☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member's Signature
1.	Implement linear search to find an item in list	33	29/11/19	29/11/19
2.	Implement binary search to find a searched number in the list	37	6/12/19	20/12/19
3.	Implementation of Bubble sort program to given list	41	20/12/19	
4.	To implement Quick sort program to give list.	43	20/12/19	
5.	Implementation of stack using python list	45	3/1/20	09/01/2020
6.	Implementing a square using Python list	47	10/1/20	10/01/20
7.	Program on evaluation of given string by using stack in python environment i.e postfix.	49	17/1/20	24/01/2020

☆☆ INDEX ☆☆

No.	Title	Page No.	Date	Staff Member's Signature
8.	Implementation of single linked list by adding the nodes from last position.	51		
9.	Program based on Binary search tree by implementing Inorder, Preorder, Postorder	61		
10.	To Demonstrate the use of circular queue	65		
11.	To sort the list using merge sort	67		

Practical No. 1

- **Aim** : Implement linear search to find an item in the list.

- **Theory** :

Linear Search

Linear search is one of the simplest 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 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 item 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 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 the list

Step4: Print the list

Step5: Accept an element from the user that to be searched into the list.

Step6: Use for loop in a range from 0 to the total no. of elements to search the elements from the list .

Step7 Use of 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.

Step9 : Use the else part of if loop to print that the element is not found 'if' the element which is accepted from user is not there in the list

unsorted program

```
x = [4,3,5,8,9,2]
a = int(input("enter a number : "))
for i in range(len(x)):
    if (a == x[i]) :
        print("element found at :",i)
        break
if (a != x[i]):
    print("No element found
```

Output

```
enter a number : 3
element found At : 1

enter a Number : 9
element found at : 4

enter a number : 10
No element found.
```

```
# Sorted program

S = list(input("Enter the Numbers : "))
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 position", i)
        break
if (a != S[i]):
    print("Number Not found !")
```

Output

```
Enter the Number : 2, 5, 8, 9, 1
[1, 2, 5, 8, 9]
Enter the Numbers to be searched : 5
Number found in position 2

Enter the Number : 2, 3, 9, 5, 1
[1, 2, 3, 5, 9]
enter the Numbers to be searched : 7
Number Not found.
```

Step 10 : Draw the output of given algorithm.

2] Sorted linear search :

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

- Algorithm.

Step 1 : Create Empty list and assign it to variable.

Step 2 : Accept total no. of elements to be inserted into the list from user, say 'n'

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

Step 4 : Use sort() method to sort the accepted element and assign in increasing order the list then print the li

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

Step 6 : The use else statement if element is not found in range then satisfy the given condition.

Step7 : Use for loop in range from 0 to the total no. of elements to be searched before using this accept or search no. from user using input statement

Step8 : Use for loop that the elements in the list equal to the element accepted from user

Step9 : If the element is found then print the statement that the element is found along with the element position.

Step10 : Use another if loop to print that the element is not found if the element which is accepted from user is not there in the list.

Step11 : Attach the input and output of above algorithm.

MM
29/11/17

Source code

```
a = list(input("Element list : "))
a = sort()
c = len(a)
s = int(input("Enter search no. : "))
if (s > a[c-1] or s < a[0]):
        print("not in list")
else :
    first, last = 0, c-1
    for i in range(a, 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
```


Practical - 2 No. 2

Aim : Implement Binary Search to find an 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, search or binary chop is a search algorithm that finds a 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 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() to sort the accepted elements and assign it in increasing order and print the list after sorting.

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

Step 6 :- Then use the else statement, if statement is found in range then satisfy the above condition

Step 7 : Accept an argument & key of the element that element has to be sorted.

Step 8 :- Initialize first to 0 and last to last element of list a array is starting from 0 if it is initialized less than the total count

Step 9 :- use for loop & assign to the given loop.

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

Step 11 : else if the item to be searched is less than middle term then Initialize last (m) = mid (m) - 1

else
Initialize first (l) = mid (m) - 1

Step12 : Repeat till you found the element stick the input & output of above algorithm.

Bubble sort

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

Output :

Enter Numbers : 2,4,8,1,9,5,6
['1','2','4','5','6','8','9']

Enter Numbers : 0.43,0.32,0.78,0.19
['0.19', '0.32', '0.43', '0.78']

Enter Numbers : 232,543,123,901
['123', '232', '543', '901']

Practical NO. 3

Bubble Sort

- **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 then swapping their position if the pair is in the wrong order, this is the simplest form of sorting available. In this, we sort the given elements in ascending and descending order by comparing two adjacent elements at a time.

Algorithm

Step 1 : Bubble sort stands for by comparing the first two elements of an array and swapping it if necessary.

Step 2 : If we want to sort the elements of array in ascending order then first element is greater than second then if we need to swap element

Step 3 : If the element is smaller than second then we do not swap the elements.

Step 4 : Again second and third elements are compared and swapped if it is necessary and this process goon until last and second last element is compared and swapped

```
print("quick sort")
def quicksort(alist):
    helpca
def helpca(alist, first, last):
    if first<last:
        split = part(alist, first, last)
        helpca(alist, first, split-1)
        helpca(alist, split+1, last)
def part(alist, first, last):
    pivot = alist[first]
    l = first+1
    r = last
    done = False
    while not done:
        while l<=r and alist[l]<=pivot:
            l=l+1
        while alist[r]>=pivot and r>=l:
            r=r-1
        if r<l:
            done = True
        else:
            t = alist[l]
            alist[l]=alist[r]
            alist[r]=t
    t=alist[first]
    alist[first]=alist[r]
    alist[r]=t
n=int(input("enter range for list:"))
alist=[]
for b in range(0,n):
    b=int(input("enter element:"))
    alist.append(b)
n=len(alist)
quick(alist)
print(alist)
```

Practical No. 21 43

Quick sort

- **Aim** : Implement quick sort to sort the given list
- **Theory** : The quick sort is a recursive algorithm based on the divide and conquer technique
- **Algorithm:**

Step 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.

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 than or greater than pivot value.

Step 3 : Partitioning begins by locating two position markers let's call them left mark & right mark at the beginning and end of remaining items in the list. The goal of partition process is to move items that are on wrong side with respect to pivot value, also converging on the split point.

Step 4 : We begin incrementing leftmark until we locate a value that is greater than the PV we then decrement rightmark until we find a value that is less than the pivot value. At this point we have two items that are out of place with respect to eventual split point.

Step 5 : At the point where rightmark becomes less than leftmark we stop the position of rightmark is now split point

Step 6 : The pivot value can be exchanged with the contents of split point and PV (pivot value) is now in place

Step 7 : In addition, all the items to left of split position are less than PV all the items to the right of split point are greater than PV. The list can now be divided at split point & quick sort can be invoked recursively on 2 halves

Step 8 : The quick sort function invokes a recursive function, quicksorthelper

Step 9 : quicksorthelper begins with some base case as merge sort

Step 10 : If length of list is less than or equals one, it is already sorted

Step 11 : If it is greater, then it can be partitioned and sorted

Step 12 : The partition function, implement the process described earlier

Step 13 : Display and stick the coding and output of above algorithm.

Output :

Enter range for list : 5
Enter elements : 4
Enter Elements : 3
Enter elements : 2
Enter elements : 1
Enter elements : 9
[1, 2, 3, 4, 9]

20/12/19

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

04/01/2022

- 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 LIFO (Last In First Out) principle as the elements that was inserted last will be removed first. A stack can be implemented as the array as well as linked list. Stack has three basic operation: push, pop, peek. The operations of adding and removing the elements is known as push & pop peek operations topmost element of stack

Algorithm:

Step1 : Create a class stack with instance variable items.

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

Step3 : Define methods push and pop under the class stack.

Output:

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

S.peek()

04/01/2020

Program

```
class Queue:
    global r
    global f
    global a
    def __init__(self):
        self.f = 0
        self.r = 0
        self.a = [0,0,0,0,0]
    def enQueue(self,value):
        self.n = len(self.a)
        if (self.r == self.n):
            print("Queue is full")
        else:
            self.a[self.r] = value
            self.r += 1
            print("Queue element inserted",value)
    def deQueue(self):
        if (self.f == len(self.a)):
            print("Queue is empty")
        else:
            value = self.a[self.f]
            self.a[self.f] = 0
            print("Queue element Deleted",value)
            self.f += 1
b = Queue()
```


PRACTICAL no. 8

- **Aim:** Implementing a Queue using Python list
- **Theory:** Queue is a linear data structure which has 2 reference front and rear.

Implementing a Queue using Python list is the simplest as the python list provides inbuilt functions to perform the specified operation of the queue. It is based on the principal that a new element is inserted after rear and element of queue is deleted which is at front. In simple terms, a queue can be described as a data structure based on first in first out FIFO principle.

- **Queue():** Creates a new empty Queue
- **EnQueue():** Insert an element at the rear of the queue and similar to that of insertion at 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.

Pg.

- Algorithm :

Step 1 : Define a class queue and assign global variables then define init() method with self argument in init() assign or initialize the initial value with the help of self argument.

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

Step 3 : use if statement that length is equal to max then queue is full or else insert the element in empty list or display that queue element added successfully and increment by 1.

Step 4 : Define deQueue() with self-argument. Under this, use if statement that front is equal to length of list then display Queue is empty or else, give that front is at zero and using that delete the increment from front side and increment it by 1.

Step 5 : Now call the queue() function and give the elements that has to be added in the empty list using enQueue() and print the list after adding and same for deleting and display the list after deleting the element from the list

Output :

```
>>> b.enQueue(4)
Queue element inserted 4
>>> b.enQueue(5)
Queue element inserted 5
>>> b.enQueue(6)
Queue element inserted 6
>>> b.enQueue(7)
Queue element inserted 7
>>> b.enQueue(8)
Queue element inserted 8
>>> b.enQueue(9)
Queue is full
>>> b.a
[4,5,6,7,8]
>>> b.deQueue()
Queue element deleted 4
>>> b.deQueue()
Queue element deleted 5
>>> b.deQueue()
Queue element deleted 6
>>> b.deQueue()
Queue element deleted 7
>>> b.deQueue()
Queue element deleted 8
>>> b.deQueue()
Queue is empty
>>> b.a
[0,0,0,0,0]
```

10/01/24

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 = "89 9 *+"
r = evaluate(s)
print("the value is :",r)
```


Aim : Program of Evaluation of given string by using stack in python environment i.e postfix

theory :

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

Algorithm :

Step 1 : Define evaluate as function, then create an empty stack in Python.

Step 2 : Convert the string to a list by using the string method 'split'

Step 3 : Calculate the length of string & print it

Step 4 : use for loop to assign the range of string then give condition using if statement

Step 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 's'.

Step 6 : If the token is an operator *, /, +, -, it will need two operands. Pop the 'S' twice. The first pop is second operand and the second pop is the first operand.

Step 7 : Perform the arithmetic operation. Push the result back on the 'S'.

Step 8 : When the input expression has been completed processed, the result is on the stack. Pop the 'P' & return the value.

Step 9 : Print the result of string after the evaluation of postfix

Step 10 : Attach output & input of above algorithm.

Output:

>>> evaluated value is : 4062

24/01/2020

Source code

```
# linked list
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 add2(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)
```


Practical No. 9

- Aim : Implementation of single Linked list by adding the nodes from last position
- Theory : A linked list is a linear data structure which stores the elements in a node in a linear fashion but no necessarily contiguous. The individual element of the linked list called a Node. Node comprises of 2 parts ① Data ② Next. Data stores all the information w.r.t the element for example roll Number, name, address, etc whereas next stores to the next node. In case of larges list if we add/remove any element in the list all the elements of list has to adjust itself every time we add it very tedious task so linked list is used to solving this type of problem.
- Algorithm :

Step 1 : Traversing of a linked list mean visiting all the nodes in the linked list in order to perform some operation on them.

Step 2 : The entire list can be accessed using the first node of linked list, the first node of the linked list in turn is referred by the head pointer of linked list.

Step 3 : Thus the entire linked list can be traversed using the node which is referred by the head pointer of linked list.

Step 4 : Now that we know that we can traverse the entire linked list using the head pointer we should only use it to refer the first Node of list only.

Step 5 : We should not use the head pointer to traverse the entire linked list because the head pointer is our only reference to the first node in the linked list, modifying the reference of the head pointer can lead to changes we cannot retrieve.

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

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

Output :

```
>>> q.addB(10)
>>> q.addB(20)
>>> q.addB(30)
>>> q.addB(40)
>>> q.addL(50)
>>> q.addL(60)
>>> q.addL(70)
>>> q.addL(80)
>>> q.display()
40
30
20
10
50
60
70
80
```

24/01/2024

Step 8: Now that current is retaining to the first node it we want to access 2nd node of list we can retap it as the next node of 1st Node.

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

Step 10: Similarly we can transverse rest nodes in the linked list using same method by while loop.

Step 11: Our concern now is to find the terminating condition 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 have any next node, the value of the next field of last node is None.

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

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

24/01/20

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

#Binary Tree

```
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 h.left == None:
                    h.left = p
                    print(p.val,"Node is added to leftside successfully",h.val)
                    break
                else:
                    h = h.left
        else:
            if h.right == None:
                h.right = p
                print(p.val,"Node is added to Rightside successfully",h.val)
                break
            else:
                h = h.right
```


Aim : Program based on Binary Search tree by implementing Inorder, Preorder, Postorder.

Theory : Binary tree is a tree which supports maximum of 2 children for any node within the tree. Thus any particular node can have either 0 or 1 or 2 children there is another identity of binary tree that is ordered such that one child is identified as left child and another as right child.

Inorder : (1) Transverse the left subtree. The left subtree inturn might have left and right subtrees.
2) Visit root node.
3) Transverse the right subtree and repeat it.

Preorder : 1) Visit the root node.
2) Transverse the left subtree. The left subtree in turn might have left and right subtrees.
3) Transverse the right subtree and repeat it.

Postorder : 1) Transverse the left subtree. The left subtree inturn might have left and right subtrees.
2) Transverse the right subtrees.
3) Visit the root Node.

Algorithm

Step 1 : Define class Node and define __init__() with 2 argument. Initialize the value in this method

Step 2 : Again define a new class BST that is Binary search tree with init() with self argument and assign root is None.

Step 3 : Define add() method for adding the Node. Define a variable p that p=node(value).

Step 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 operating that is leftside.

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

Step 6 : Use if statement within that else statement for checking that Node is greater than main root then find the right side

Step 7 : After this left subtree and right subtree repeat this method to arrange the node according to Binary search tree.

```
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)
```

Output :

```
>>> t = BST()
>>> t.add(7)
('Root is added successfully', 7)
>>> t.add(5)
('Root is added successfully', 5)
>>> t.add(12)
('Root is added successfully', 12)
>>> t.add(3)
('Root is added successfully', 3)
```

```
>>> t.add(6)
('Root is added successfully', 6)
>>> t.add(9)
('Root is added successfully', 9)
>>> t.add(15)
('Root is added successfully', 15)
>>> t.add(1)
('Root is added successfully', 1)
>>> t.add(4)
('Root is added successfully', 4)
>>> t.add(8)
('Root is added successfully', 8)
>>> t.add(10)
('Root is added successfully', 10)
>>> t.add(13)
('Root is added successfully', 13)
>>> t.add(17)
('Root is added successfully', 17)
```


Step 8: Define Inorder(), Pre-order(), Postorder() with root argument and use if statement that root is none and return that in all.

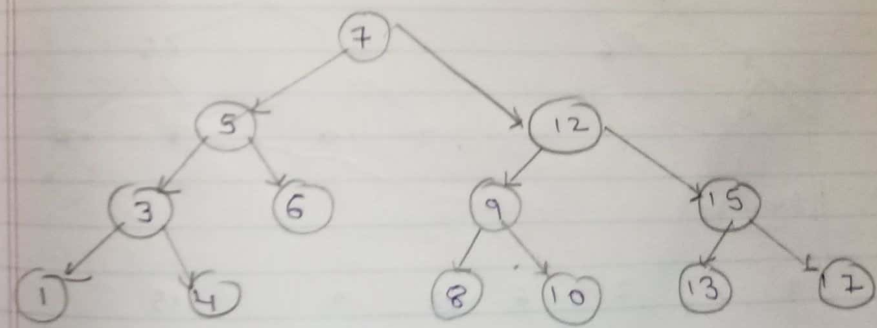
Step 9: In Inorder else statement used for giving that condition first left, root and right node.

Step 10: For Preorder we have to give condition in else that first root, left and the right node

Step 11: For Postorder In else part assign left then right and then go for root node.

Step 12: Display the output & input of above algorithm.

Binary Search Tree.



Inorder (LVR)

Step1:

Step2:

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

Preorder : VLR

Step1

Step 2

7 5 2 6 12 9 15
1 4 8 10 13 17

>>7 Inorder (root)

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

>>> Preorder (root)

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

#a

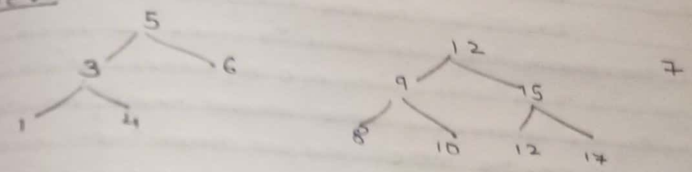
>>> Postorder (t.root)

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

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

Postorder : (LRV)

Step 1 :



Step 2 : 3 6 5 9 15 12 7
1 4 8 10 13 17

Step 3

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

PRACTICAL NO:10

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 elements can be inserted. When we dequeue any element we are 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 the queue behave like a circular data structure. In case of a circular queue head pointer will always point to the front of the queue and tail pointer will always point to the end of queue. Initially, the head and tail pointer will be pointing to the same location, this would mean that queue is empty. New data is always added to the location pointed by the tail pointer & once the data is added, tail pointer is incremented by one to point to the next available location. Before we have some common real world examples where circular queues are used.


```
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:
            print("Queue is empty")
            self.f = s
q = Queue()
```

```
else:
    s = self.f
    self.f = 0
    for i in range(self.f, self.r):
        print(self.a[i], end=" ")
        self.f = self.f + 1
else:
    print("Queue is empty")
    self.f = 5

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
[0, 200, 300, 0, 0]
```


1) Computer controlled traffic signal system uses circular queue
2) CPU scheduling and memory management.

PRACTICAL NO: 11

Aim: To sort the list using merge sort

Theory: Like Quicksort, mergesort is a Divide and conquer algorithm. It divides input array in the halves and then merges the two 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 merges the two sub-array into one. The array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge process comes into action and starts merging back till the complete array is merged.

Applications:

Mergesort is is useful for sorting linked list in O(nlogn) time. Merge sort accesses data sequentially and the need of random access is low.

Inversion count problem

used in external sorting

Mergesort is more efficient than quicksort some types of list if the data to be sorted can only be efficiently accessed potentially.

Code:

```
def mergesort(arr):
    if len(arr)>1:
        mid = len(arr)//2
        lefthalf = arr[:mid]
        righthalf = arr[mid:]
        mergesort(lefthalf)
        mergesort(righthalf)
        i=j=k=0
        while(i< len(lefthalf) and j < (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 (j<len(lefthalf)):
            arr[k] = righthalf[j]
            j=j+1
            k=k+1
arr = [27, 89, 70, 55, 62, 99, 45, 14, 10]
print("Random list : ", arr)
mergeSort(arr)
print("\n merge sorted list : ", arr)
```

Output:

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

merge sorted list : [10, 14, 27, 45, 62, 70, 89, 90]