Data Structures and Algorithms

Queue Abstract Data Type

Dr. Lina Xu
lina.xu@ucd.ie

School of Computer Science,
University College Dublin

October 18, 2018

Learning outcomes

After this lecture and the related practical students should...

- understand the operations of the queue abstract data type
- be able to implement an array based queue
- be able to implement a link based queue

Table of Contents

The Queue Abstract Data Type

Concept

The queue is very similar in difficulty to the stack abstract data type to understand and implement

- A queue is a container for items
- The main idea of the queue is that insertion and removal from the data structure are based on the first-in-first-out (FIFO) principle
  - This means that whenever you remove an item from the queue, it will always be the item that has been in the queue the longest time
- Special terminology is used to describe the operations in a queue
  - Items are enqueued into the queue (insertion)
  - Items are dequeued out of the queue (removal)
  - The front of the queue is the next item to be dequeued
  - The rear of the queue is the last item that was enqueued

The Queue Abstract Data Type

Functional Specification

- Queues should work with any type of data
  - In our implementations we will only use integers to simplify the process
- Core operations of the queue are:
  - `enqueue(i)`: The integer i is inserted into the rear of the queue
  - `dequeue()`: The item in the front of the queue is removed and returned
  - `front()`: The item in the front of the queue is returned but not removed
- The queue also contains some support operations that make it easier to use
  - `size()`: Returns the number of items currently stored in the queue
  - `isEmpty()`: Returns `true` if the queue has no items and `false` if there are none

The Queue Abstract Data Type

Java Interface

```
1  public interface Queue {
2    public void enqueue(int element);
3
4    public int dequeue();
5
6    public int front();
7
8    public int size();
9
10   public boolean isEmpty();
11 }
```

The Queue Abstract Data Type

Implementation Strategies

There are two typical implementations of the queue abstract data type

- Array based implementation
  - Data is actually stored in an array
  - Extra variables required to remember what indexes store the front and rear items in the array
  - Finite capacity - limited by size of array
- Link based implementations
  - Elements are stored in custom objects called nodes
  - Object references are used to keep track of the order of the items
  - Need references to the "front" and "rear" nodes
  - Infinite capacity - Can grow and shrink as more items are added and removed

Table of Contents

Link Based Implementation

To implement the stack abstract data type without using an array first we need a class

- We will create a class called `LinkQueue`
- We need a `Node` class that is very similar to the one we used in the link based implementation of the stack
- We need to keep references to both the front and rear nodes
  - `private Node front;`
  - `private Node rear;`
- References will need to be updated during operations
- We need to keep track of the number of elements
  - `private int size;`

Node Class

```
1 public class Node{
2    int data;
3    Node next;
4    public Node(int d){
5       data = d;
6    }
7 }
```

Link Based Queue Implementation

Implementation strategy

We must decide how we will change the variables to make sure that all items are inserted in the correct place

- The initial value of the variable `size` is 0
- Every time we `enqueue` a new item we need to do the following:
  - Create a new `Node` object `n`, containing the item to be inserted
  - We change the reference of the `next` variable in `rear` to the value of `n`
  - We change the reference `rear` to the value of `n`
- What happens if the queue is empty?
- Every time we `dequeue` an item from the queue, we need to do the following:
  - Copy the data stored in the `front` variable to a temporary variable
  - Change the reference `front`, to the value of the next variable inside `front`

Enqueue Pseudocode

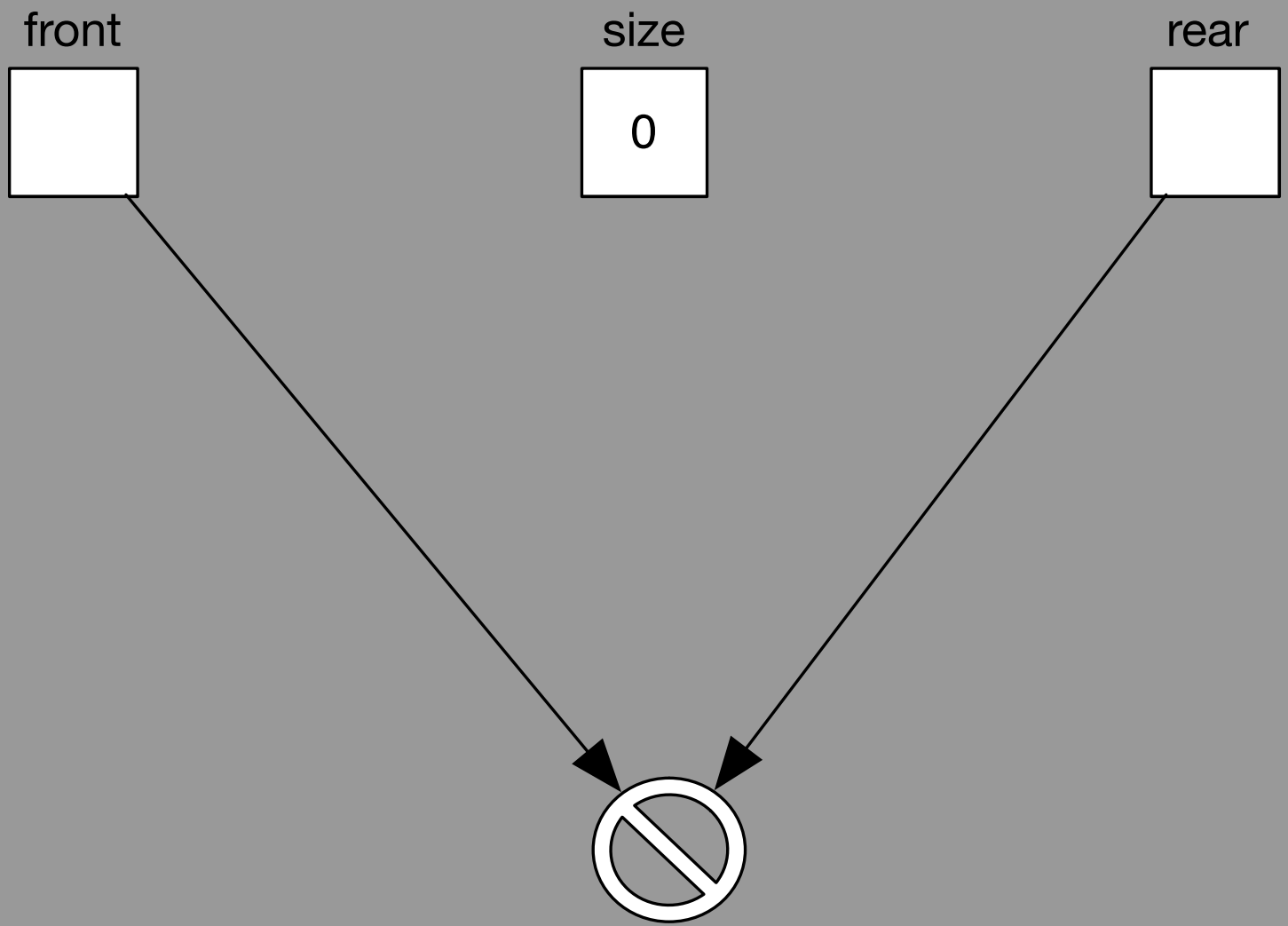
```
1  Algorithm enqueue(i):
2    Input: An integer to be stored in the queue.
3
4  N ← new Node(i)
5  if size = 0 then
6    front ← N
7  else
8    rear.next ← N
9  rear ← N
10 size ← size + 1
```

Dequeue Pseudocode

```
1 Algorithm dequeue():
2   Output: The integer that is in the front of
     the queue
3
4 i ← front.data
5 front ← front.next
6 size ← size - 1
7 return i
```

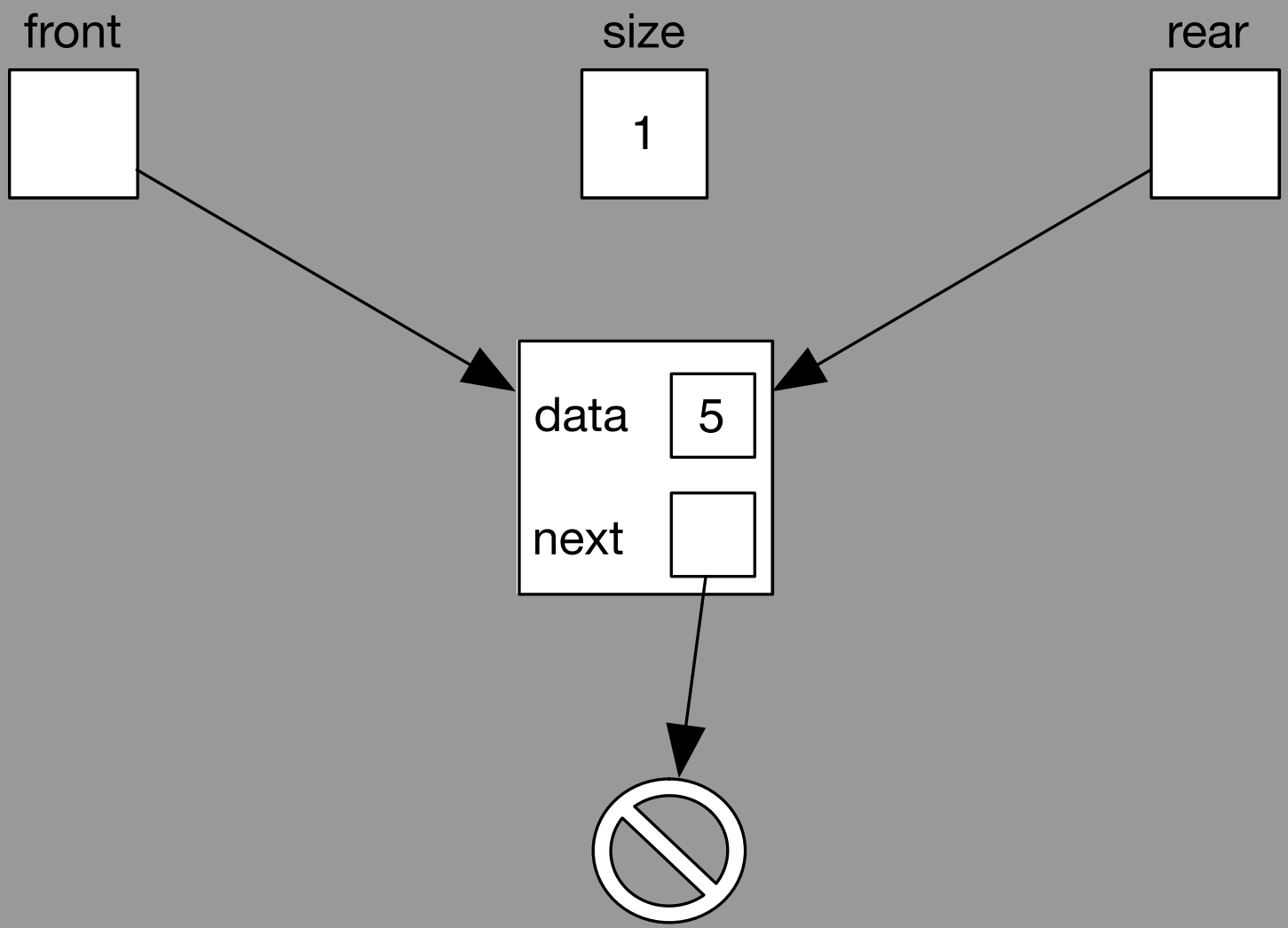
Link Based Queue Example

- Operation:



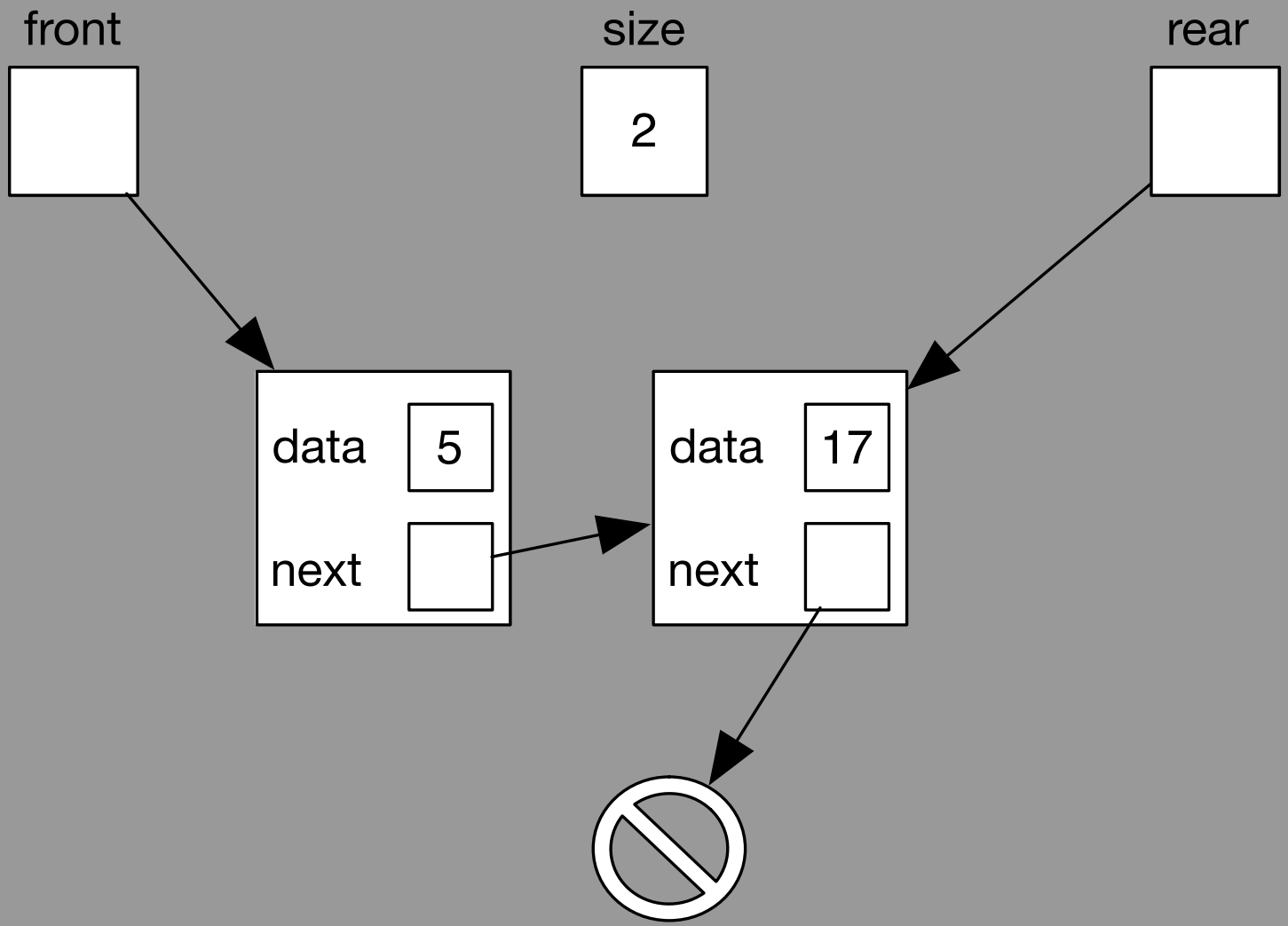
Link Based Queue Example

- Operation: `enqueue(5)`



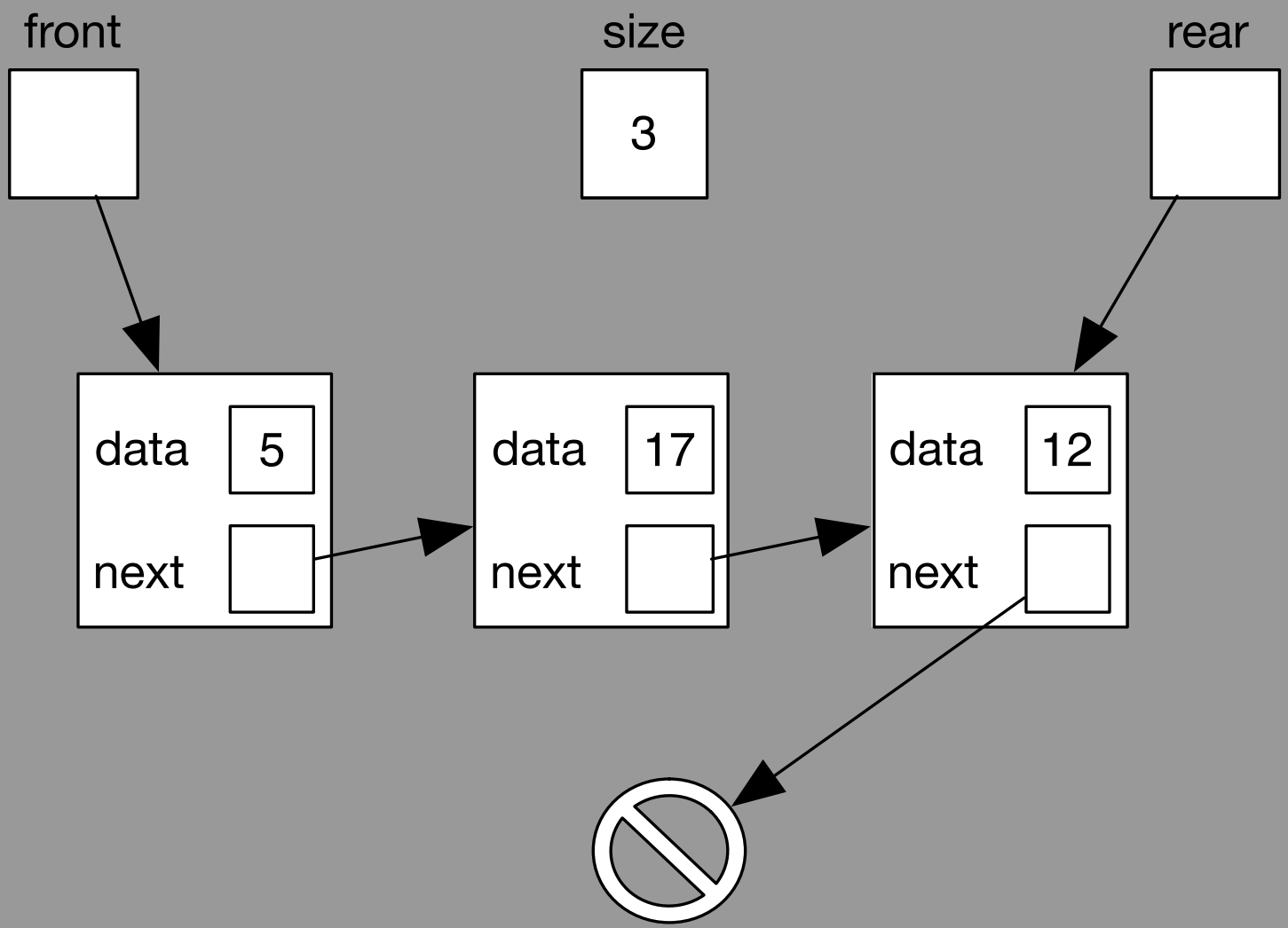
Link Based Queue Example

- Operation: `enqueue(17)`



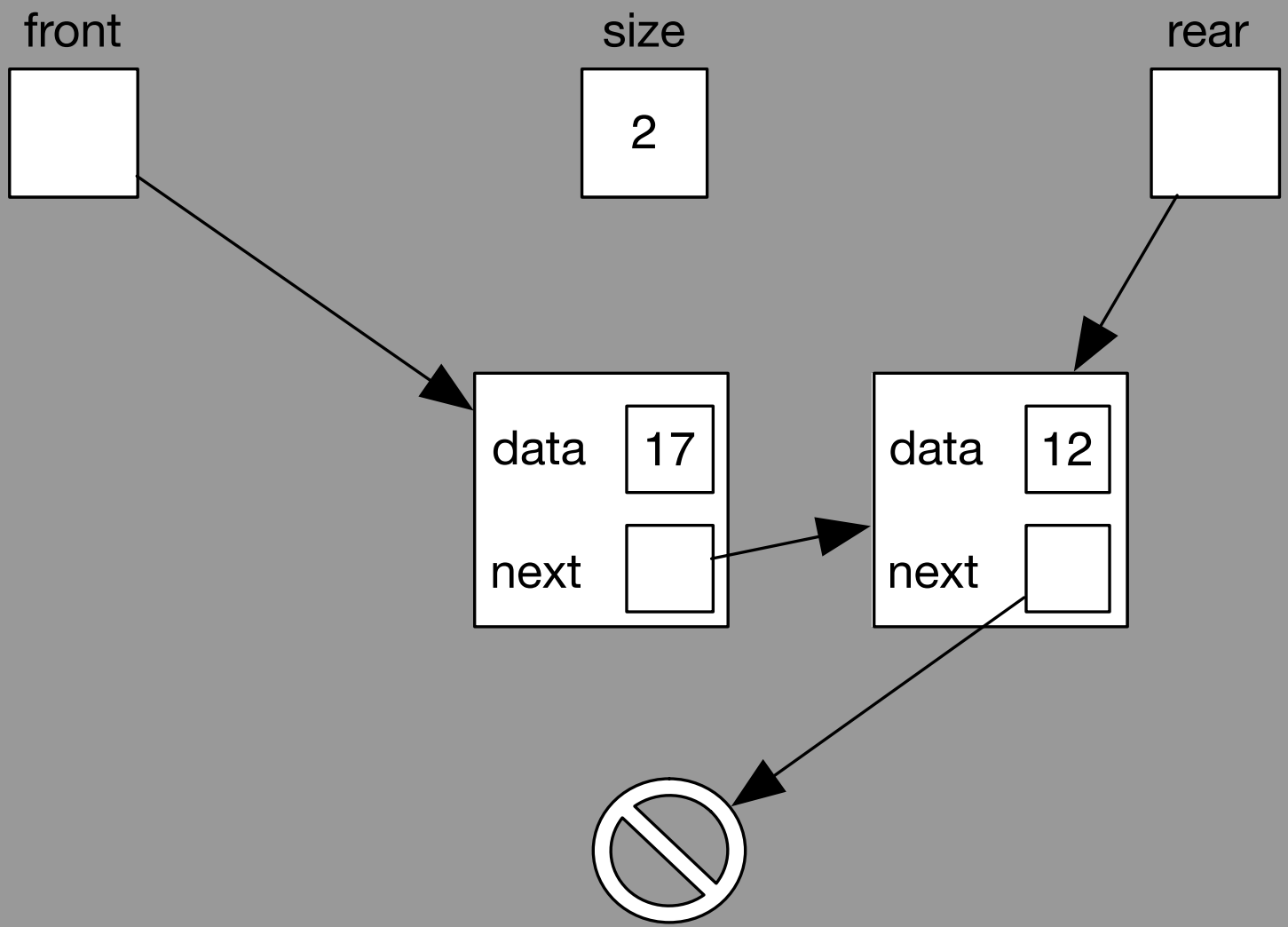
Link Based Queue Example

- Operation: `enqueue(12)`



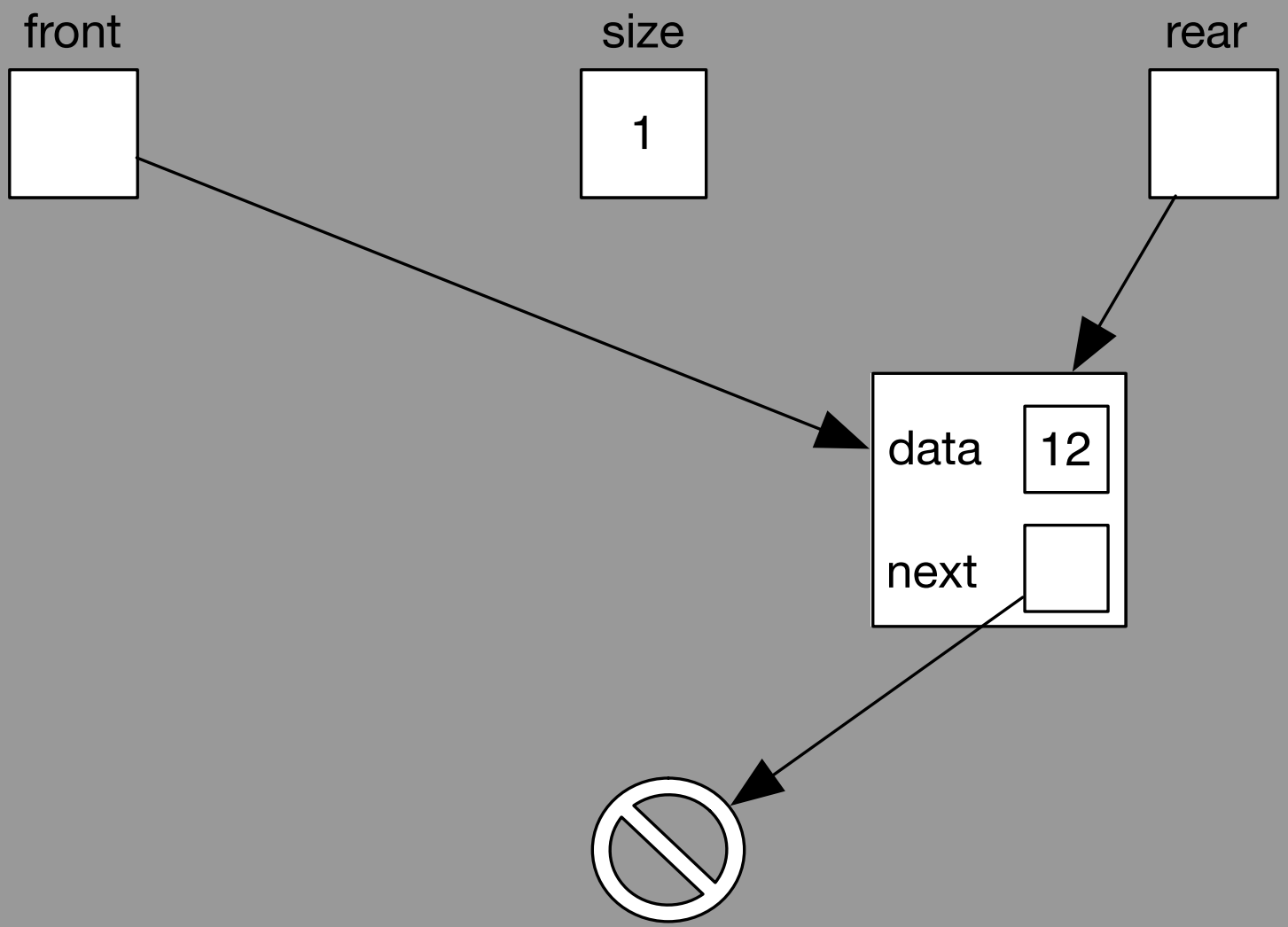
Link Based Queue Example

- Operation: `dequeue()`



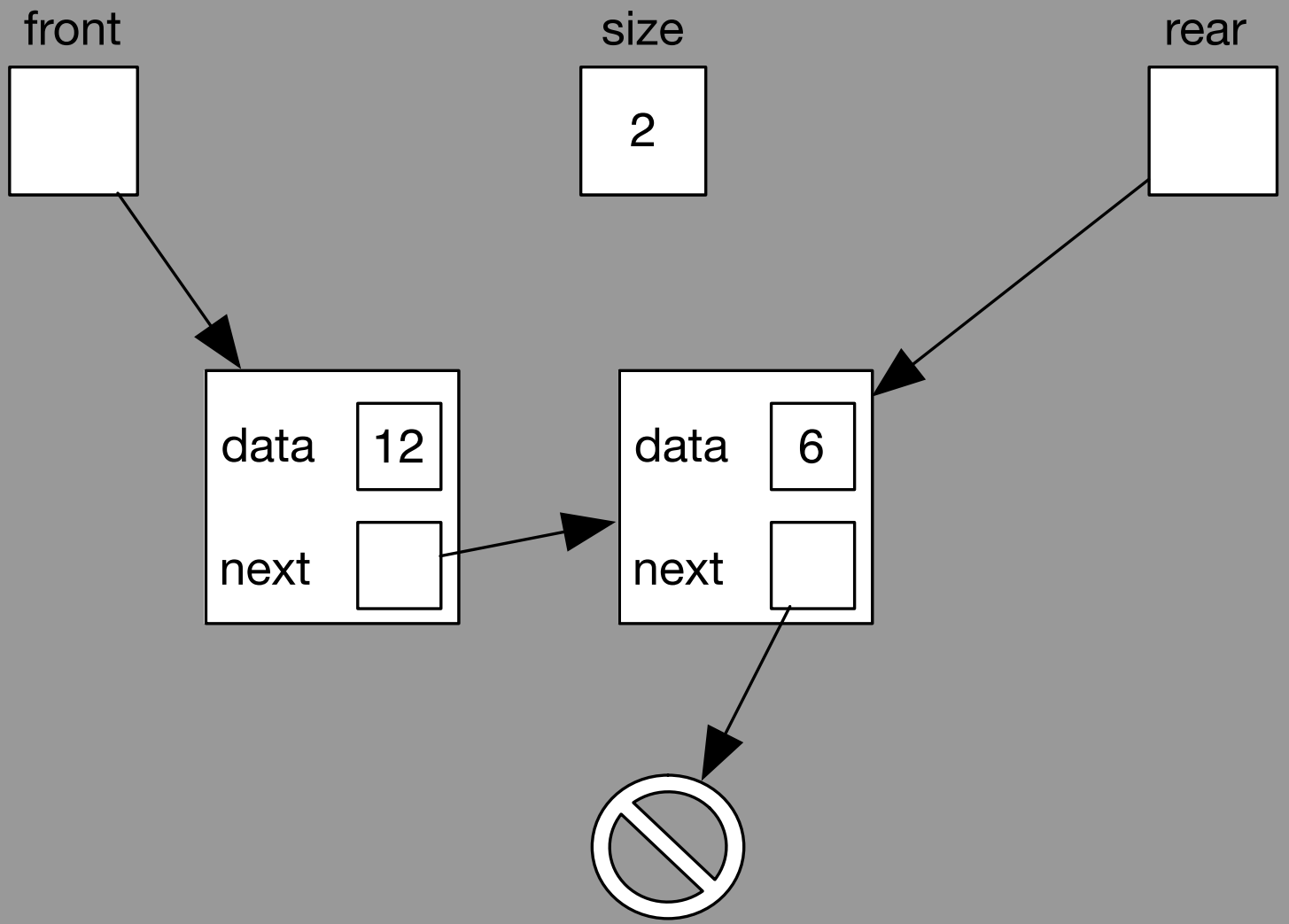
Link Based Queue Example

- Operation: `dequeue()`



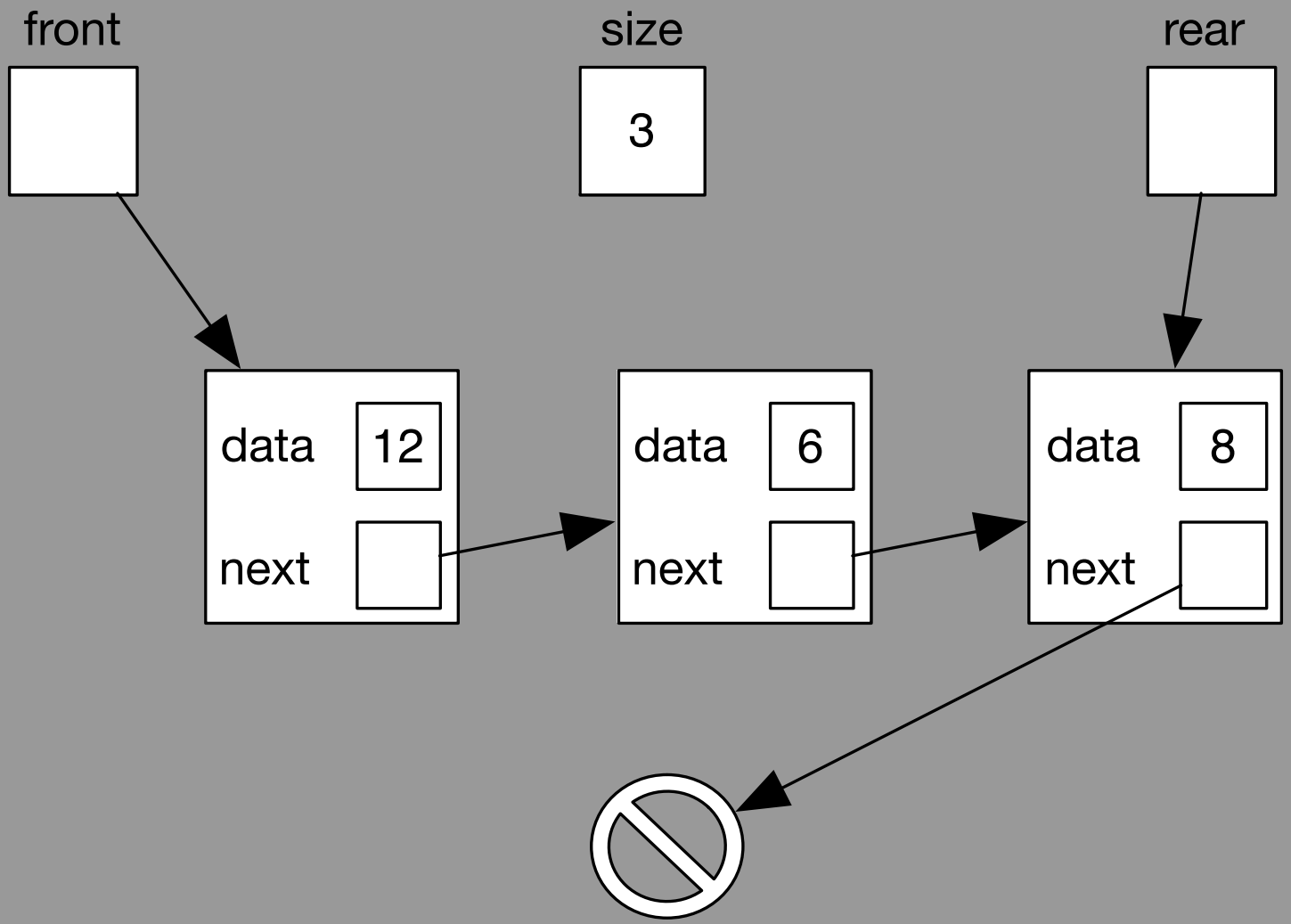
Link Based Queue Example

- Operation: `enqueue(6)`



Link Based Queue Example

- Operation: `enqueue(8)`



Link Based Queue Example

- Operation: `enqueue(23)`

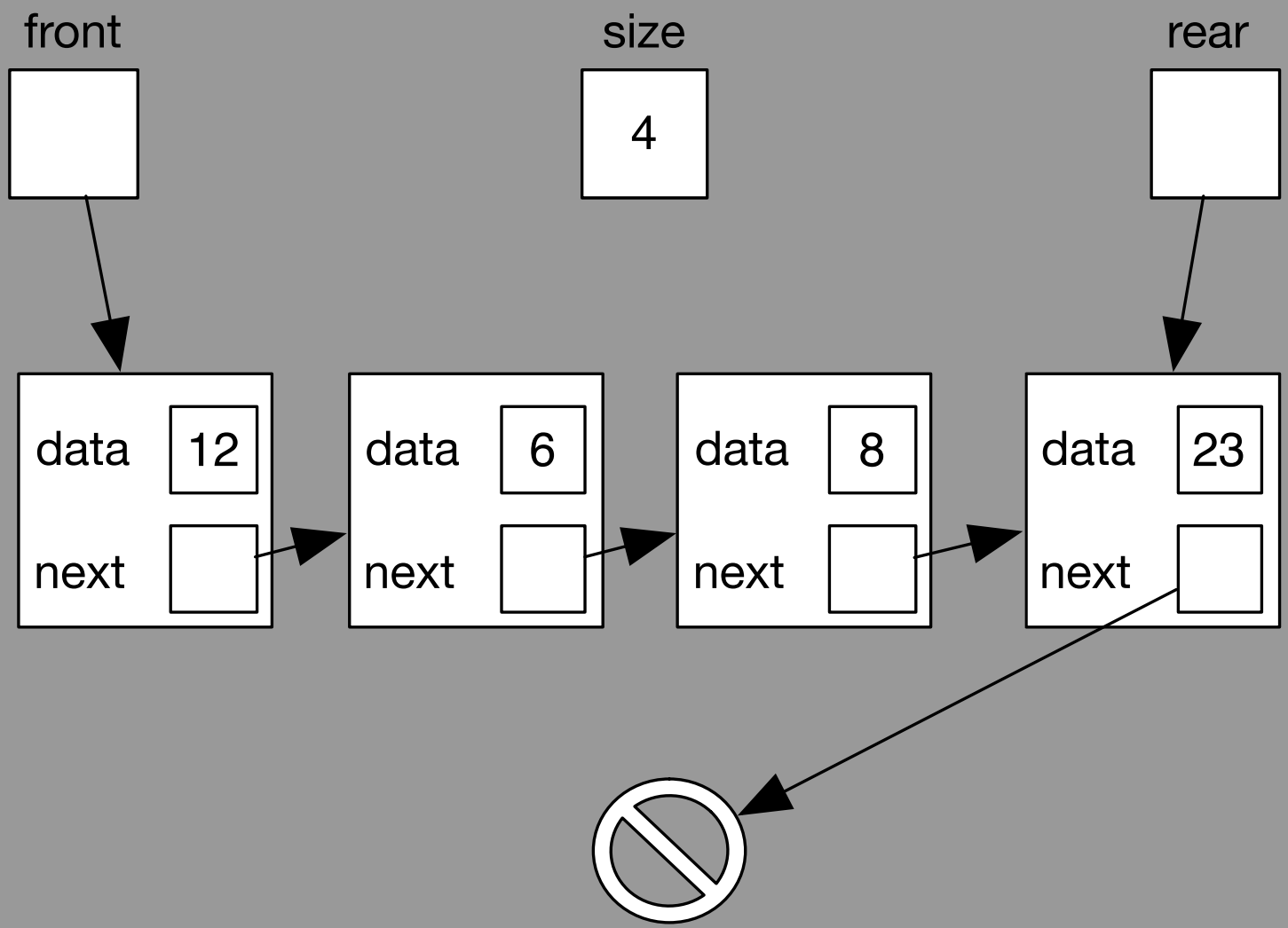


Table of Contents

Array Based Queue Implementation

To implement the queue abstract data type using an array first we need a class

- We will create a class called `ArrayQueue`
- In this class we will create an array to store the data
- The methods we write will allow the data to be accessed in the way we want
  - We can use this to enforce the FIFO rule of the stack abstract data type

Array Based Queue Implementation

Instance variables required

To implement the array based queue, we will need the following variables

- An array of `ints` to store our items
  - `private int[] values;`
- An integer variable to remember where the front item is
  - `private int front;`
- An integer variable to remember where the rear item is
  - `private int rear;`
- All variables must be declared `private` so there values cannot be accessed or changed from outside this class

Array Based Queue Implementation

Implementation Strategy

We must decide how we will change the variables to make sure that all items are inserted in the correct place.

- The initial value of the variables `front` and `rear` is 0
- Every time we `enqueue` a new item we will insert it into index rear in the array `values`
  - After inserting the item we need to increment `rear`, so that the next item is inserted in the next place in the array
- Every time we `dequeue` an item from the queue, we return the value in index front
  - After returning the item we need to increment `front`, so that the same item is not returned again

Enqueue Pseudocode

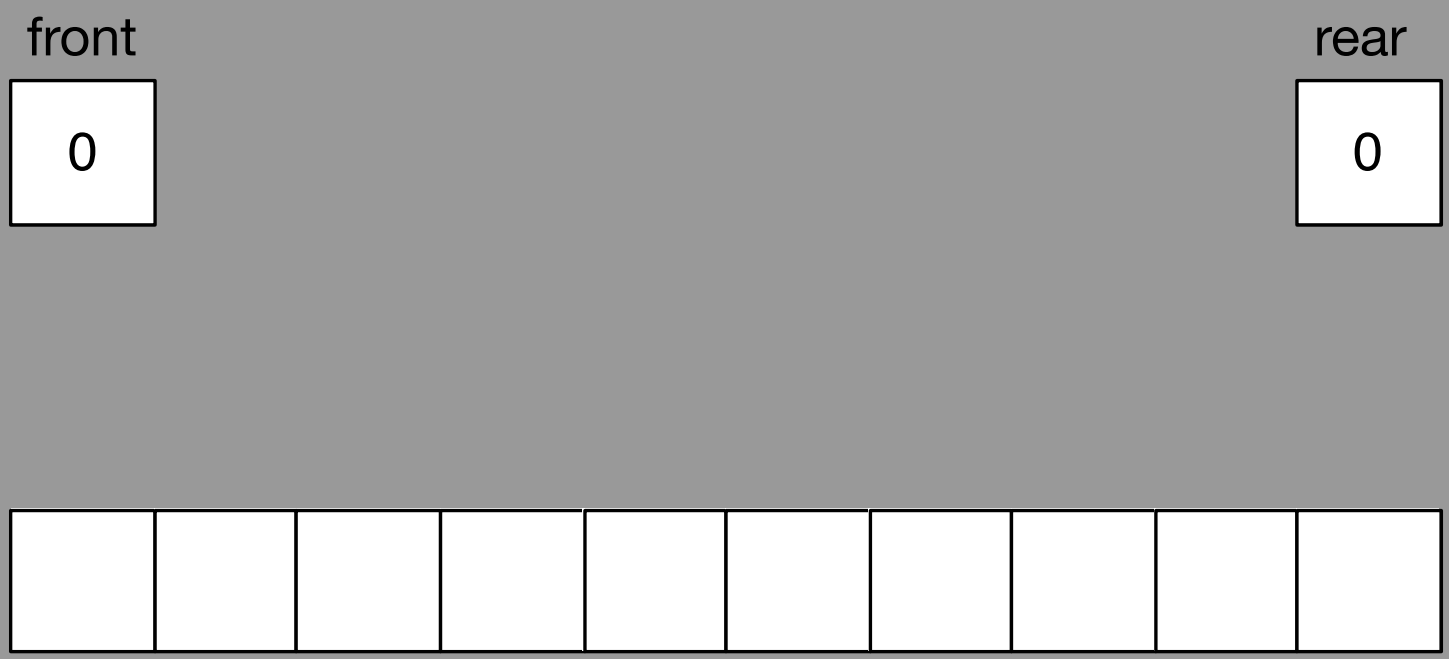
```
1 Algorithm enqueue(i):
2   Input: An integer to be stored in the queue
3
4
5 values[rear] ← o
6 rear ← rear + 1
```

Dequeue Pseudocode

```
1 Algorithm dequeue():
2   Output: The integer value stored in the
     front of the queue
3
4 i ← values[front]
5 front ← front + 1
6 return i
```

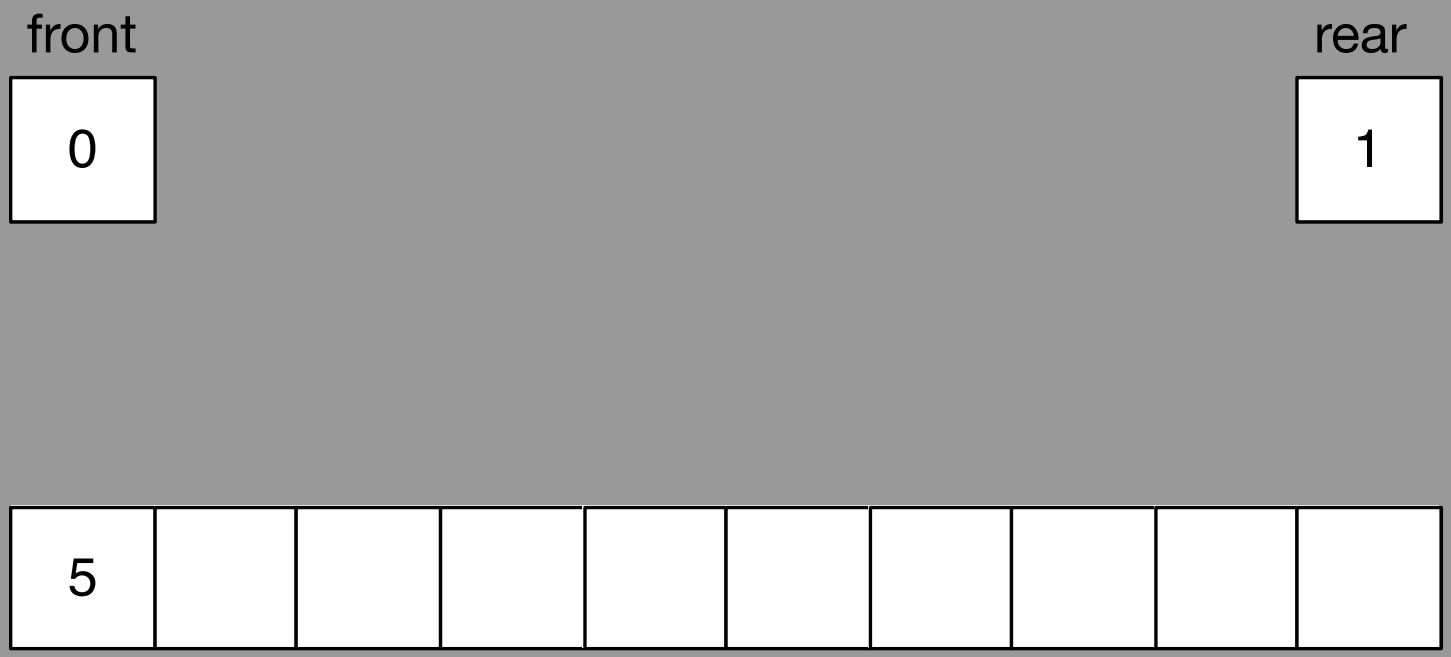
Array Based Queue Example

- Operation:



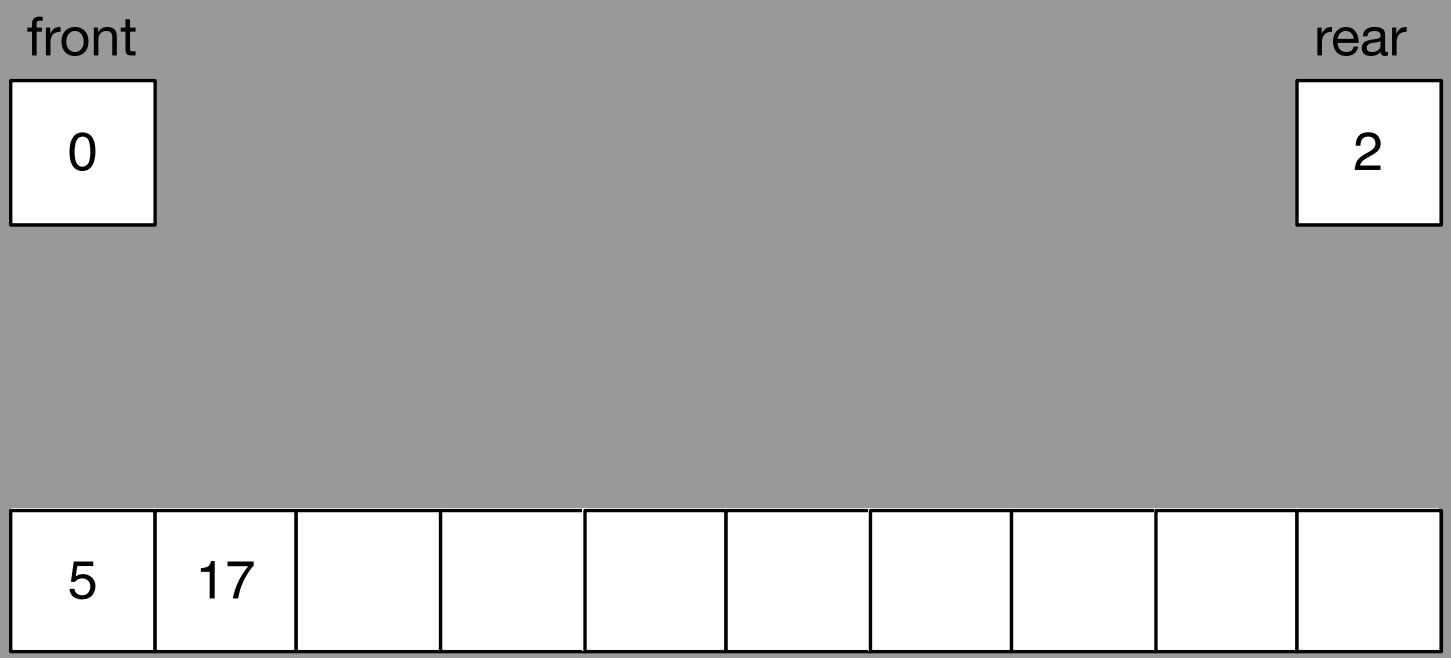
Array Based Queue Example

- Operation: `enqueue(5)`



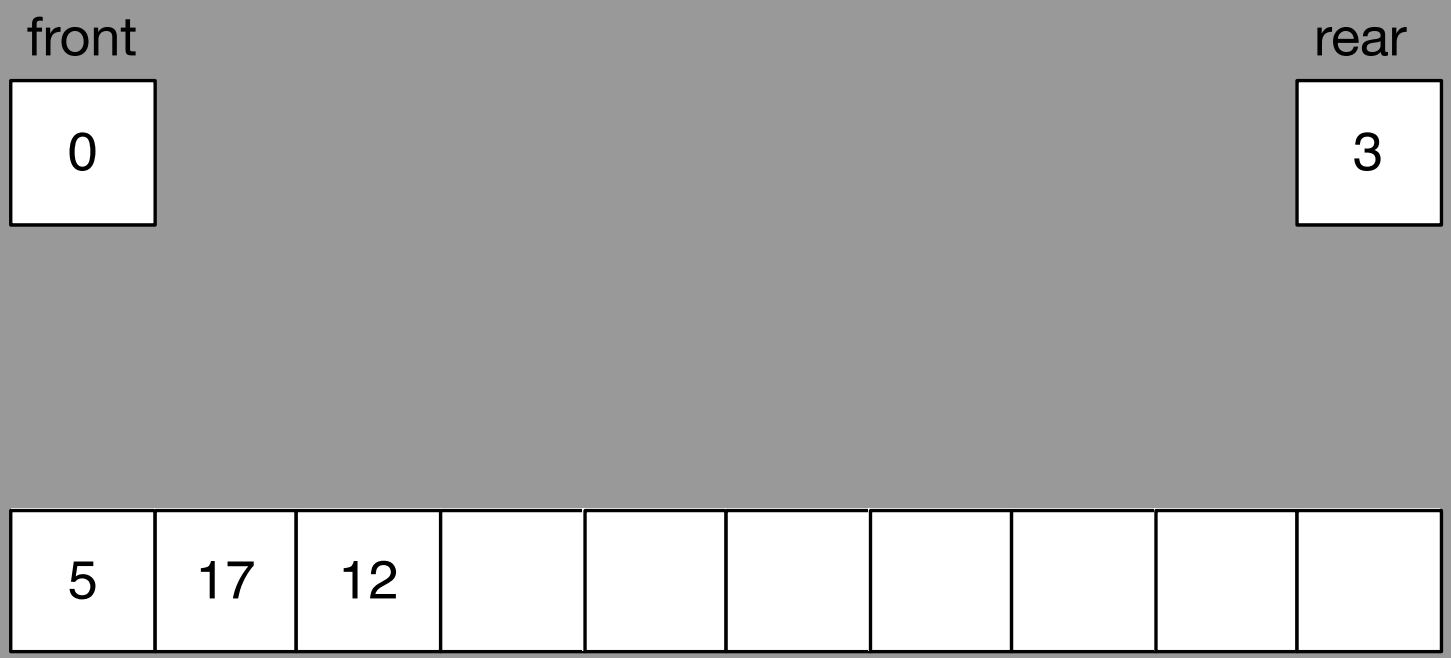
Array Based Queue Example

- Operation: `enqueue(17)`



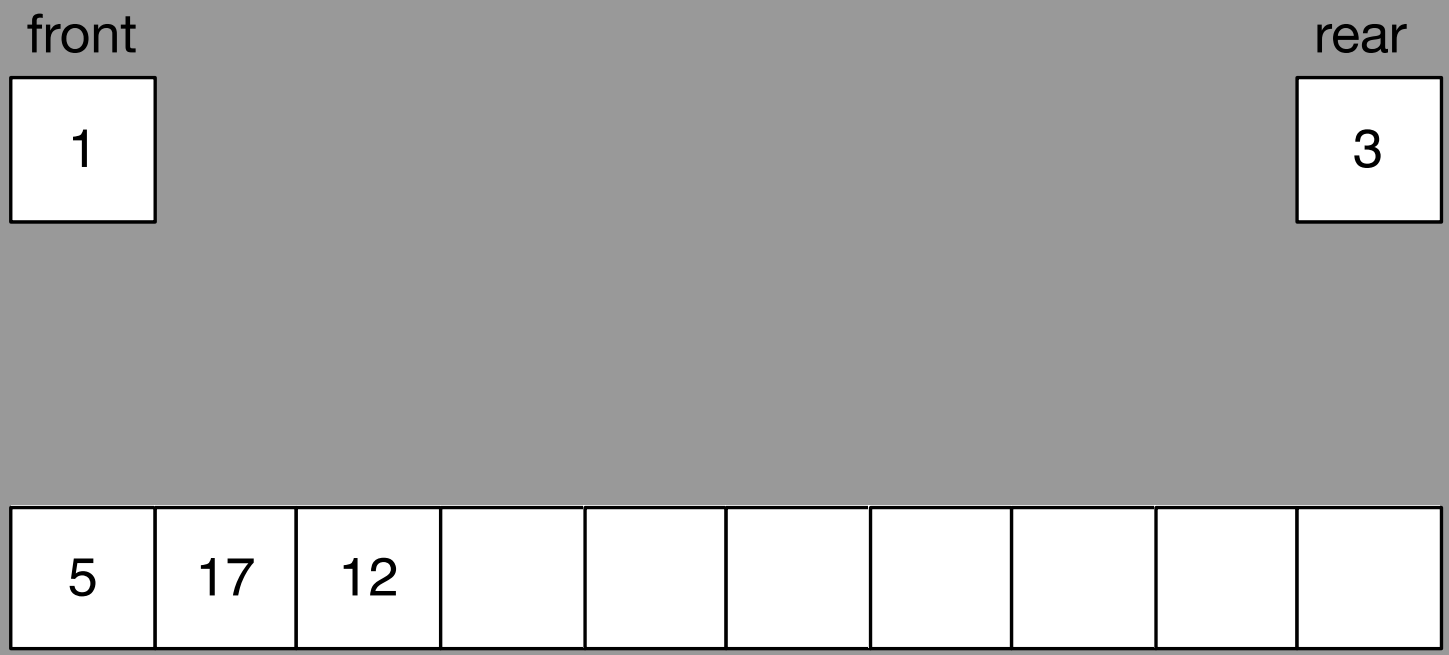
Array Based Queue Example

- Operation: `enqueue(12)`



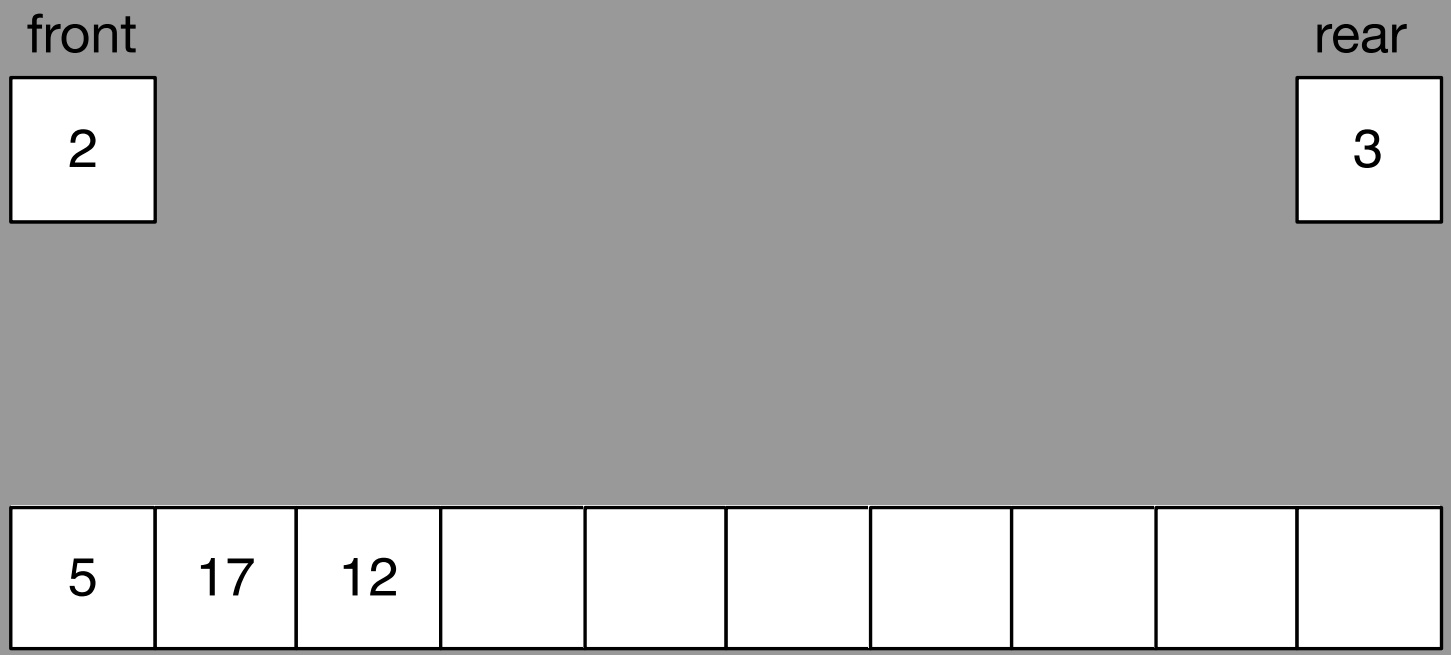
Array Based Queue Example

- Operation: `dequeue()`



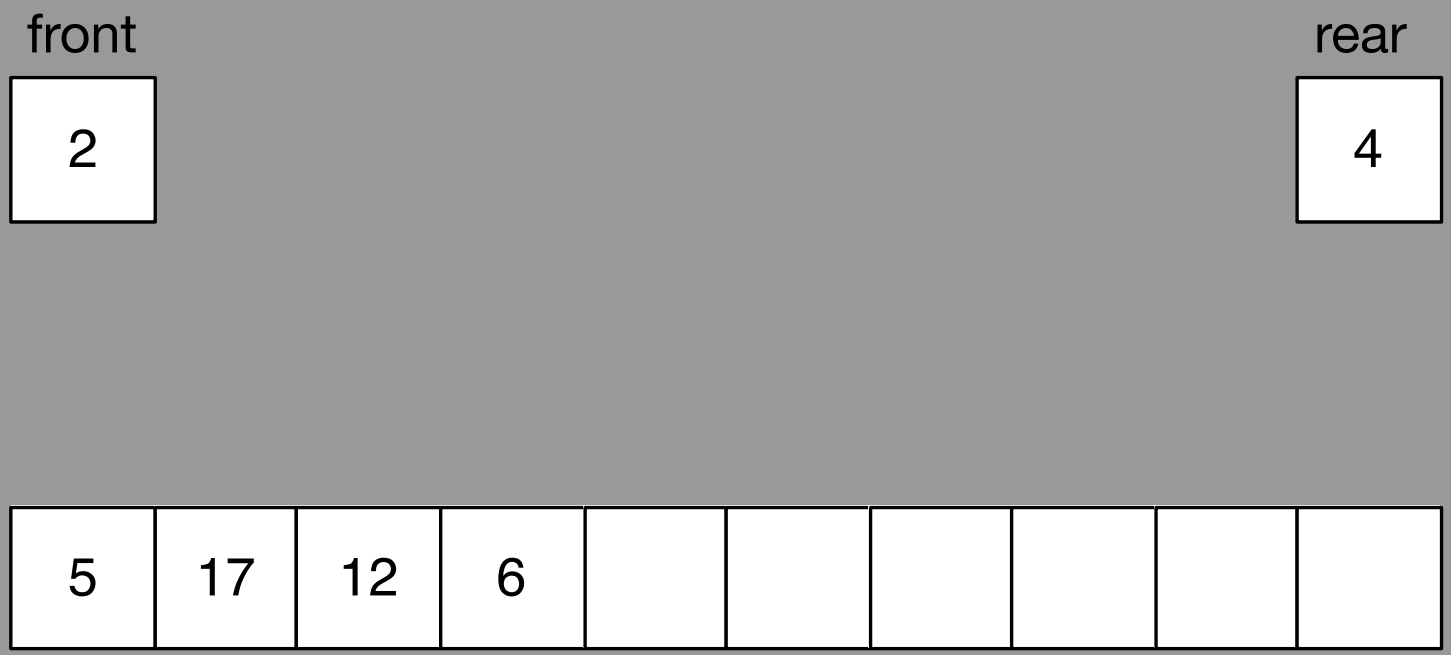
Array Based Queue Example

- Operation: `dequeue()`



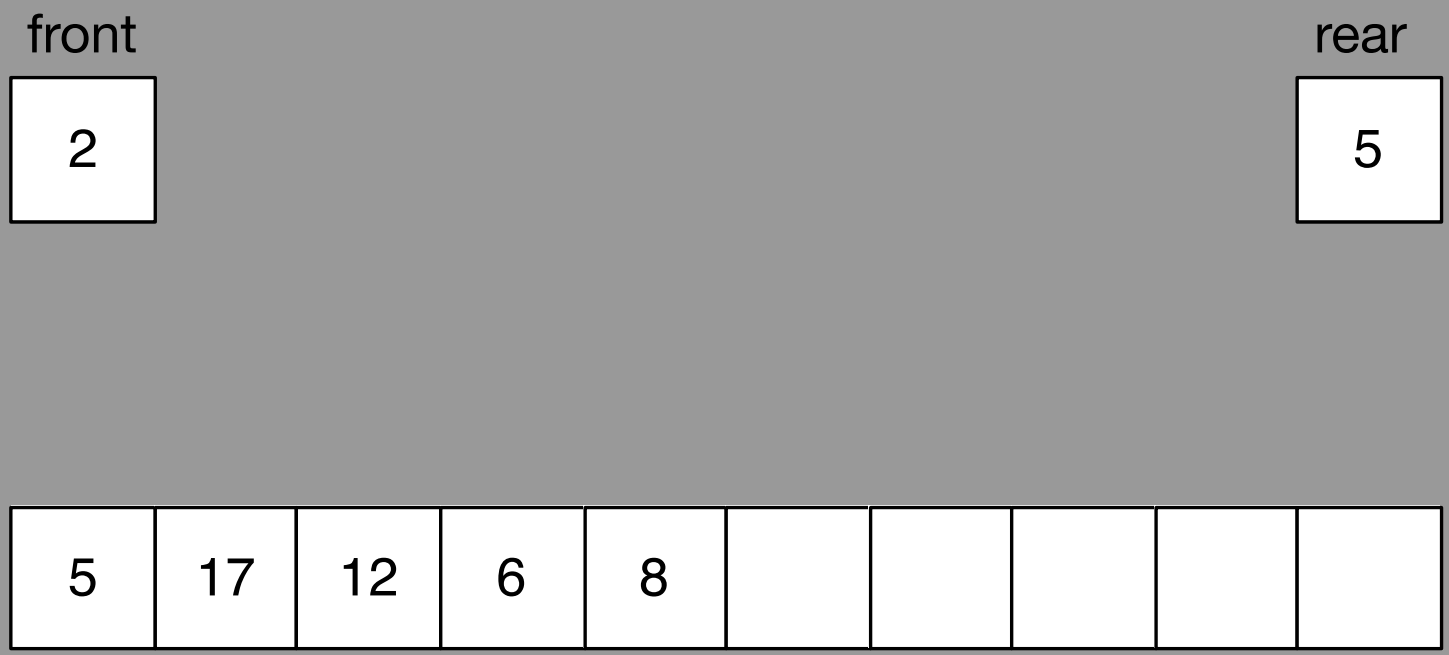
Array Based Queue Example

- Operation: `enqueue(6)`



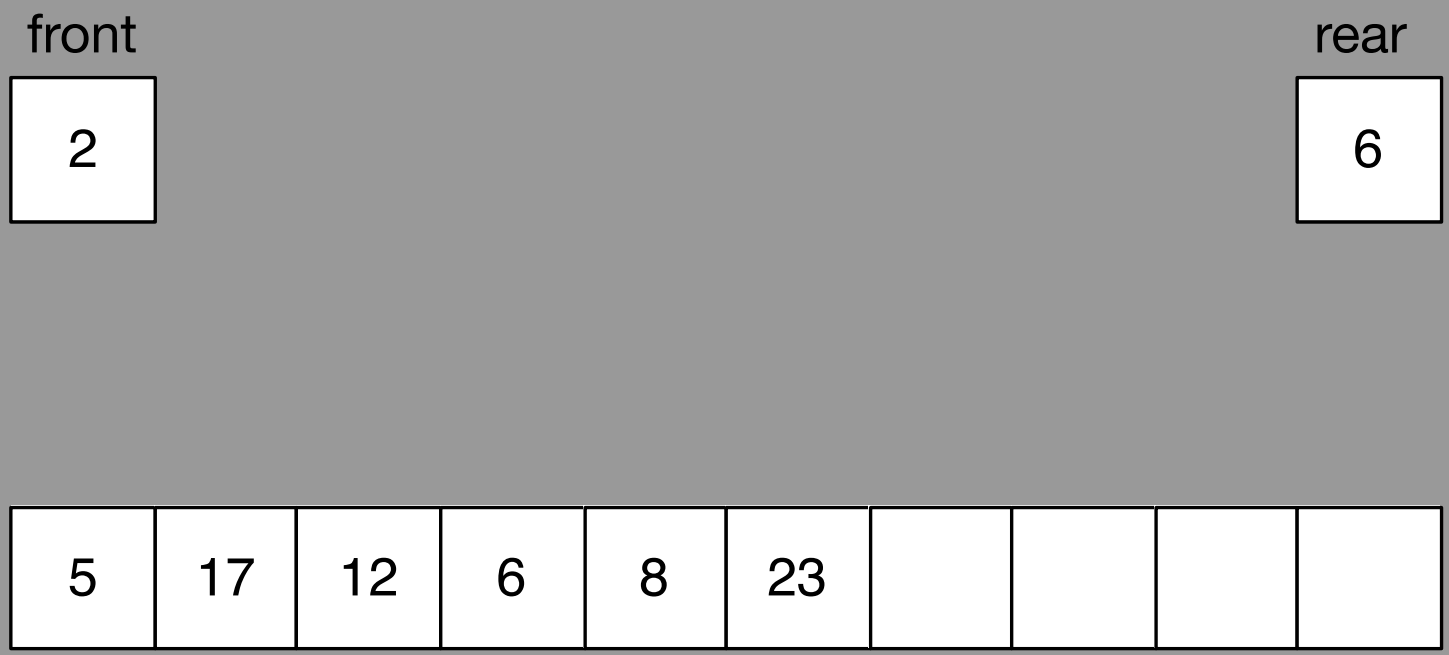
Array Based Queue Example

- Operation: `enqueue(8)`



Array Based Queue Example

- Operation: `enqueue(23)`



Calculating the Size

- So far we have not mentioned how to track the number of elements
  - We could add an extra variable and increment and decrement it every time we enqueue and dequeue
  - We could find a way to use the variables we already have to calculate the size
- The size can be calculated using the expression `rear - front`

Issues With Array Based Implementation

- There is a very significant problem with our array based implementation of the Queue
- Once we have dequeued an item, the index that contained it is never used again
- This leaves a lot of wasted space in a queue
- We could move all of the items to eliminate this problem
- But the complexity of this operation would be O(n)

Table of Contents

A Circular Array

- The solution to the problem with the array based implementation is to logically connect the beginning and end of the array
- This means that when the rear (or front) reaches the end of the array, instead of incrementing it we change the value to 0
- This brings us back to the start of the array where there is unused space

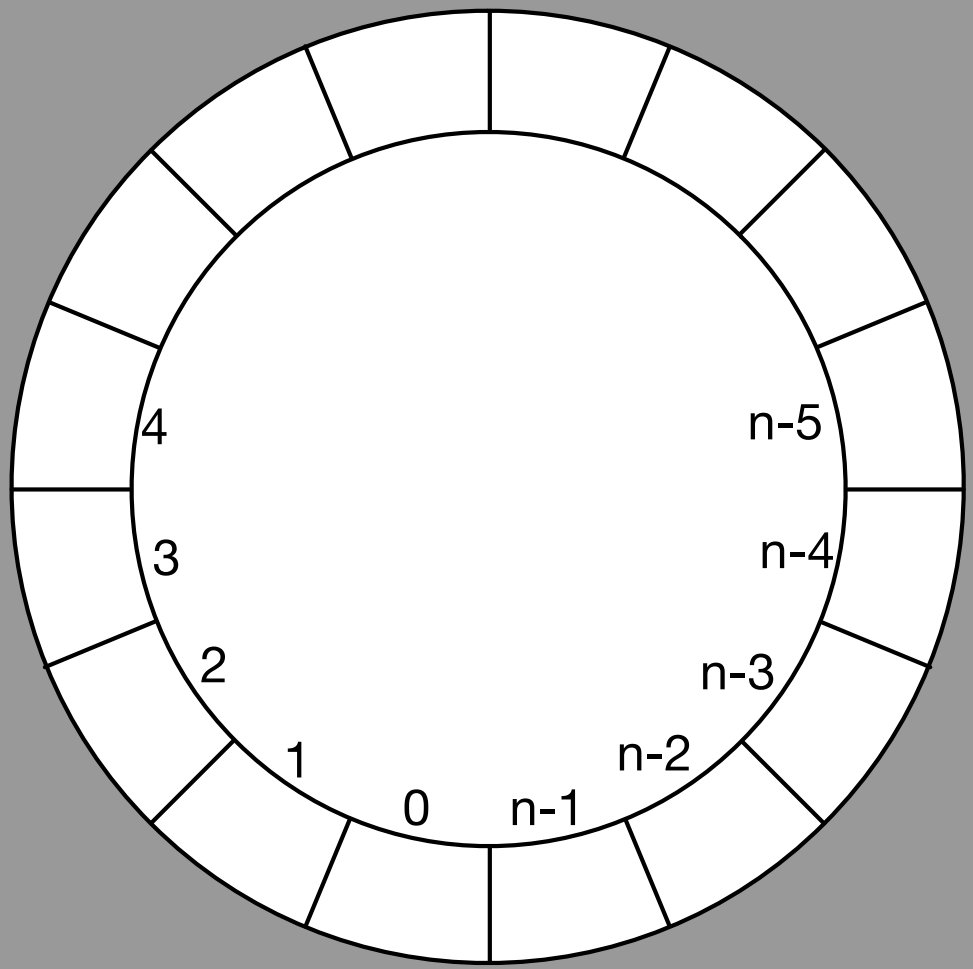
Circular Array Based Queue

front

0

rear

0



Changes to the Code

- We will create a class named `CircularQueue`
- Much of the code an operations will be the same as in the `ArrayQueue` class
- However, there will be some differences
  - Size can be difficult to calculate when the rear of the queue is less than the front
  - Instead we add an integer to keep count of the size
  - Enqueue and dequeue are unchanged, except for the calculation of the new values for rear and front
  - If the value reaches the size of the array the new value should be 0

Enqueue Pseudocode

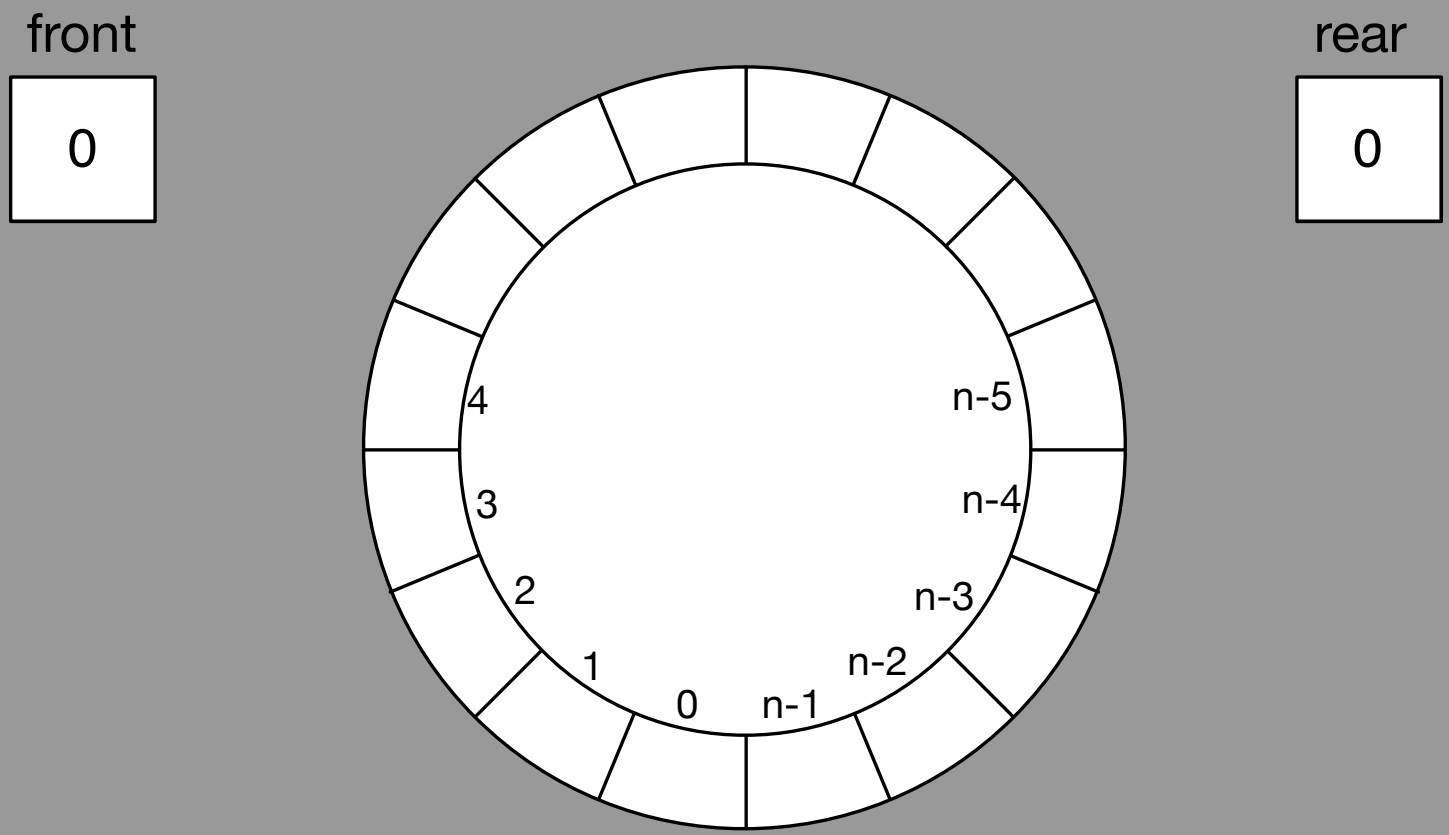
```
1 Algorithm enqueue(i):
2   Input: An integer to be stored in the queue
3
4 if size < N
5   values[rear] ← i
6   rear ← (rear + 1) % N
7   size ← size + 1
```

Dequeue Pseudocode

```
1 Algorithm dequeue():
2   Output: The integer value stored in the
     front of the queue
3
4 if size > 0 then
5   i ← values[front]
6   front ← (front + 1) % N
7   size ← size - 1
8 return i
```

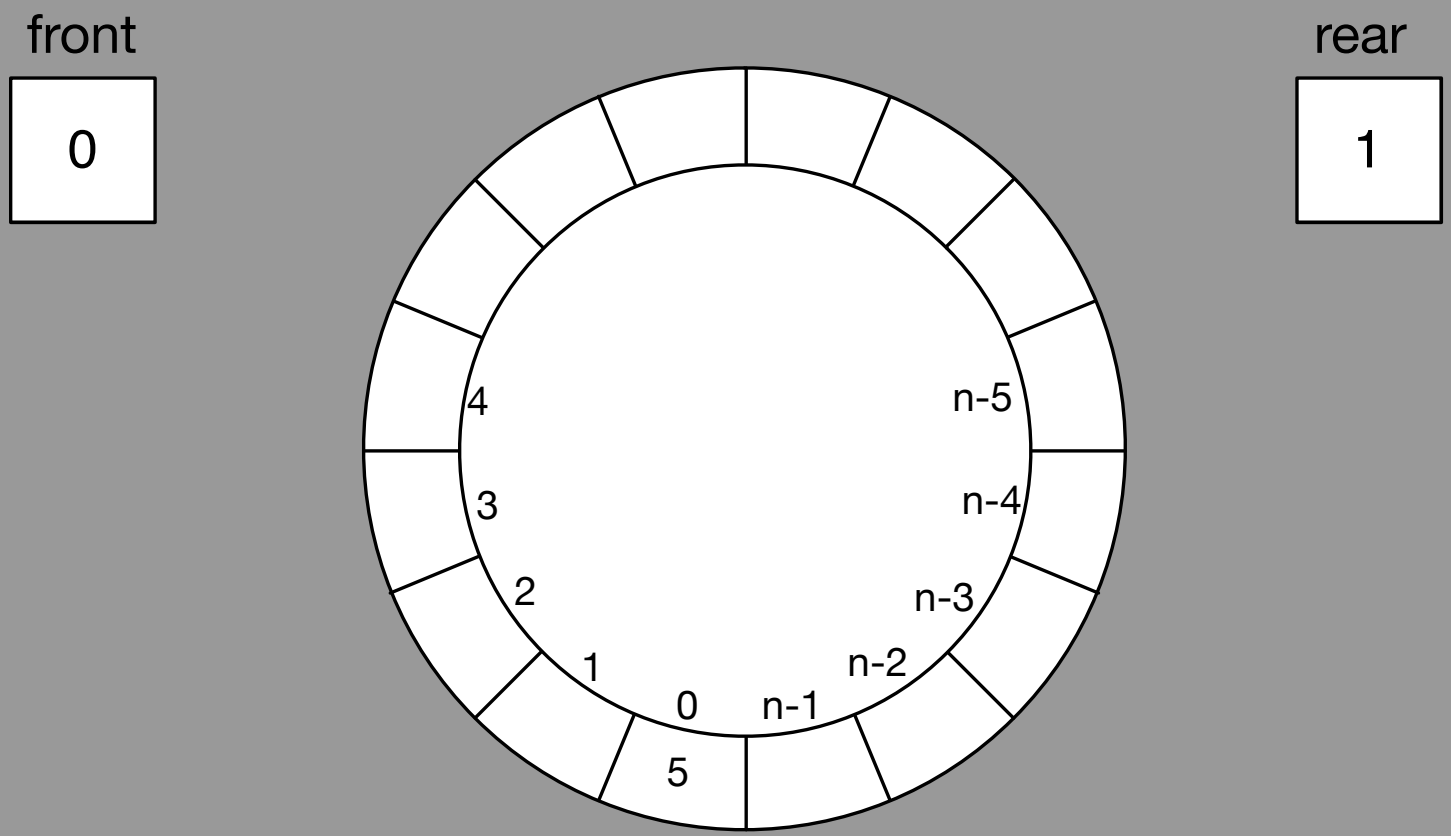
Circular Array Based Queue Example

- Operation:



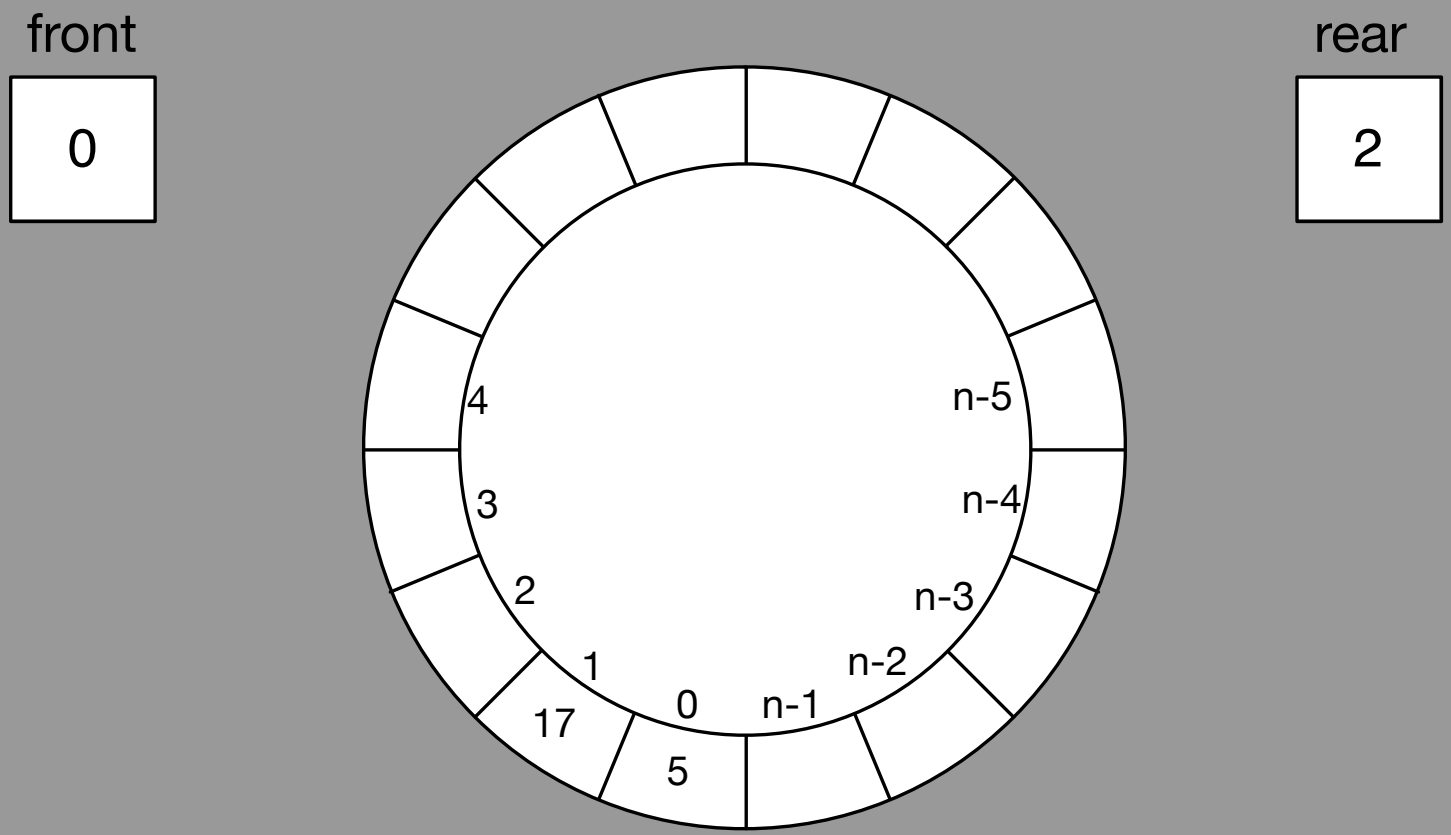
Circular Array Based Queue Example

- Operation: `enqueue(5)`



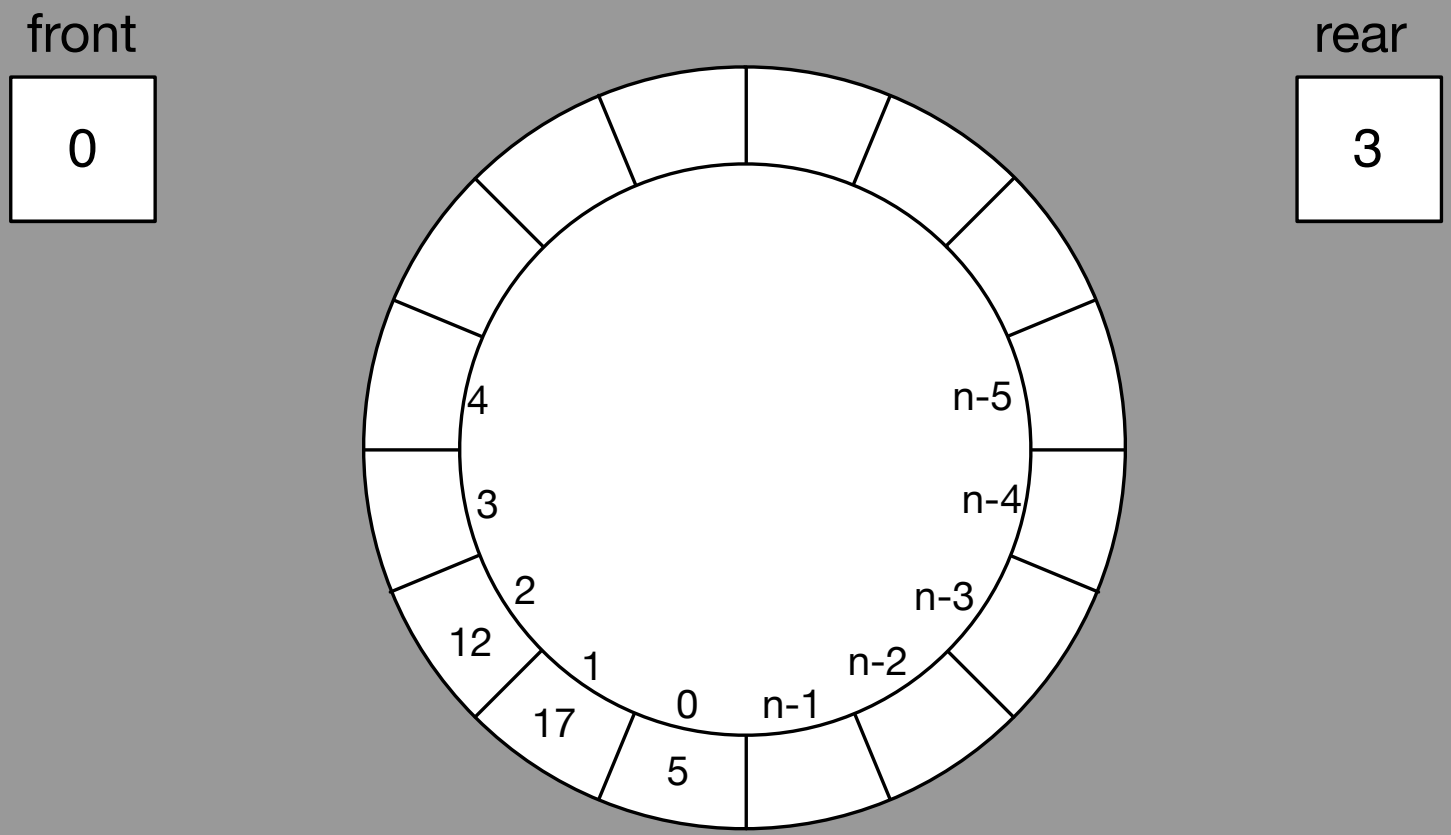
Circular Array Based Queue Example

- Operation: `enqueue(17)`



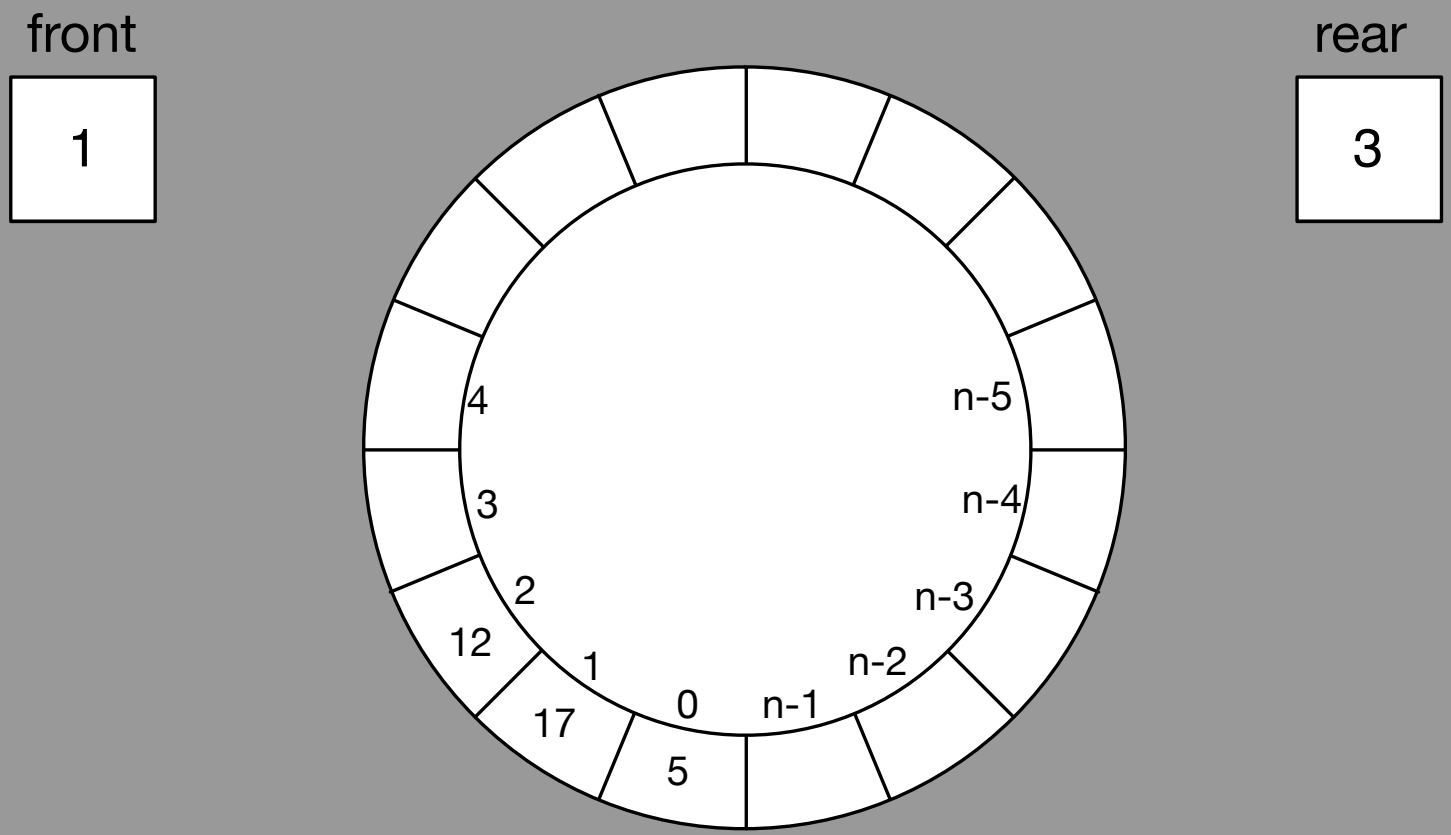
Circular Array Based Queue Example

- Operation: `enqueue(12)`



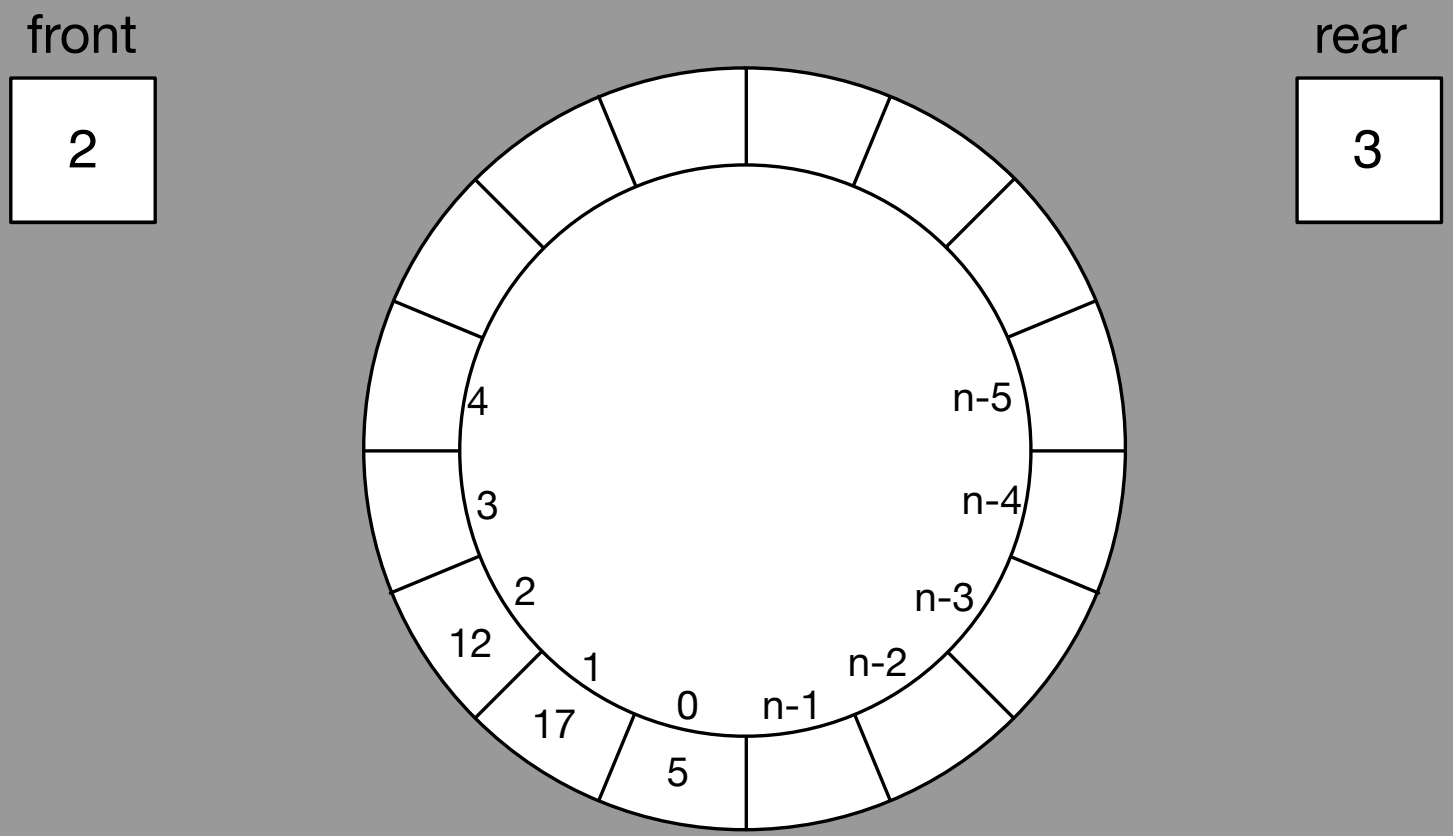
Circular Array Based Queue Example

- Operation: `dequeue()`



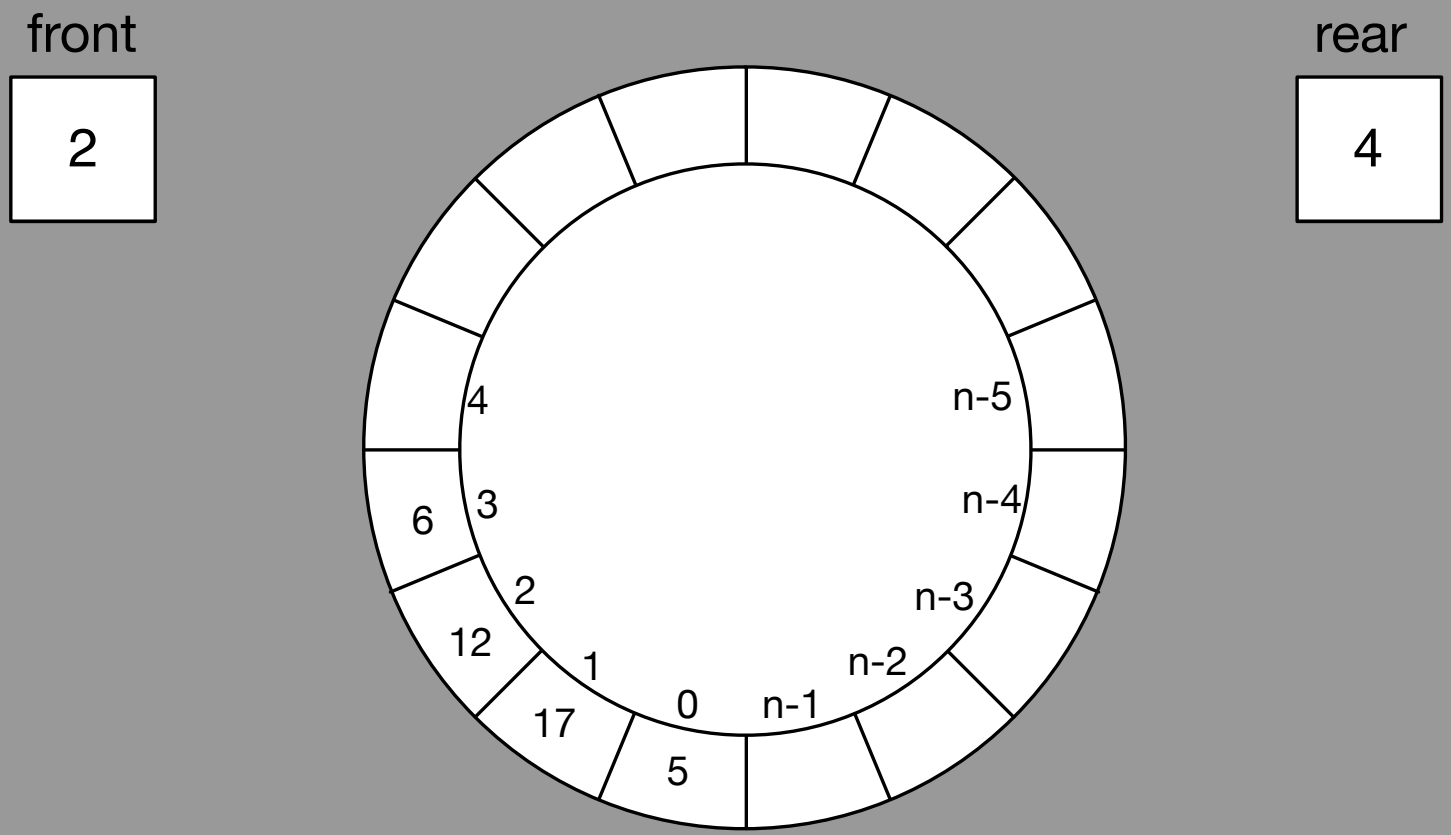
Circular Array Based Queue Example

- Operation: `dequeue()`



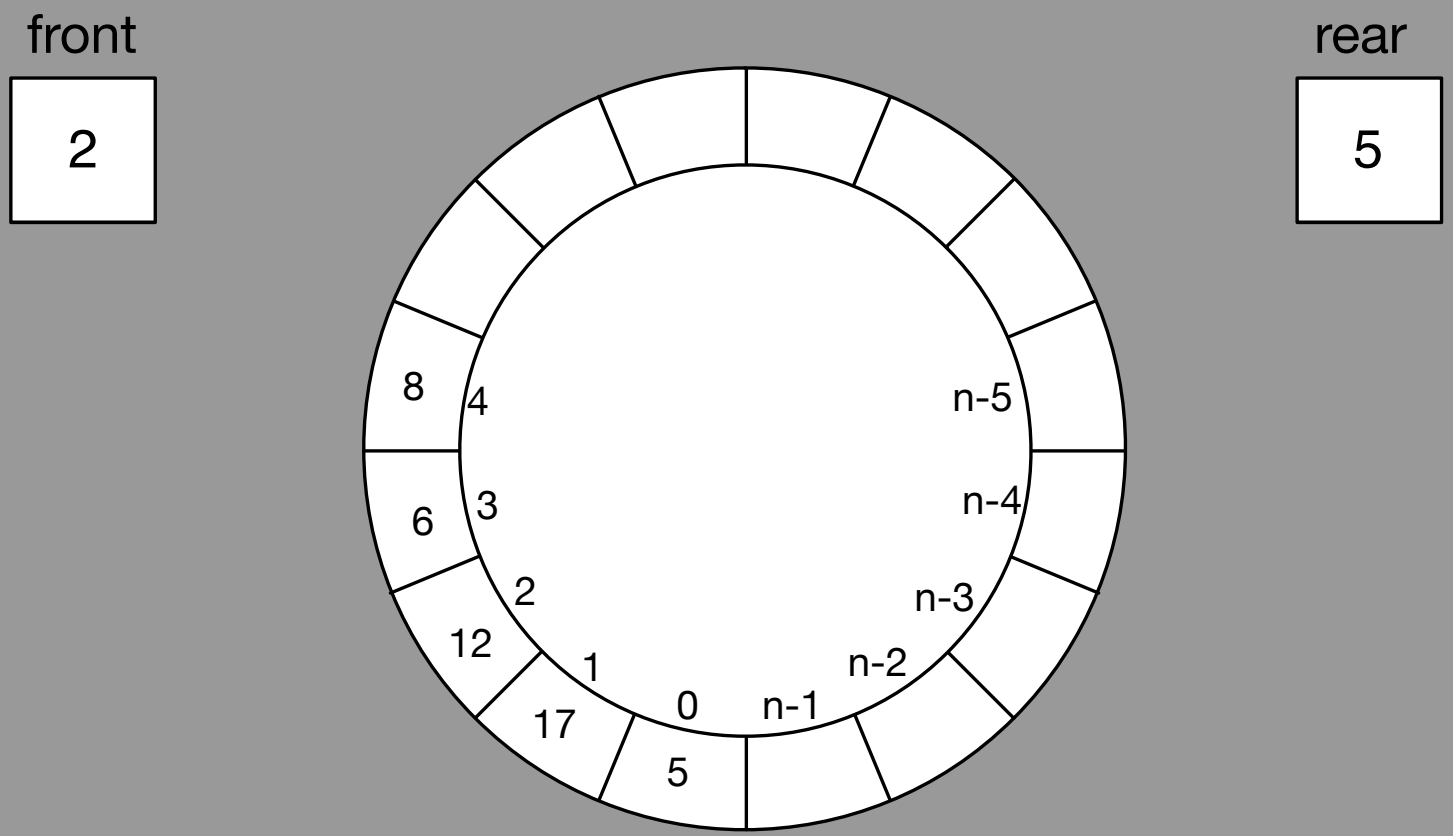
Circular Array Based Queue Example

- Operation: `enqueue(6)`



Circular Array Based Queue Example

- Operation: `enqueue(8)`



Circular Array Based Queue Example

- Operation: `enqueue(23)`

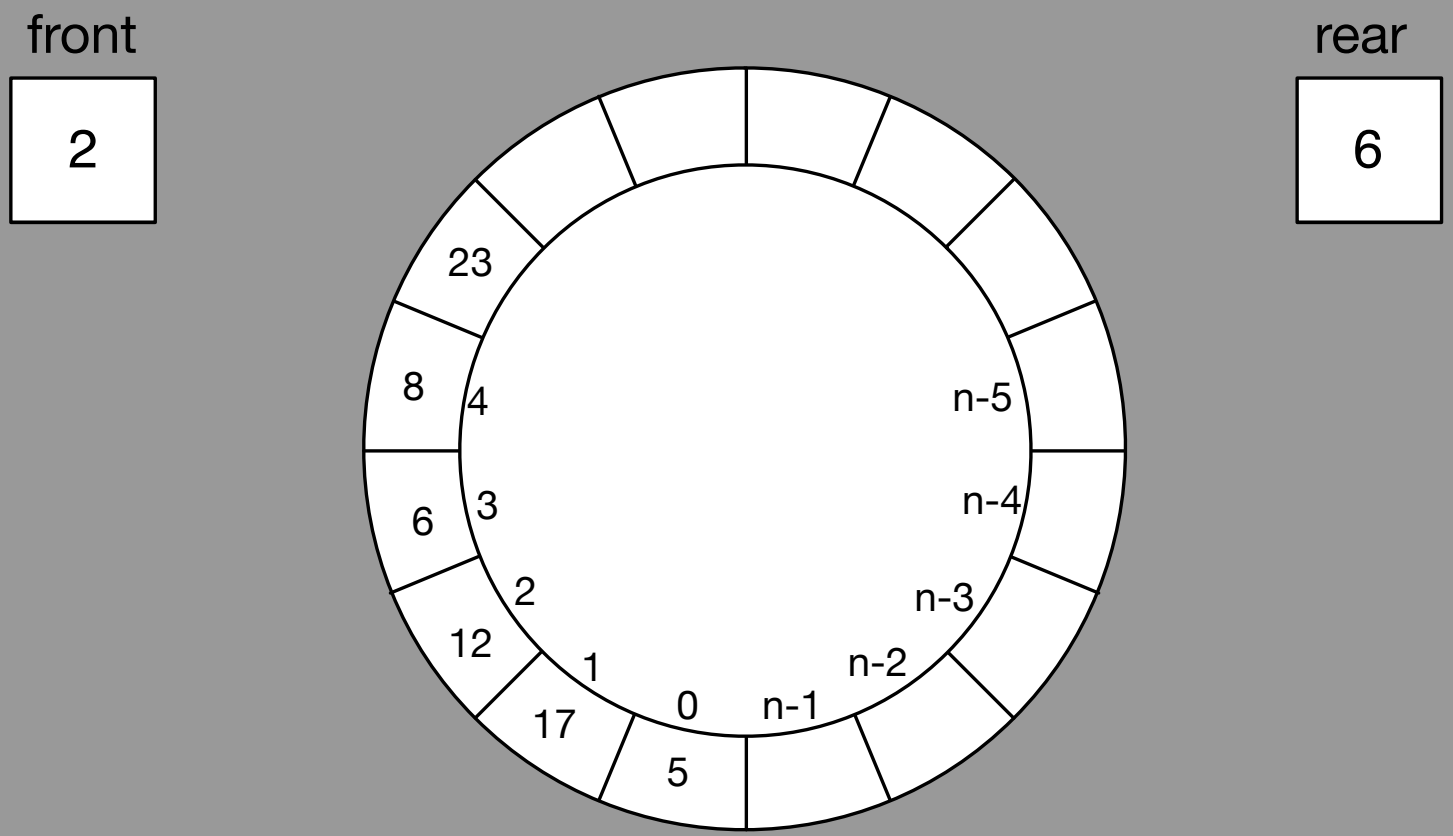


Table of Contents

Comparing Implementations

Operation	Link Based Complexity	Array Based Complexity
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
size	O(1)	O(1)
isEmpty	O(1)	O(1)
front	O(1)	O(1)

Further Information and Review

If you wish to review the materials covered in this lecture or get further information, read the following sections in Data Structures and Algorithms textbook.

- 5.2 - Queues