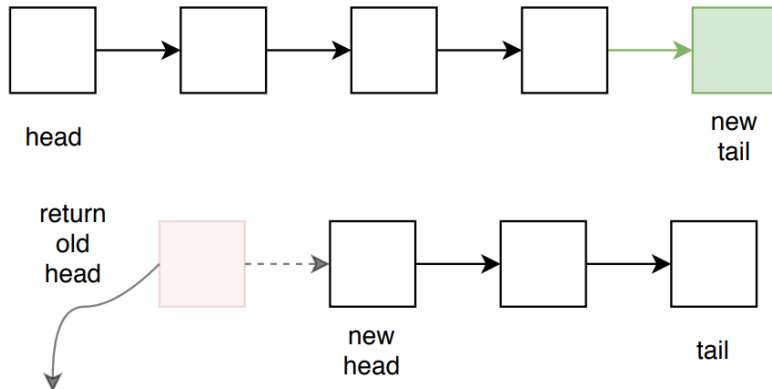


Recitation 3 –Queues

Implementing a Queue with a singly linked list with both head and tail



```
import java.util.LinkedList;

public class MyQueue {

    private LinkedList<Integer> lst;
    //private keyword is important!

    //constructor
    MyQueue() {
        lst = new LinkedList<Integer>();
    }

    public void enqueue(Integer value) {
        lst.addLast(value);
    }

    public Integer dequeue() {
        return lst.removeFirst();
    }

    public void print() {
        System.out.println(lst.toString());
    }

}
```

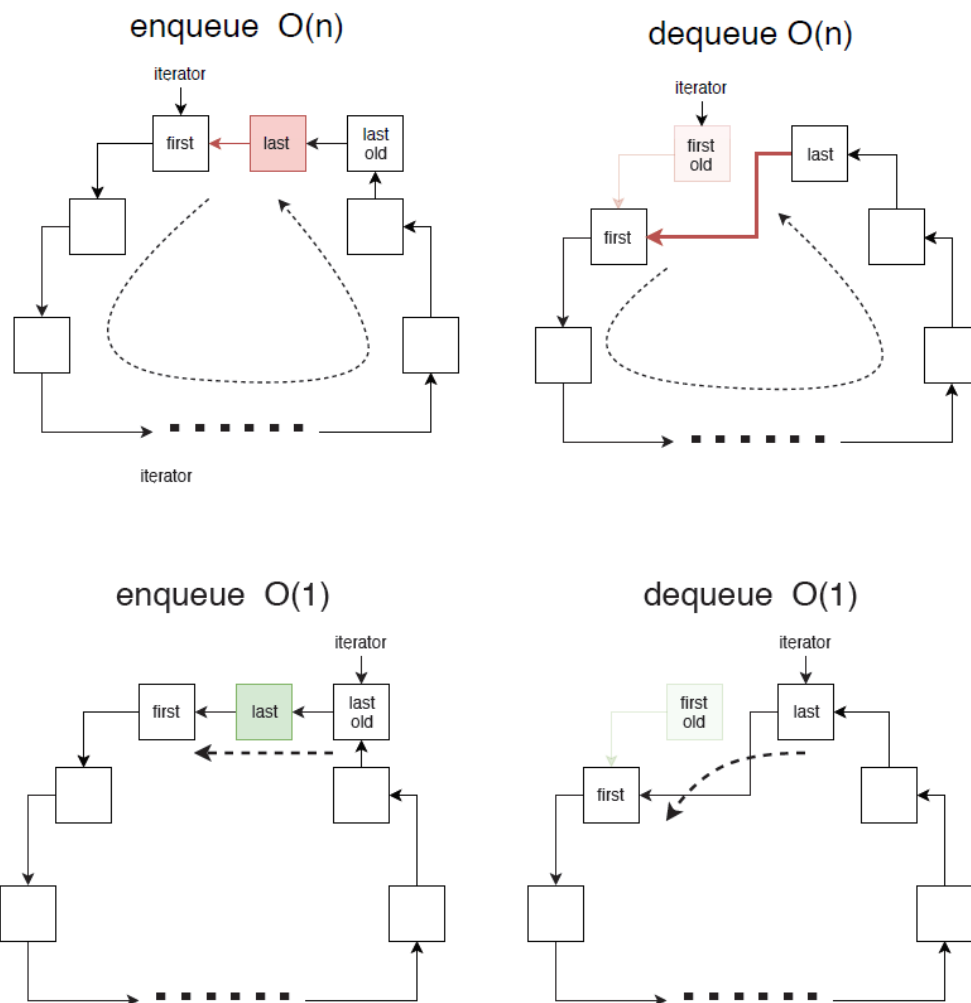
Is it possible to reverse the logic? Use tail to dequeue, head to enqueue? Yes!

What if only head? Only head: enqueue $O(n)$, dequeue $O(1)$ or vice versa by reverse logic

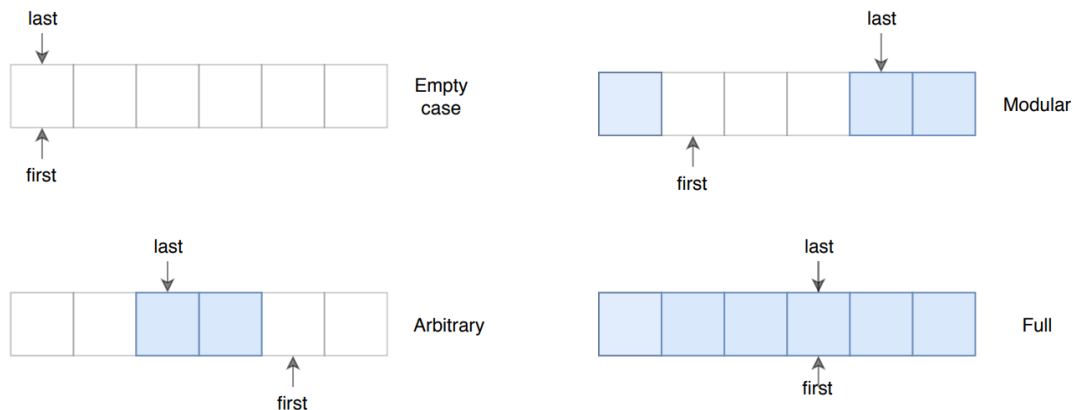
Why is overflow not an issue this time? Dynamic allocation comes with linked lists

3.35 One way to implement a queue is to use a circular linked list. In a **circular linked list**, the last node's next link links to the first node. Assume the list does not contain a header and that we can maintain, at most, one iterator corresponding to a node in the list. For which of the following representations can all basic queue operations be performed in constant worst-case time? Justify your answers.

- Maintain an iterator that corresponds to the first item in the list.
- Maintain an iterator that corresponds to the last item in the list.



Implementing a Queue with an array (if time permits)



```
public class MyQueue {
```

```
    private int[] arr;
    private int first = 0;
    private int last = 0;
    private int size;
    private int count = 0;
```

```
    MyQueue(int n) {
        size = n;
        arr = new int[n];
    }
```

```
    boolean isEmpty() {
        return (count == 0);
    }
```

```
    boolean isFull() {
        return (count == size);
    }
```

```
    public void enqueue(int value) throws Exception {
        if(!isFull()) {
            arr[(first++) % size] = value;
            count++;
        }
        else
            throw new Exception("Queue is full!");
    }
```

```
    public int dequeue() throws Exception {
        if(!isEmpty()) {
            count--;
            return arr[(last++) % size];
        }
        throw new Exception("Queue is empty!");
    }
}
```

Notice empty and full cases look the same! That's why we need to keep somewhere the number of elements in the queue to check for overflow!

Note, right now first refers to end of queue, while last refers to head. They might as well be renamed, but it will not affect the implementation logic.