① Circular Array Queue

```
class FullException(Exception):
    pass
class EmptyException(Exception):
    pass
class ArrayQueue:
    def __init__(self, ele):
        self.data = [data] * capacity
        self.capacity = capacity
        self.front = 0
        self.size = 0
    def enqueue(self, ele):
        if self.size == self.capacity:
            raise FullException("Queue Overflow!")
        rear = (self.front + self.size) % self.capacity
        self.data[rear] = ele
        self.size += 1
    def dequeue(self):
        if self.size == 0:
            raise EmptyException("Queue Underflow!")
        deletedValue = self.data[self.front]
        self.data[self.front] = None
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return deletedValue
    def first(self):
        if self.size == 0:
            raise EmptyException("Queue Underflow!")
        return self.data[self.front]
    def IsEmpty(self):
        return self.size == 0
    def __len__(self):
        return self.size
```

② Linked_List Queue

```
class LinkedListQueue:
    class Node:
        def __init__(self, data = None, next = None):
            self.data = data
            self.next = next
    def __init__(self):
        self.head = None
        self.size = 0

    def enqueue(self, ele):
        newNode = self.Node(ele, None)
        if newNode == None:
            print("Could not create newNode")
            return
        if self.size == 0:
            self.head = newNode
            self.size += 1
            return
        currentNode = self.head
        tailNode = self.head
        while(currentNode):
            if currentNode.next == None:
                tailNode = currentNode
                break
            else:
                currentNode = currentNode.next
        tailNode.next = newNode
        self.size += 1

    def dequeue(self):
        if self.head == None:
            print("Queue underflow!")
            return
        deletedValue = self.head.data
        self.head = self.head.next
        self.size -= 1
        return deletedValue

    def TraverseQueue(self):
        currentNode = self.head
        while(currentNode):
            print(currentNode.data, end = " ")
            currentNode = currentNode.next
```

③ [Array] Circular Array Deque

```
class CirArrayDeque:
    def __init__(self, capacity=10):
        self.data = [None]*capacity
        self.capacity = capacity
        self.size = 0
        self.front = 0

    def addFirst(self, ele):
        if self.size == self.capacity:
            print("Dequeue Overflow!")
            return
        self.data[self.front-1] = ele
        self.front = (self.front-1)%self.capacity
        self.size += 1

    def addLast(self, ele):
        if self.size == self.capacity:
            print("Dequeue Overflow!")
            return
        rear = (self.front + self.size)%self.capacity
        self.data[rear] = ele
        self.size += 1

    def deleteFirst(self):
        if self.size==0:
            print("Dequeue underflow!")
            return
        deletedValue = self.data[self.front]
        self.front = (self.front+1)%self.capacity
        self.size -= 1
        return deletedValue

    def deleteLast(self):
        if self.size==0:
            print("Dequeue underflow!")
            return
        rear = (self.front+self.size)%self.capacity
        deletedValue = self.data[rear-1]
        self.data[rear-1] = None
        self.size = self.size-1
        return deletedvalue

    def IsEmpty(self):
        return self.size==0
```

```
def first(self):
    if self.size == 0:
        print("Dequeue underflow!")
        return
    return self.data[self.front]

def last(self):
    if self.size == 0:
        print("Dequeue underflow!")
        return
    rear = (self.front + self.size) % self.capacity
    return self.data[rear - 1]

def __len__(self):
    return self.size
```

④ Circular LinkList Queue

```
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

class CQ:
    def __init__(self):
        self.tail = None
        self.size = 0

    def enqueue(self, ele):
        newNode = Node(ele)
        if newNode == None:
            print("Error")
            return
        if self.size == 0:
            newNode.next = newNode
        else:
            newNode.next = self.tail.next
            self.tail.next = newNode
        self.tail = newNode
        self.size += 1

    def dequeue(self):
        if self.size == 0:
            print("Empty")
            return
        head = self.tail.next
        deletedValue = head.data
        if self.size == 1:
            self.tail = None
        else:
            self.tail.next = head.next
        self.size -= 1
        return deletedValue

    def Traverse(self):
        currentNode = self.tail.next
        for i in range(self.size):
            print(currentNode.data, end=" ")
            currentNode = currentNode.next
```

```
def first(self):
    if self.size==0:
        print("Error")
    head= self.tail.next
    return head.data

def is_Empty(self):
    return self.size==0

def __len__(self):
    return self.size

def top(self):
    return self.tail.data

def traverse(self):
    currentNode =self.tail.next
    for i in range(self.size):
        print(currentNode.data ,end=" ")
        currentNode = currentNode.next
```

⑤ Doubly Linked List

```
class Node:
    def __init__(self, data=None, prev=None, next=None):
        self.data = data
        self.prev = prev
        self.next = next

class DLL:
    def __init__(self):
        self.header = Node()
        self.trailer = Node()
        self.header.next = self.trailer
        self.trailer.prev = self.header
        self.size = 0

    def Insert_between(self, ele, predecessor, successor):
        newNode = Node(ele, predecessor, successor)
        if newNode == None:
            print("could not create new Node")
            return
        predecessor.next = newNode
        successor.prev = newNode
        self.size += 1
        return newNode

    def ForwardTraversal(self):
        currentNode = self.header.next
        for i in range(self.size):
            print(currentNode.data)
            currentNode = currentNode.next

    def BackwardTraversal(self):
        currentNode = self.trailer.prev
        for i in range(self.size):
            print(currentNode.data)
            currentNode = currentNode.prev

    def InsertAtHead(self, ele):
        return self.Insert_between(ele, self.header, self.header.next)

    def InsertAtTail(self, ele):
        return self.Insert_between(ele, self.trailer.prev, self.trailer)
```

```
def DeleteGivenNode(self, node):
    deletedValue = node.data
    predecessor = node.prev
    successor = node.next
    predecessor.next = successor
    successor.next = predecessor
    predecessor = None
    successor = None
    self.size -= 1
    return deletedValue
```

⑥ LinkedList Deque

```
class LinkedListDeque(DLL):
def addfirst(self,ele):
  self.Insert_between(ele,self.header,self.header.next)

def addlast(self,ele):
  self.Insert_between(ele,self.trailer.prev,self.trailer)

def deleteFirst(self):
  if self.size==0:
     print("Deque underflow!")
     return
  return self.DeleteGivenNode(self.header.next)

def deletelast(self):
  if self.size==0:
     print("Deque underflow!")
     return
  return self.DeleteGivenNode(self.trailer.prev)

def first(self):
   if self.size==0:
      print("Deque underflow!")
      return
   return self.header.next.data

def last(self):
   if self.size==0:
      print("Deque underflow")
      return
   return self.trailer.prev.next

def isEmpty(self):
    return self.size==0

def __len__(self):
    return self.size
```