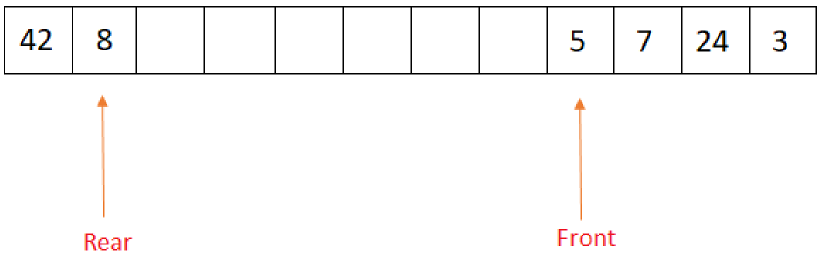
Introduction to Deques

A deque, also known as the **double-ended queue** is an ordered list in which elements can be inserted or deleted at either end. It is also known as a **head-tail linked list** because elements can be added to or removed from either the front (head) or the back (tail).



Properties of Deques

- Deque can be used both as **stack** and **queue** as it allows insertion and deletion of elements from both ends.
- Deque does not require LIFO and FIFO orderings enforced by data structures like stacks and queues.
- There are two variants of double-ended queues -
  - **Input restricted deque:** In this deque, insertions can only be done at one of the ends, while deletions can be done from both ends.
  - **Output restricted deque:** In this deque, deletions can only be done at one of the ends, while insertions can be done on both ends.

Operations on Deques

- **enqueue_front(data):** Insert an element at the front end.
- **enqueue_rear(data):** Insert an element at the rear end.
- **dequeue_front():** Delete an element from the front end.
- **dequeue_rear():** Delete an element from the rear end.
- **front():** Return the front element of the dequeue.
- **rear():** Return the rear element of the dequeue.
- **isEmpty():** Returns true if the deque is empty.
- **isFull():** Returns true if the deque is full.

Implementation of Deques

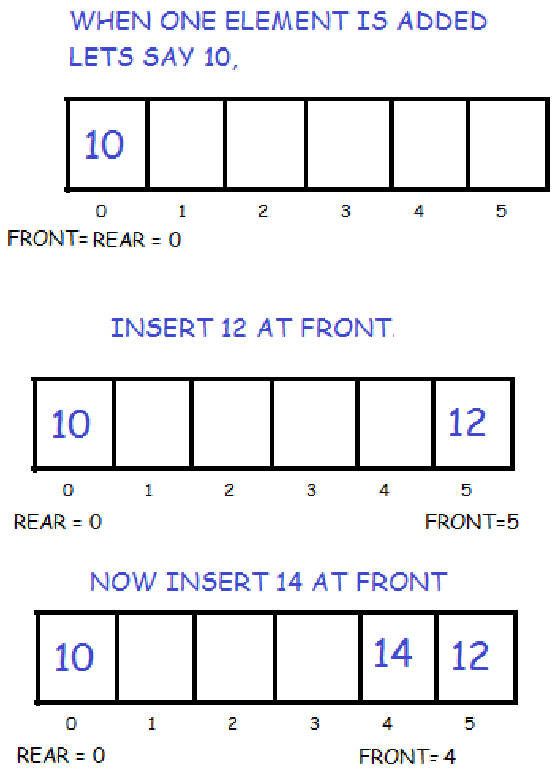
Deques can be implemented using data structures like **circular arrays** or **doubly-linked lists**. Below is the circular array implementation for deques, the same approach can be used to implement deque using doubly-linked lists.

We maintain two variables: **front** and **rear**, front represents the **front end** of the deque and rear represents the **rear end** of the deque.

The circular array is represented as **"carr"**, and size represented by **size**, having elements indexed from **0** to **size - 1**.

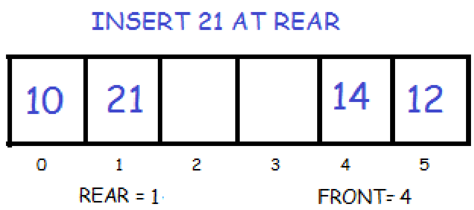
- **Inserting** an element at the **front** end involves **decrementing** the front pointer.
- **Deleting** an element from the **front** end involves **incrementing** the front pointer.
- **Inserting** an element at the **rear** end involves **incrementing** the rear pointer.
- **Deleting** an element from the **rear** end involves **decrementing** the rear pointer.
- **NOTE:** However, front and rear pointers need to be maintained, such that they remain within the bounds of indexing of **0** to **size - 1** of the circular array.
- Initially, the deque is empty, so front and rear pointers are initialized to **-1**, denoting that the deque contains no element.
- **Enqueue_front operation**

  Steps:
  - If the array is full, the data can't be inserted.
  - If there are not any components within the Deque(or array) it means the front is equal to -1, increment front and rear, and set carr[front] as data.
  - Else decrement front and set carr[front] as data.



```
function enqueue_front(data)

        //  Check if deque is full
        if (front equals 0 and rear equals size-1) or (front equals rear+1)
                print ("Overflow")
                return

        /*
                Check if the deque is empty, insertion of an element from the
                front or rear will be equivalent.
                So update both front and rear to 0.
        */
        if front equals -1
                front = 0
                rear = 0
                carr[front] = data
                return

        //  Otherwise check if the front is 0
        if front equals 0
                /*
                        Updating front to size-1, so that front remains within the
                        bounds of the circular array.
                */

                front = size-1
        else
                front = front-1

        carr[front] = data
        return
```

- **Enqueue_rear operation**
  Steps:
    - If the array is already full then it is not possible to insert more elements.
    - If there are not any elements within the Deque i.e. rear is equal to -1, increase front and rear and set carr[rear] as data.
    - Else increment rear and set carr[rear] as data.

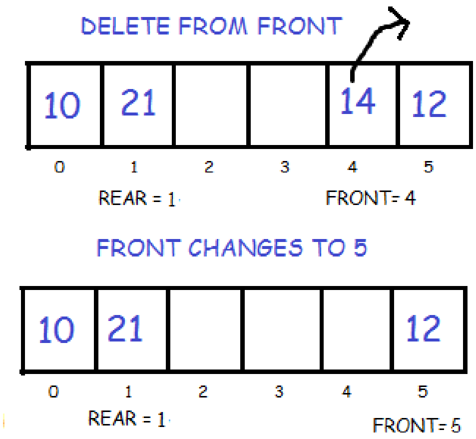



```
function enqueue_rear(data)

        //  Check if deque is full
        if (front equals 0 and rear equals size-1) or (front equals rear+1)
                print ("Overflow")
                return

        /*
                Check if the deque is empty, insertion of an element from the
                front or rear will be equivalent.
                So update both front and rear to 0.
        */

        if front equals -1
                front = 0
                rear = 0
                carr[front] = data
                return

        //  Otherwise check if rear equals size-1

        if rear equals size-1
                /*
                        Updating rear to 0, so that rear remains within the bounds
                        of the circular array.
                */
                rear = 0
        else
                rear = rear+1

        carr[rear] = data
        return
```

- **Dequeue_front operation**

Steps:

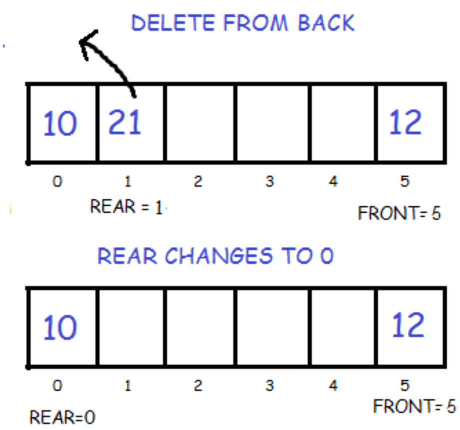
- If the Deque is empty, return.
- If there is only one element in the Deque, that is, front equals rear, set front and rear as -1.
- Else increment front by 1.




```
function dequeue_front()

        //  Check if deque is empty
        if front equals -1
                print ("Underflow")
                return

        /*
                Otherwise, check if the deque has a single element i.e front and
                rear are equal and will be non-negative as the deque is
                non-empty.
        */
        if front equals the rear
                /*
                        Update front and rear back to -1, as the deque becomes
                        empty.
                */
                front = -1
                rear = -1
                return

        //  If the deque contains at least 2 elements.

        if front equals size-1
                //  Bring front back to the start of the circular array.
```

```
            front = 0
    else
            front = front+1
    return
```

- **Dequeue_rear operation**
  Steps:
    - If the Deque is empty, return.
    - If there's just one component within the Deque, that is, rear equals front, set front and rear as -1.
    - Else decrement rear by one.




```
function dequeue_rear()

    //  Check if deque is empty
    if front equals -1
            print ("Underflow")
            return

    /*
            Otherwise, check if the deque has a single element i.e front and
            rear are equal and will be non-negative as the deque is
            non-empty.
    */
    if front equals the rear
            /*
                    Update front and rear back to -1, as the deque becomes
                    empty.
            */
```

```
            front = -1
            rear = -1
            return

    //  If the deque contains at least 2 elements.

    if rear equals 0
            //  Bring rear back to the last index i.e size-1 of the circular array.
            rear = size-1
    else
            rear = rear-1
    return
```

- **Front operation**

  Steps:
  - If the Deque is empty, return.
  - Else return carr[front].

```
function front()

    // Check if deque is empty
    if front equals -1
            print ("Deque is empty")
            return

    //  Otherwise return the element present at the front end
    return carr[front]
```

- **Rear operation**

  Steps:
  - If the Deque is empty, return.
  - Else return carr[rear].

```
function rear()

    //  Check if deque is empty
    if front equals -1
            print ("Deque is empty")
            return
```

```
    //  Otherwise return the element present at the rear end
    return carr[rear]
```

- **Is Empty operation**
  Steps:
  - If front equals -1, the Deque is empty, else it's not.

```
function isEmpty()

    //  Check if front is -1 i.e no elements are present in deque.
    if front equals -1
        return true
    else
        return false
```

- **Is Full operation**
  Steps:
  - If front equals 0 and rear equals size - 1, or front equals rear + 1, then the Deque is full, else it's not. Here **"size"** is the size of the circular array.

```
function isFull()

    /*
        Check if the front is 0 and rear is size-1 or front == rear+1, in both
        cases, we cannot move front and rear to perform any insertions
    */

    if (front equals 0 and rear equals size-1) or (front equals rear+1)
        return true
    else
        return false
```

Time complexity of various operations

Let 'n' be the number of elements in the deque. The time complexities of deque operations in the worst case can be given as:

Operations	Time Complexity
Enqueue_front(data)	O(1)
Enqueue_rear(data)	O(1)
Dequeue_front()	O(1)
Dequeue_rear()	O(1)
Front()	O(1)
Rear()	O(1)
isEmpty()	O(1)
isFull()	O(1)

Applications of Deques

- Since deques can be used as stack and queue, they can be used to perform **undo-redo** operations in software applications.
- Deques are used in the **A-steal job scheduling algorithm** that implements task scheduling for multiple processors (multiprocessor scheduling).
- They are also helpful in finding max/min values of all **subarrays of size k** in the array in O(n) time, where n is the size of the array