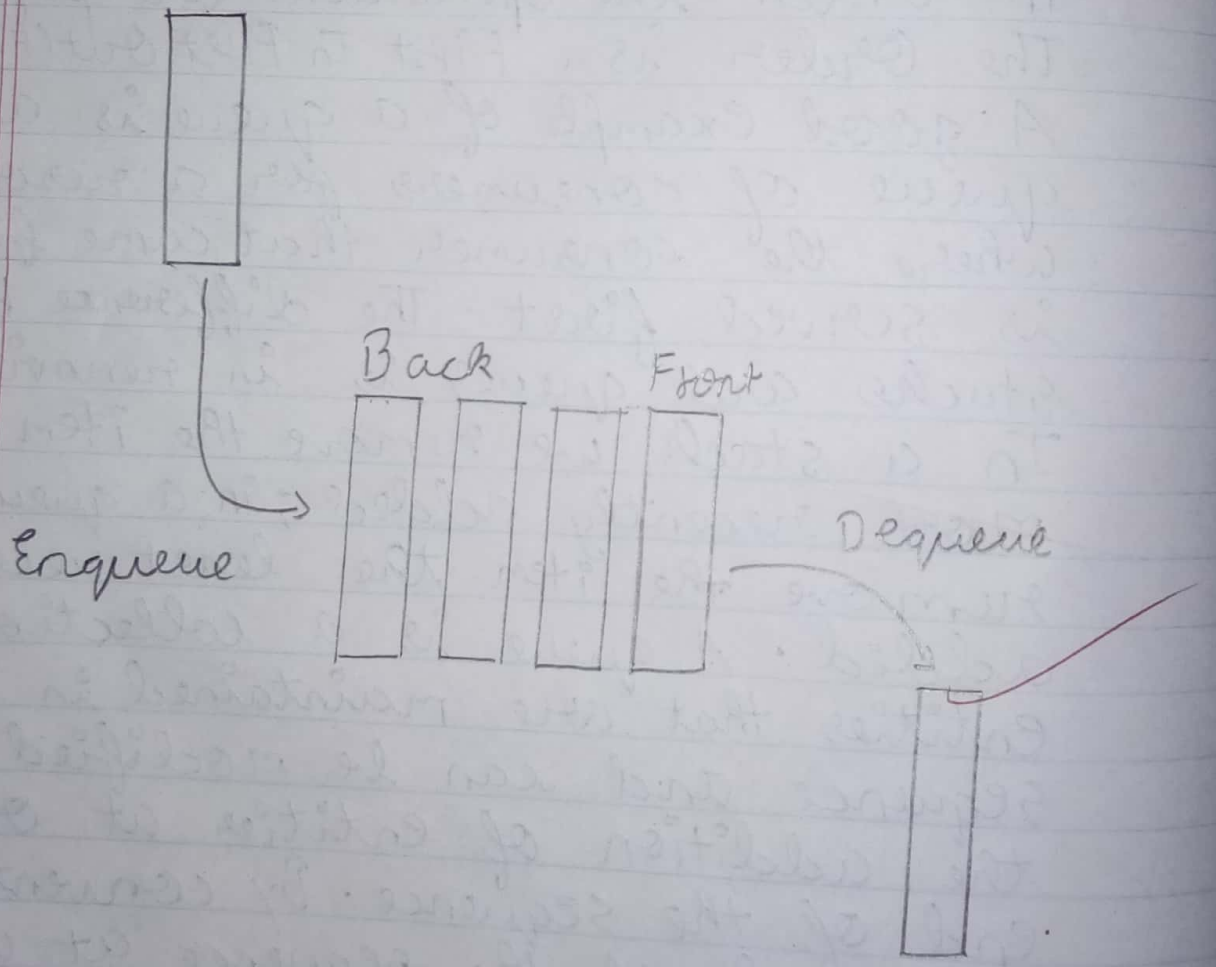
Aim :- To add and delete items from Queue

Theory :-

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added. A queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence. By convention, the end of the sequence at which elements are added is called the back or rear of the queue and the end at which elements are

removed is called the head or front of the queue, analogously to the words used when people line up to wait for goods or services.

Queue



Queue

Source Code:-

```
Q.add(33)

Q.add(44)

Q.add(55)

Q.add(66)

Q.remove()

Q.remove()

Q.remove()

Q.remove()

Q.remove()

Q.remove()
```

Output:-

```
Mahesh Rasam

Queue is full

11

22

33

44

55

Queue is empty
```

Circular Queue

Source Code :-

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

Practical - 6 - Circular Queue.

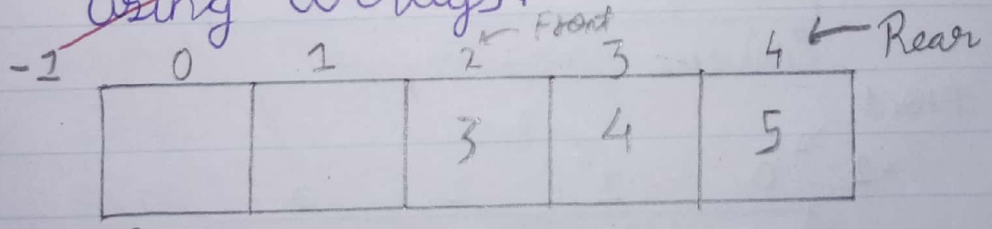
Aim :- To demonstrate the use of circular queue in data - structure.

Theory :-

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.

...en Queue (value) This function is used to insert an element into the circular queue.

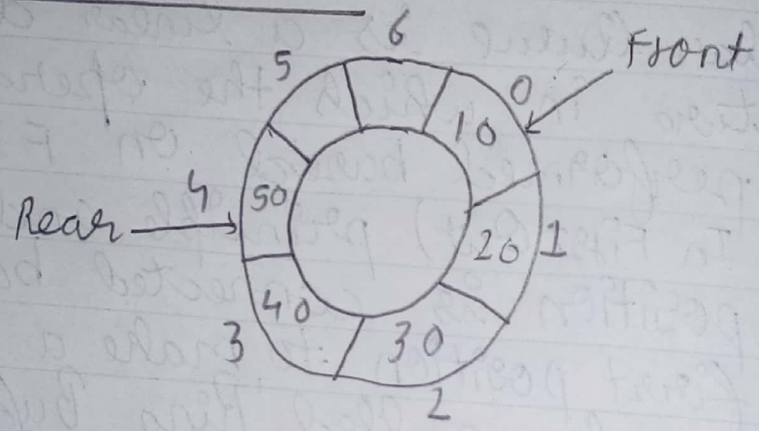
Circular queue avoids the wastage of space in a regular queue implementation using arrays.



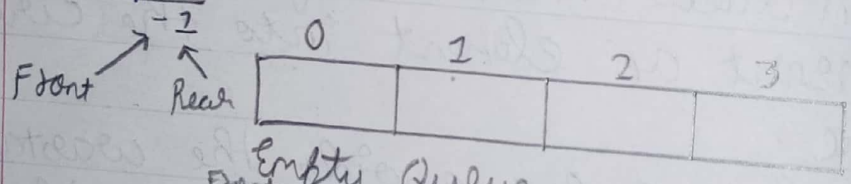
Circular Queue works by the process of circular increment ie when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

The most common queue implementation is using arrays, but it can also be implemented using lists.

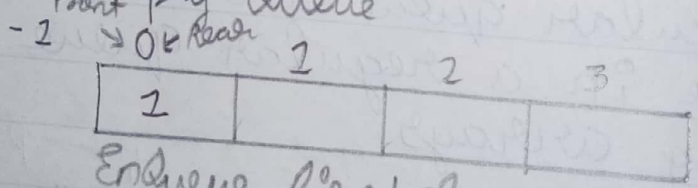
Circular Queue



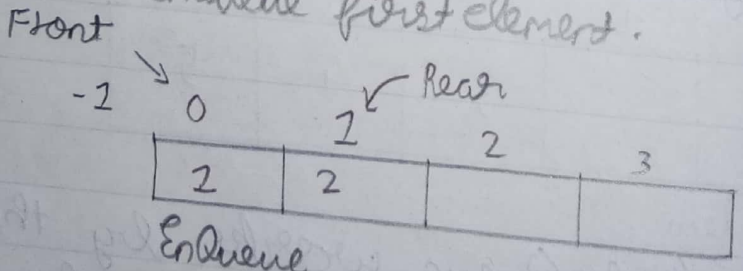
Example :-



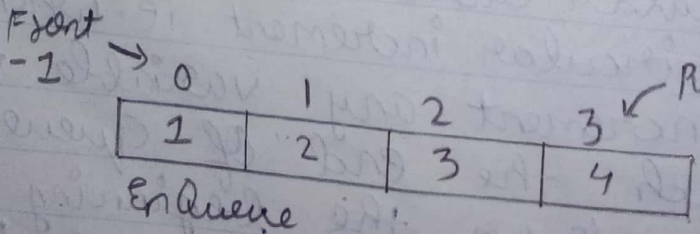
Empty Queue



EnQueue first element.



EnQueue



EnQueue

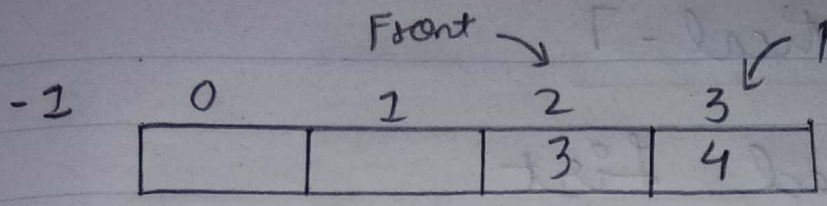
Circular Queue

Source Code :-

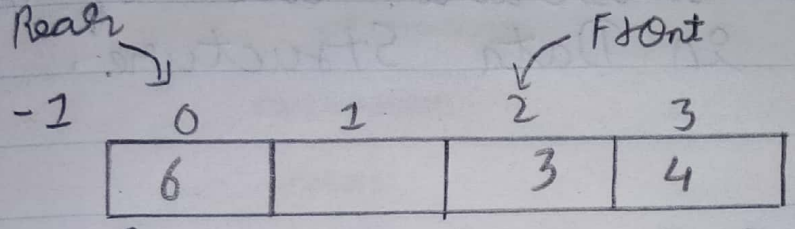
```
        if self.f<=n-1:
            print("Data removed:",self.l[self.f])
            self.f=self.f+1
        else:
            s=self.f
            self.f=0
            if self.f<self.r:
                print(self.l[self.f])
                self.f=self.f+1
            else:
                print("Queue is empty")
                self.f=s
Q=Queue()
Q.add(45)
Q.add(55)
Q.add(65)
Q.add(75)
Q.add(85)
Q.add(95)
Q.remove()
Q.add(65)
```

Output :-

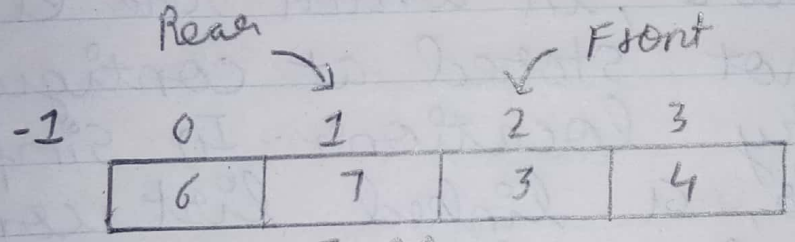
```
Data added: 45
Data added: 55
Data added: 65
Data added: 75
Data added: 85
Data added: 95
Data removed: 45
```



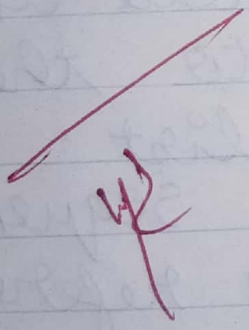
DeQueue



EnQueue



Queue Full



Practical - 7

Linked List

Aim :- To demonstrate use of Linked List in Data Structure.

Theory :-

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list. Linked list uses a sequence of nodes with the reference or pointer to indicate the next node in the list. One disadvantage of using arrays to store data is that arrays are static structure and therefore cannot be easily extended or reduced to fit the data set. A linked list is a linear data

Source Code :-

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

```
    def display(self):
        head=self.s
        while head.next!=None:
            print(head.data)
            head=head.next
        print(head.data)
start=linkedlist()
start.addL(55)
start.addL(66)
start.addL(77)
start.addL(88)
start.addB(33)
start.addB(22)
start.addB(11)
start.display()
print("Mahesh Rasam")
```

Output :-

11

22

33

55

66

77

88

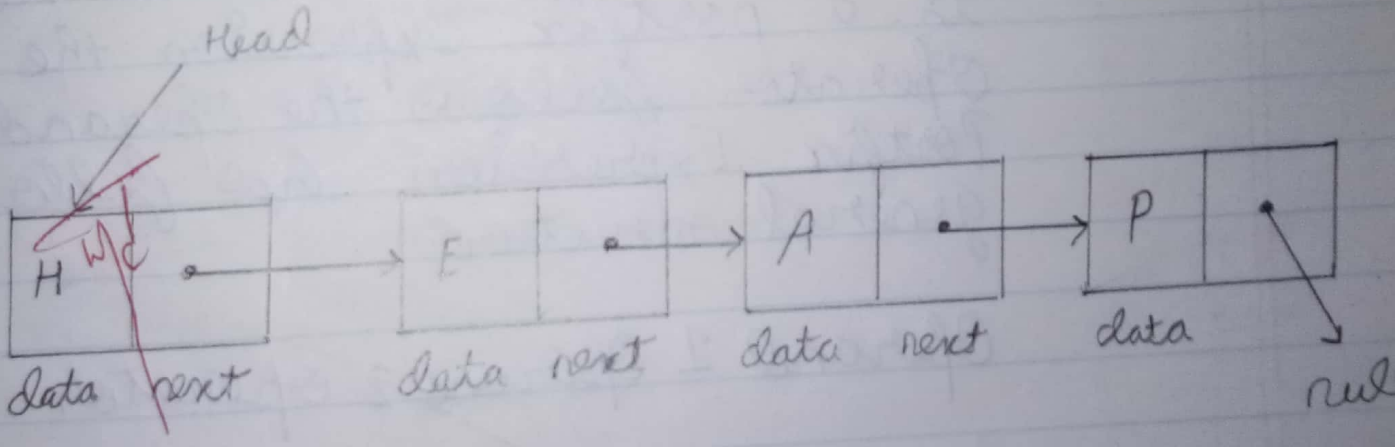
Mahesh Rasam

data structure where each element is a separate object

TYPES Of Linked List

1. Singly linked list
2. Doubly linked list
3. Circular linked list
4. Priority linked list

Presentation of linked list



Practical - 8

Postfix Evaluation

Aim :- To demonstrate working of Postfix Evaluation in Data Structure.

Theory :-

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands Postfix Expression has following general structure

Operand 1 Operand 2 operator

Eg:

a b +

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps:

Postfix Evaluation

Source Code :-

```
def evaluate(s):
    k=s.split()
    n=len(k)
    stack=[]
    for i in range(n):
        if k[i].isdigit():
            stack.append(int(k[i]))
        elif k[i]=='+':
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)+int(a))
        elif k[i]=='-':
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)-int(a))
        elif k[i]=='*':
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)*int(a))
        else:
            a=stack.pop()
            b=stack.pop()
            stack.append(int(b)/int(a))
```

```
    return stack.pop()

s="2 5 8 * +"

r=evaluate(s)

print("Mahesh Rasam")

print("The evaluated value is:",r)
```

Output :-

Mahesh Rasam

The evaluated value is: 42

Steps :

① Read all the symbols one by one from left to right in the given Postfix Expression
② If the reading symbol is operand, then push it on to the stack
③ If the reading symbol is operator (+, -, *, / etc) then perform TWO pop operations and store the two popped operand in two different variables (operand 1 and operand 2). Then perform reading symbol operation using operand 1 and operand 2 and push result back on to the stack.
④ Finally, perform a pop operation and display the popped value as final result.

Practical - 9

Bubble Sort

Aim :- To demonstrate working of Bubble Sort in Data Structures.

Theory :-

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong. Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Worst complexity - n^2

Average complexity - n^2

Best complexity - n

The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

#Bubble Sort#

Source Code :-

```
print("Mahesh Rasam")
a=[50,14,3,7,90]
print(a)
for i in range(len(a)-1):
    for j in range(len(a)-1):
        if(a[j]>a[j+1]):
            t=a[j]
            a[j]=a[j+1]
            a[j+1]=t
print(a)
```

Output :-

Mahesh Rasam

[50, 14, 3, 7, 90]

[3, 7, 14, 50, 90]

This simple algorithm performs poorly in real world use and is used primarily as an educational tool. More efficient algorithms such as timsort, or merge sort are used by the sorting libraries built into popular programming languages such as Python and Java.

Working

5	1	12	-5	16	Unsorted
5	1	12	-5	16	5 > 1, swap } Pass 1
1	5	12	-5	16	5 < 12, OK
1	5	12	-5	16	12 > -5, swap
1	5	-5	12	16	12 < 16, OK
1	5	-5	12	16	1 < 5, OK } Pass 2
1	5	-5	12	16	5 > -5, swap
1	-5	5	12	16	5 < 12, OK
1	-5	5	12	16	1 > -5, swap } Pass 3
-5	1	5	12	16	1 < 5, OK
-5	1	5	12	16	-5 < 1, OK } Pass 4
-5	1	5	12	16	SORTED

Practical - 10

Selection Sort

Aim : To demonstrate working of Selection Sort in data structures.

Theory :

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. It has an $O(n^2)$ time complexity, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort.

Worst complexity : n^2
Average complexity : n^2
Best complexity : n^2

Selection Sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

#Selection Sort#

Source Code :-

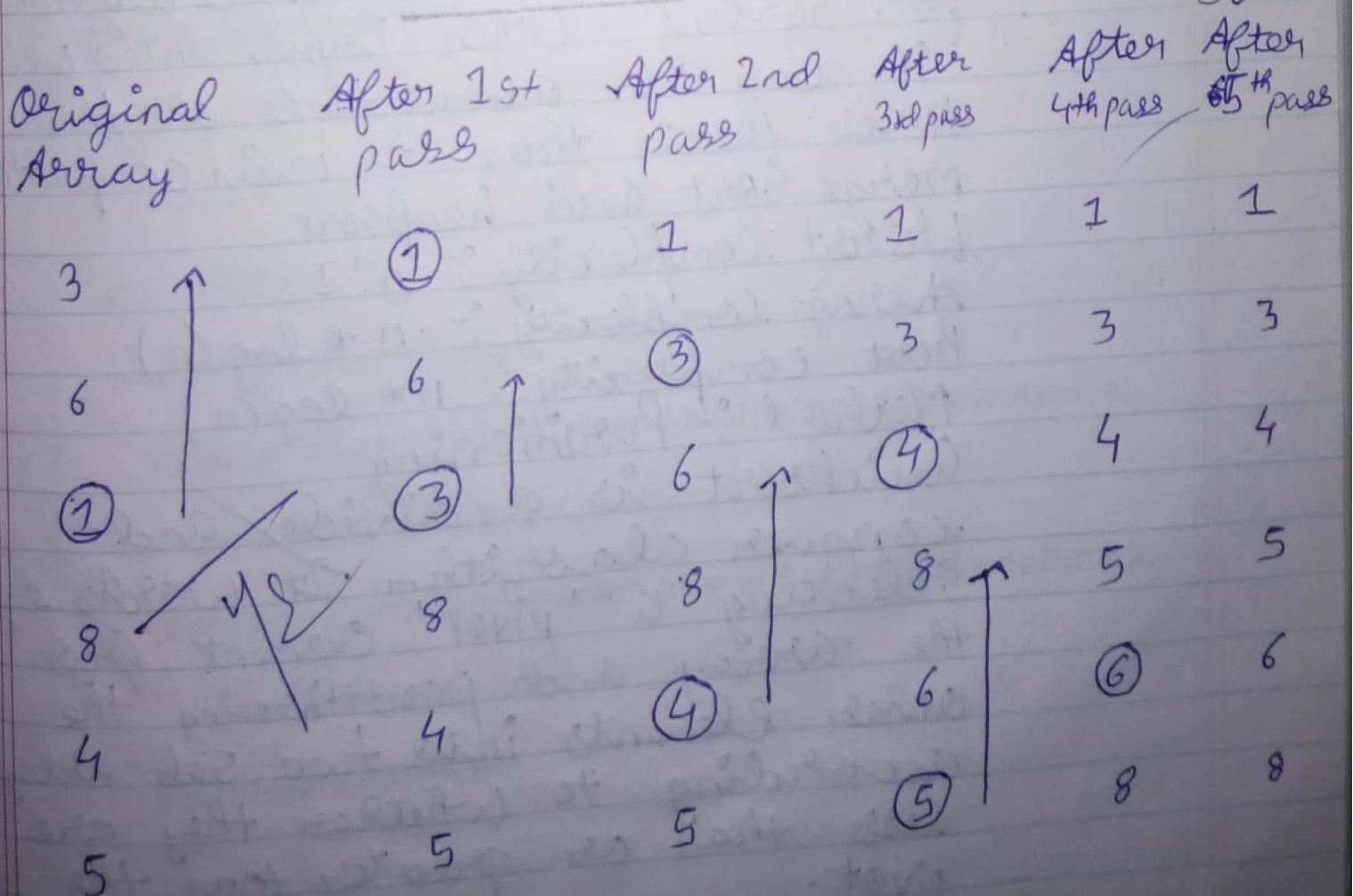
```
print("Mahesh Rasam")
a=[55,95,15,20,45,10]
print(a)
for i in range(len(a)-1):
    for j in range(len(a)-1):
        if(a[j]>a[i+1]):
            t=a[j]
            a[j]=a[i+1]
            a[i+1]=t
print(a)
```

Output :-

```
Mahesh Rasam
[55, 95, 15, 20, 45, 10]
[10, 15, 20, 45, 55, 95]
```

The time efficiency of selection sort is quadratic; so there are a number of sorting techniques which have better time complexity than selection sort. One thing which distinguishes selection sort from other sorting algorithms is that it makes the minimum possible number of swaps, n-1 in the worst case. It compares a element with every element of list and sorts accordingly.

Working



Practical - 11

Quick Sort

Aim :- To demonstrate use of Quick Sort in data structure

Theory :-

Quicksort is an efficient sorting algorithm Developed by British computer Scientist Tony Hoare in 1959. and published in 1961, it is still a commonly used algorithm for sorting. When implemented well it can be about two or three times faster than its main competitors merge sort and heapsort.

Worst complexity :- n^2
Average complexity :- $n * \log(n)$
Best complexity :- $n * \log(n)$
Method :- Partitioning

Quicksort is a divide and conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot.

#Quick Sort#

Soruce Code :-

```
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)
def quickSortHelper(alist,first,last):
   if first<last:
      splitpoint=partition(alist,first,last)
      quickSortHelper(alist,first,splitpoint-1)
      quickSortHelper(alist,splitpoint+1,last)
def partition(alist,first,last):
   pivotvalue=alist[first]
   leftmark=first+1
   rightmark=last
   done=False
   while not done:
      while leftmark<=rightmark and alist[leftmark]<=pivotvalue:
         leftmark=leftmark+1
      while alist[rightmark]>=pivotvalue and rightmark>=leftmark:
         rightmark=rightmark-1
      if rightmark<leftmark:
```

```
            done=True
        else:
            temp=alist[leftmark]
            alist[leftmark]=alist[rightmark]
            alist[rightmark]=temp
    temp=alist[first]
    alist[first]=alist[rightmark]
    alist[rightmark]=temp
    return rightmark
alist=[47,5,54,7,89,65,55,80,90]
quickSort(alist)
print(alist)
```

Output :-

Mahesh Rasam

[5, 7, 47, 54, 55, 65, 80, 89, 90]

The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.
Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved.

Working

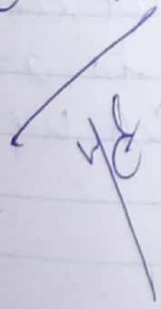
(a) 2 8 7 1 3 5 6 [4] — Pivot

(b) 2 1 [3] 4 7 5 6 [8] — Pivot, Pivot

(c) 2 [1] 3 4 7 5 [6] 8 — Pivot, Pivot

(d) 1 2 3 4 [5] 6 7 8 — Pivot

(e) 1 2 3 4 5 6 7 8



Practical - 12

Binary Tree and Traversal

Aim :- To demonstrate working of Binary Tr and Traversal

Theory :-

Traversing in the Binary Tree - Tree traversal is the process of visiting each node in the tree exactly once Visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traver Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edg (links) we always start from the root (head) node. That is, we cann randomly access a node in a tr There are three ways which we u to traverse a tree -

1. In - order Traversal
2. Pre - order Traversal
3. Post - order Traversal

Binary Tree and Traversal

Source Code :-

```
class Node:
  global r
  global l
  global data
  def __init__(self,l):
    self.l=None
    self.data=l
    self.r=None
class Tree:
  global root
  def __init__(self):
    self.root=None
  def add(self,val):
    if self.root==None:
      self.root=Node(val)
    else:
      newnode=Node(val)
      h=self.root
      while True:
        if newnode.data<h.data:
          if h.l!=None:
            h=h.l
```

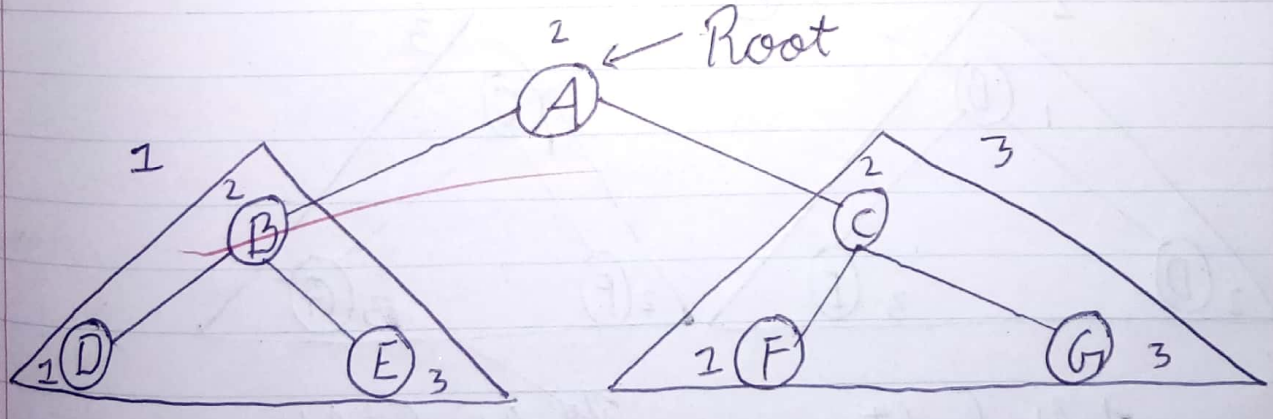
```
                else:
                    h.l=newnode
                    print(newnode.data,"added on left of",h.data)
                    break
            else:
                if h.r!=None:
                    h=h.r
                else:
                    h.r=newnode
                    print(newnode.data,"added on right of",h.data)
                    break
    def preorder(self,start):
        if start!=None:
            print(start.data)
            self.preorder(start.l)
            self.preorder(start.r)
    def inorder(self,start):
        if start!=None:
            self.inorder(start.l)
            print(start.data)
            self.inorder(start.r)
    def postorder(self,start):
        if start!=None:
            self.inorder(start.l)
            self.inorder(start.r)
```

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

→ In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



Left Subtree

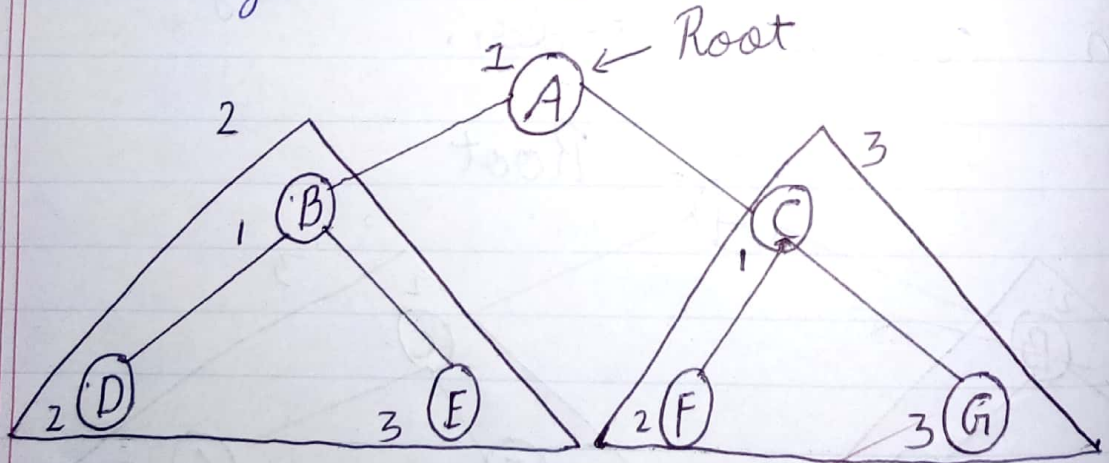
Right Subtree

We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in. The process goes on until all the nodes are visited. The output of inord traversal of this tree will be -

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

→ Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



Left Subtree Right Subtree

We start from A, and following pre-o traversal, we first visit A itself and th move to its left subtree B. B is also traversed pre-order. The process goes o

```
        print(start.data)

T=Tree()

T.add(100)

T.add(80)

T.add(70)

T.add(85)

T.add(10)

T.add(78)

T.add(60)

T.add(88)

T.add(15)

T.add(12)

print("preorder")

T.preorder(T.root)

print("inorder")

T.inorder(T.root)

print("postorder")

T.postorder(T.root)

print("Mahesh Rasam")
```

Output :-

80 added on left of 100

70 added on left of 80

85 added on right of 80

10 added on left of 70

78 added on right of 70

60 added on right of 10

88 added on right of 85

15 added on left of 60

12 added on left of 15

preorder

100

80

70

10

60

15

12

78

85

88

inorder

10

12

15

60

70

78

80

85

88

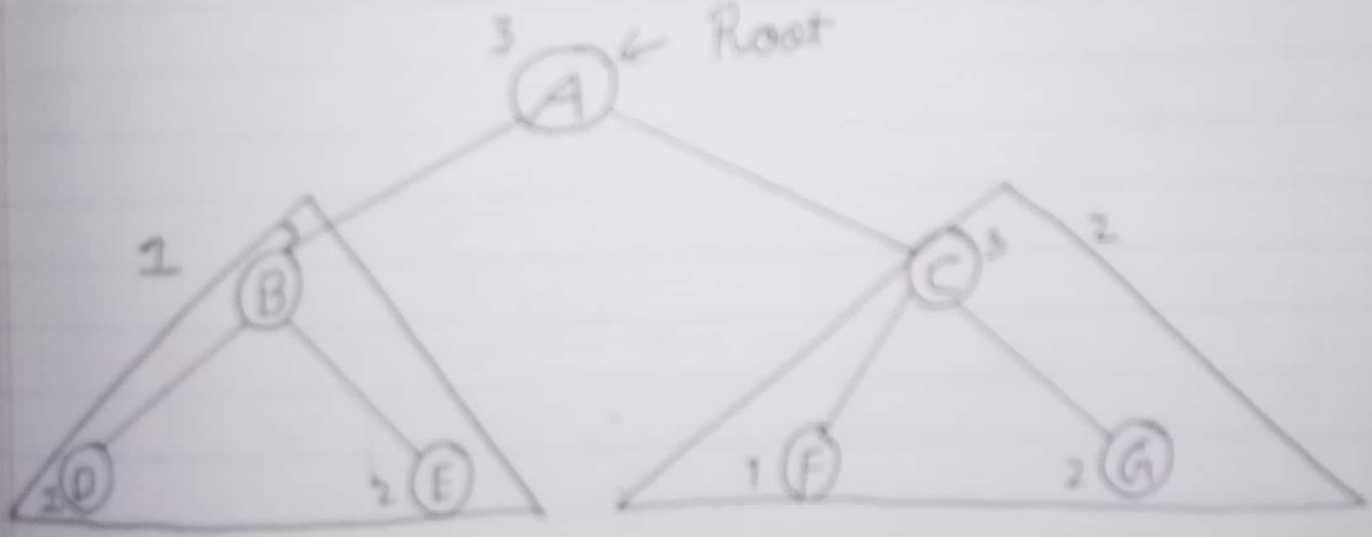
100

until all the nodes are visited. The output of pre-order traversal of this tree will be -

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Post-Order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



Left Subtree
Right Subtree

We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of

post-order traversal of this tree will be-

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

postorder

10

12

15

60

70

78

80

85

88

100

Mahesh Rasam

##Merge Sort##

Source Code :-

```
print("Mahesh Rasam\n")
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
```

Practical -13

Merge Sort

Aim :- To implement working of Merge Sort.

Theory :-

Merge Sort is a sorting technique bared on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms. Merge Sort first divides the array into equal halves and then combines them in a sorted manner. Merge Sort first divides the array into equal halves and then combines them in a sorted manner. Merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here this does not changes the sequence of appearance of items in the original. Merge Sort keeps on dividing the list into equal halves until it can no more be divided. If it is only one element in the list, it is sorted. Then merge sort combines the smaller sorted lists keeping the new list sorted too.

```
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=[31,12,7,82,11,96,14,41,20]
print("Original List is : ",arr)
mergesort(arr)
print("\nSorted List is : ",arr)
```

Output :-

Mahesh Rasam

Original List is : [31, 12, 7, 82, 11, 96, 14, 41, 20]

Sorted List is : [7, 11, 12, 14, 20, 31, 41, 82, 96]