WEEK 12

1. **Write a program to implement circular queue with following operations**
   a. **Insertion**
   b. **Deletion**
   c. **Display**

AIM: A program to implement circular queue with the given operations

DESCRIPTION:

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.

Operations on Circular Queue:

- enQueue(value): This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position. Check whether queue is full - check (rear==size-1 && front == 0) || (rear == front-1)). If it is full then display queue is full. If queue is not full then, check if (rear==size-1 && front!=0) if it true then set rear =0 and insert element
- deQueue(): This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. Check whether queue is empty or not. Then if it is not empty, check if front is equal to size-1 and then if it true, set front = 0 and return the element.

PROGRAM:

```
class circular:
  def __init__(self,size):
    self.size=size
    self.cq=[None]*size
    self.front=-1
    self.rear=-1
  def insert(self):
    data=int(input("Enter value to be inserted: "))
    if (self.front==(self.rear+1)%self.size):
      print("Overflow")
    elif (self.front==-1 and self.rear==-1):
      self.front=0
      self.rear=0
```

```
else:
            self.rear=(self.rear+1)%self.size
        self.cq[self.rear]=data
    def delete(self):
        if self.rear==-1:
            print("Underflow")
        else:
            if (self.front==self.rear):
                self.front=-1
                self.rear=-1
            elif (self.front==self.size-1):
                self.front=0
            else:
                self.front=(self.front+1)%self.size
    def display(self):
        if self.rear==-1:
            print("Queue is empty")
        if self.front>=self.rear:
            for x in range(self.front,self.size):
                print(self.cq[x],end=' ')
            for x in range(0,self.rear+1):
                print(self.cq[x],end=' ')
        else:
            for x in range(self.front,self.rear+1):
                print(self.cq[x],end=" ")
            print("\n")
n=int(input("Enter size of the Queue: "))
c=circular(n)
```

```
while True:
    co=int(input("\n1. Enqueue\t2. Dequeue\t3. Display\t4. Exit\nEnter your choice: "))
    if co==1:
        c.insert()
    elif co==2:
        c.delete()
    elif co==3:
        c.display()
    elif co==4:
        break
    else:
        print("Wrong choice, try again")
```

OUTPUT:

```
Enter size of the Queue: 5

1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 1
Enter value to be inserted: 1

1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 1
Enter value to be inserted: 2

1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 1
Enter value to be inserted: 3

1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 3
1 2 3


1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 1
Enter value to be inserted: 4

1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 3
1 2 3 4


1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 2

1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 3
2 3 4
```

```
1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 1
Enter value to be inserted: 5

1. Enqueue      2. Dequeue      3. Display      4. Exit
Enter your choice: 4
>>>
```

CONCLUSION: The code is error free and runs as expected.

TIME COMPLEXITY: Time complexity of enqueue(), dequeue() operation is O(1).

2. Write an application to implement hashing, when collision occurred use linear probing method to resolve collision and after storing elements find location of specified element in hashing.

AIM: An application to implement hashing using linear probing method to resolve collision and after storing elements, find location of specified element in hashing.

DESCRIPTION:

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.
   hash = hashfunc(key)
   index = hash % array_size
   In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).
   Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is index. The probing sequence for linear probing will be:
   index = index % hashTableSize
   index = (index + 1) % hashTableSize
   index = (index + 2) % hashTableSize
   index = (index + 3) % hashTableSize

PROGRAM:

```
class Hashtable:
  def __init__(self,size) -> None:
    self.a=[None]*size
    self.size=size
  def add_element(self,ele):
    if self.a[ele%self.size]==None:
      self.a[ele%self.size]=ele
    elif self.a[ele%self.size]!=None:
      i=1
      while(True):
        if self.a[(ele+i)%self.size]==None:
          self.a[(ele+i)%self.size]=ele
          break
```

```
            i+=1
    def search_index(self,ele):
        if self.a[ele%self.size]!=ele:
            i=0
            while(True):


                if self.a[(ele+i)%self.size]==None:
                    self.a[(ele+i)%self.size]=ele
                    return ((ele+i)%self.size)
                i+=1
        else:
            return ele%self.size
size=int(input("Enter size of the hash table: "))
H=Hashtable(size)
for x in range(size):
    a=int(input("ENter ele: "))
    H.add_element(a)
    print(*H.a)
ele=int(input("Find :"))
print(H.search_index(ele))
ele=int(input("Find :"))
print(H.search_index(ele))
```

OUTPUT:

```
========== RESTART: C:\Users\S.BALACHANDRASHEKAR\Downloads\hashing.py ==
Enter size of the hash table: 5
ENter ele: 3
None None None 3 None
ENter ele: 4
None None None 3 4
ENter ele: 5
5 None None 3 4
ENter ele: 1
5 1 None 3 4
ENter ele: 2
5 1 2 3 4
Find :5
0
Find :1
1
>>>
```

CONCLUSION: The code is error free and runs as expected.

TIME COMPLEXITY:

Worst-case: O(n)

Average and best cases: O(1).