DEQUE DATA STRUCTURE

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First In First Out).

Representation of Deque

Types of Deque

- **Input Restricted Deque**
  In this deque, input is restricted at a single end but allows deletion at both the ends.
- **Output Restricted Deque**
  In this deque, output is restricted at a single end but allows insertion at both the ends.

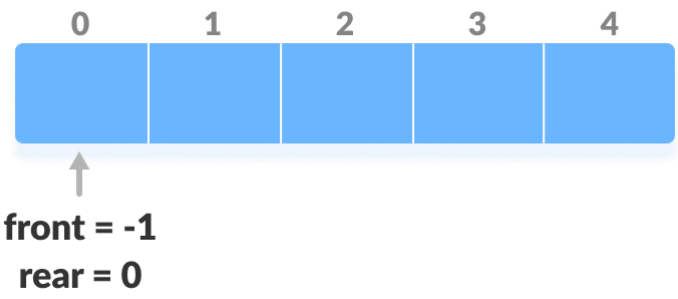
Operations on a Deque

Below is the circular array implementation of deque. In a circular array, if the array is full, we start from the beginning.

But in a linear array implementation, if the array is full, no more elements can be inserted. In each of the operations below, if the array is full, "overflow message" is thrown.

Before performing the following operations, these steps are followed.

1. Take an array (deque) of size n.
2. Set two pointers at the first position and set front = -1 and rear = 0.

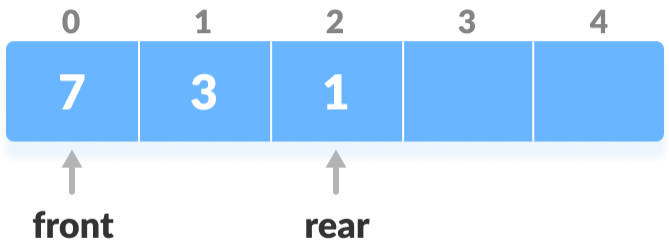


Initialize an array and pointers for deque

1. Insert at the Front

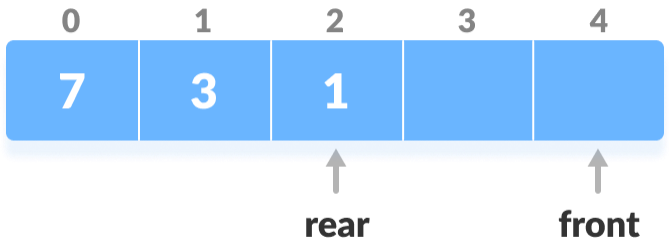
This operation adds an element at the front.

1. Check the position of front.



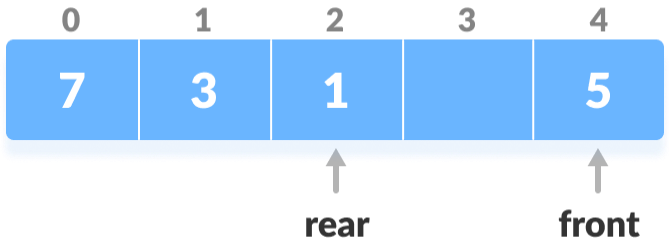
Check the position of front

2. If front < 1, reinitialize front = n-1 (last index).



Shift front to the end

3. Else, decrease front by 1.
4. Add the new key 5 into array[front].

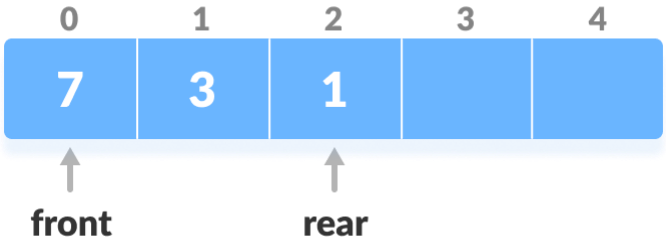


Insert the element at Front

2. Insert at the Rear

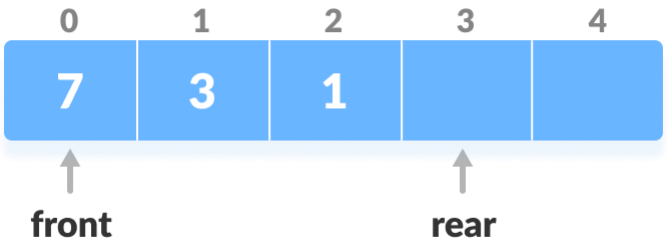
This operation adds an element to the rear.

1. Check if the array is full.



Check if deque is full

2. If the deque is full, reinitialize rear = 0.
3. Else, increase rear by 1. Add the new key 5 into array[rear].



Increase the rear

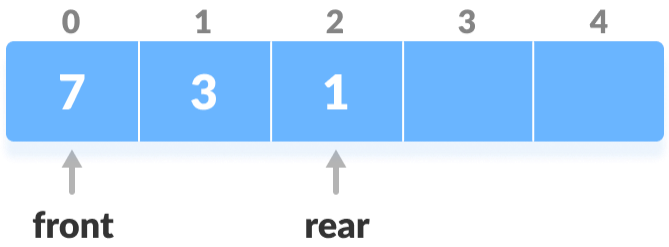
4. Add the new key 5 into array[rear]

Insert the element at rear

3. Delete from the Front

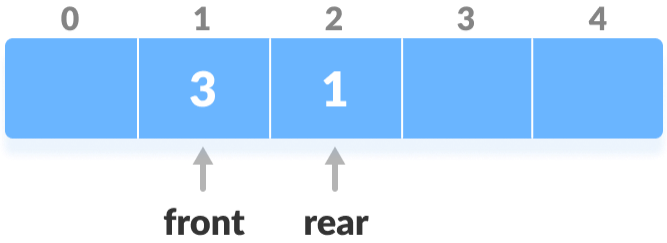
The operation deletes an element from the front.

1. Check if the deque is empty.



Check if deque is empty

2. If the deque is empty (i.e. front = -1), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1.
4. Else if front is at the end (i.e. front = n - 1), set go to the front front = 0.



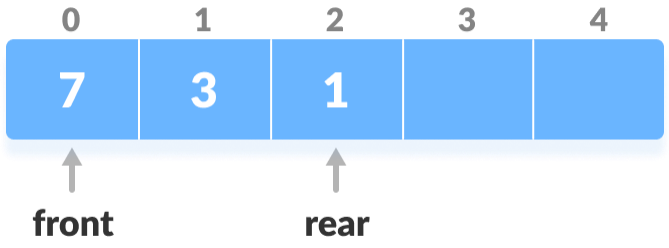
Increase the front

5. Else, front = front + 1.

4. Delete from the Rear

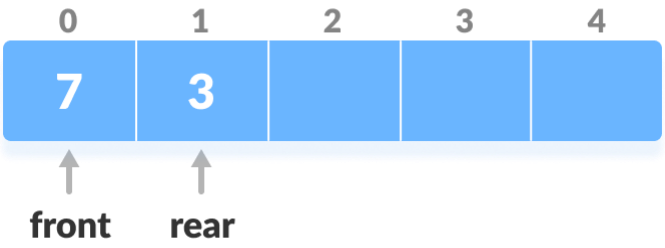
This operation deletes an element from the rear.

1. Check if the deque is empty.



Check if deque is empty

2. If the deque is empty (i.e. front = -1), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1, else follow the steps below.
4. If rear is at the front (i.e. rear = 0), set go to the front rear = n - 1.
5. Else, rear = rear - 1.



Decrease the rear

5. Check Empty

This operation checks if the deque is empty. If front = -1, the deque is empty.

6. Check Full

This operation checks if the deque is full. If front = 0 and rear = n - 1 OR front = rear + 1, the deque is full.

Deque Implementation in Python

```
class Deque:
  def __init__(self):
    self.items = []
  def isEmpty(self):
    return self.items == []
  def addRear(self, item):
    self.items.append(item)
  def addFront(self, item):
    self.items.insert(0, item)
  def removeFront(self):
    return self.items.pop(0)
```

```
    def removeRear(self):
        return self.items.pop()
    def size(self):
        return len(self.items)
d = Deque()
print(d.isEmpty())
d.addRear(8)
d.addRear(5)
d.addFront(7)
d.addFront(10)
print(d.size())
print(d.isEmpty())
d.addRear(11)
print(d.removeRear())
print(d.removeFront())
d.addFront(55)
d.addRear(45)
print(d.items)
```

Time Complexity

The time complexity of all the above operations is constant i.e. O(1).

Applications of Deque Data Structure

1. In undo operations on software.
2. To store history in browsers.
3. For implementing both stacks and queues.