Lecture 5

Abstract Data Types

Abstract Data Types (ADT)

- We are familiar with data types in Java
  - For example some primitive data types: int, float, double, boolean, or object types such as String, StringBuilder, JButton and File
  - We know how to define a new data type

- Each data type encapsulates inside:
  - Some **data** and its representation in memory
    - For classes these are the instance variables
  - The **operations** by which the data can be manipulated
    - For classes these are the methods

Abstraction in Computer Science

- *Abstraction* - the process of extracting only **essential property** from a real-life entity
- In CS: Problem → storage + operations

Abstract Data Type (**ADT**):

result of the process of abstraction

- A specification of ***data to be stored*** together with a set of ***operations*** on that data
- **ADT = Data + Operations**

ADT is a mathematical concept (from *theory of concepts*)

ADT is a **language-agnostic** concept

- Different languages support ADT in different ways
- In Java, use *class* (abstract class, interface) construct to declare a new ADT

ADT includes:

- **Specification:**
  - What needs to be stored
  - What operations should be supported
- **Implementation:**
  - Data structures and algorithms used to meet the specification

ADT: Specification vs. implementation

Specification and **implementation** have to be disjoint:

- **One** specification
- **One or more** implementations
  - **Using different data structures**
  - **Using different algorithms**

Specification is expressed by defining the public variables and methods

Implementation implements these declared methods

- ADT is a specification of a data type (data + operations) which is **separated from its implementation**
- It is just an idea of "what we want"

Example: ADT Big Integer

- **Specification**:
  - Data to be stored: arbitrarily sized whole numbers
  - Supported operations: +, -, *, /, %

- **Sample implementation:**
  - **BigInteger** type in Java

Using ADT vs. implementing it

- In order to use BigIntegers in our programs, we ONLY need to know **what they are** and **what their operations do**
  - We do NOT need to know their implementation details
  - How the BigInteger is represented in memory?
  - What algorithm is used for the division operation?
- For the purposes of **using** BigInteger in your program, the implementation details do not matter
- These are abstracted out of our view!

- In this course you will look at ADTs both from the **user's** and **implementer's** point of view

ADT specification in Java: *interface*

- We can use **Interfaces** to specify the idea of the desired type:
  - Interface is the most abstract concept in Java
  - Specifies a set of methods – a set of behaviors or abilities
  - Does not specify how those methods should actually be implemented
  - Does not specify how the data needs to be structures in memory
- The textbook will typically use interfaces as ADT specifications and classes as ADT implementations

ADT specification in Java: class

- However any **class with its public methods** can be used as an ADT:
  - The body of the method can be implemented differently without affecting the users of the class
  - The data storage can be replaced with more efficient data structures
  - All this without affecting the users of a class – as long as the public interface (method signatures) does not change

The art of Abstraction

- We model a real-life problem using a concept of ADT:
  - Take a problem
  - Think what is important to solve this particular problem
  - Abstract the multitude of properties of a real-life entity into a minimal set of data + supported operations

Enso – abstract symbol for absolute enlightenment: strength, elegance, the universe, and mu (the void)

◀ This is NOT the kind of abstraction we are talking about

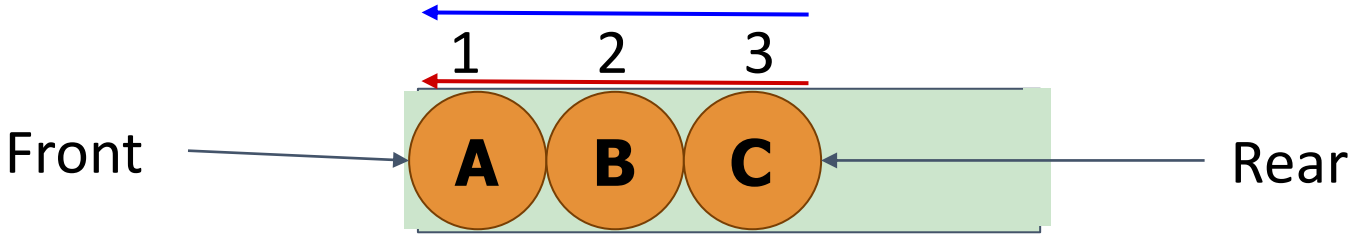
Example: Doctor queue

We want to model a list of patients waiting in the Hospital ER

- When a new patient arrives - we should be able to add him to the queue
- When the doctor calls for the next patient, we should be able to remove the patient from the front of the queue

Abstraction of Patient List: Queue

- If these are the only two required operations, then we can model the Doctor queue using a ***Queue* ADT**
- The elements in the Queue are sorted by timestamp: from the earlier to the later
- This ADT is called a *FIFO Queue* (First In First Out)



Specification

Queue: Abstract Data Type which supports the following operations:

- ***Enqueue(e)***: adds element *e* to collection
- ***Dequeue()***: removes and returns least recently-added key
- Boolean ***IsEmpty( )***: are there any elements?
- Boolean ***IsFull()***: is there any space left?

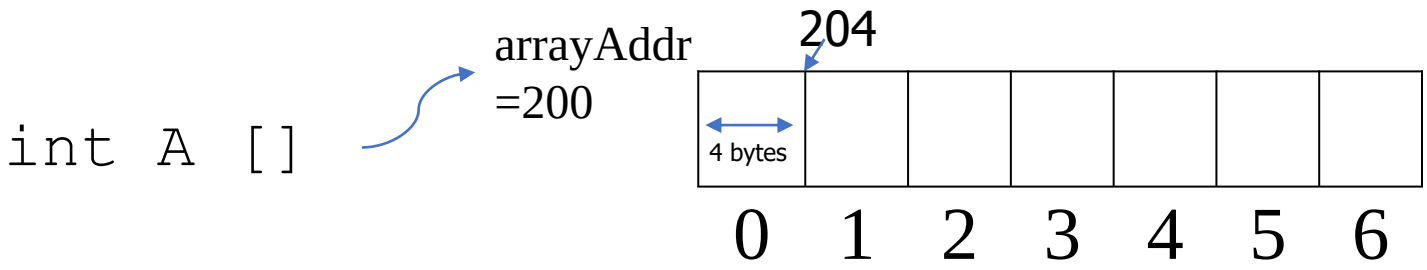
Implementation

- We need a data structure to store the elements of the queue
- The only data structure we used so far is an array

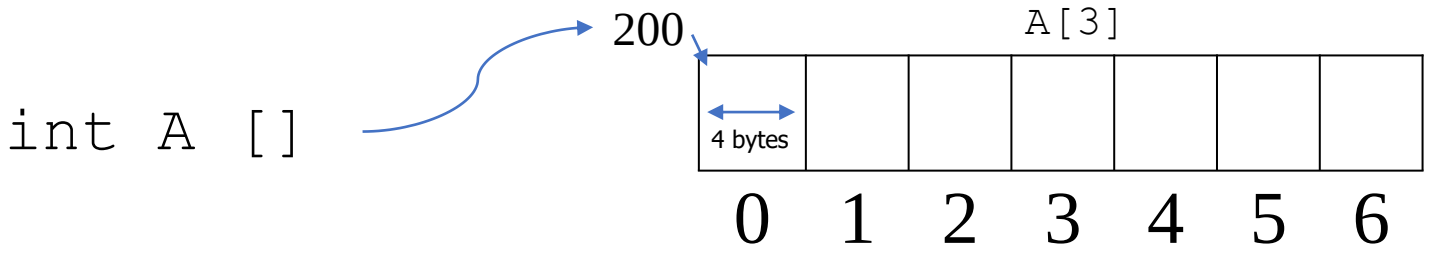
- So let's try to maintain the queue as an array and implement the required operations

Definition

Array is a **contiguous** area of memory containing **equal-sized** elements indexed by contiguous integers



Array elements must stay contiguous



- Because of contiguous arrangement we can directly access any element of the array by index *i*.

- The address of A[i] is computed as:

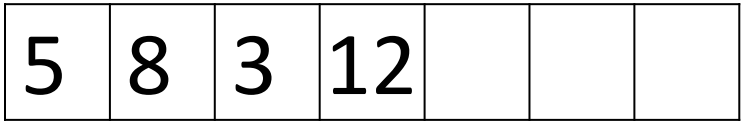
  arrayAddr + elemSize × (*i*)

  and we can jump directly to this address

- For example, address of A[3] = 200+3*4 = 212
- Array elements must stay contiguous (no gaps)

Edit operations: add/remove

- We can use space allocated for the array to store a variable number of elements
- We just need to distinguish between the array capacity (the number of allocated slots) and the actual number of elements in the array (we will call it size)
- This is especially useful if we have array of references – we can keep track of the number of actual objects in the array

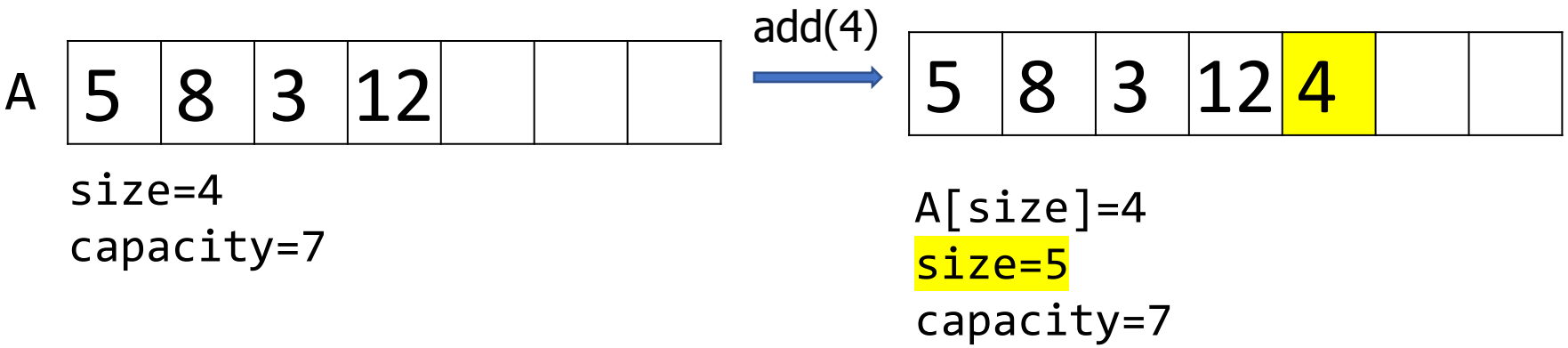


`size=4`
`capacity=7`

We can store the actual number of the elements added to the array in a variable `size`

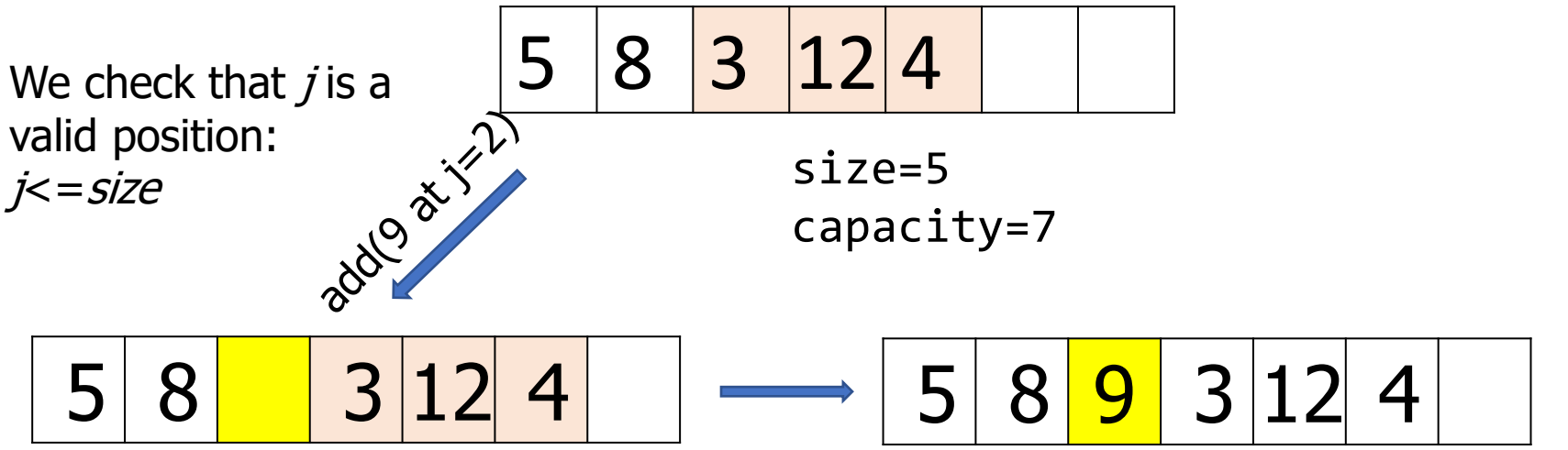
Add to the end of A

1. As long as capacity permits, add new element to the empty slot at position *size*
2. Increment *size* by 1



Add in the middle of A

- We must keep elements consecutive: array is a contiguous sequence in memory
- If we want to insert an element at some position *j* of A, we must move all the elements from *j* to *size*-1 to the right



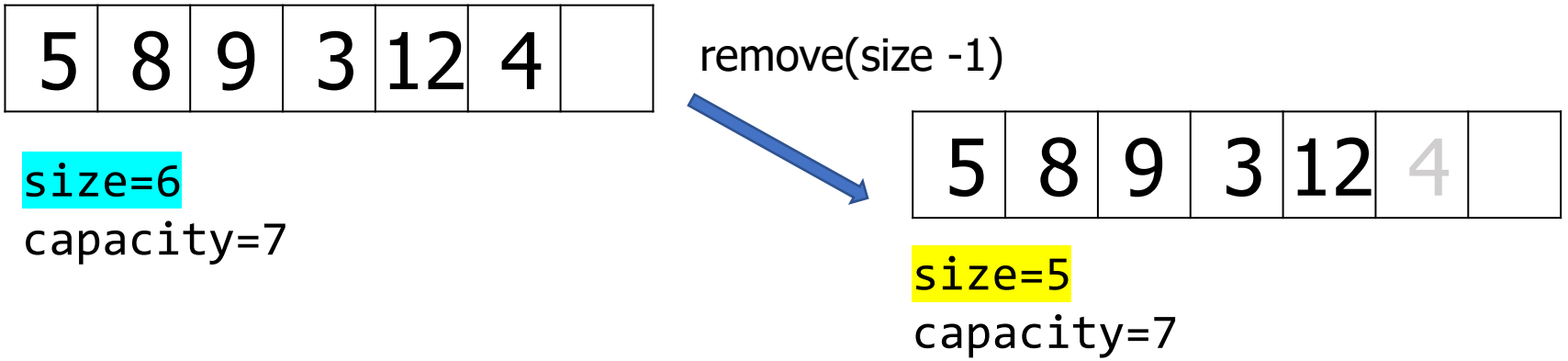
Shifting to the right

```
for (int i=size; i>j; i--)
      A[i] = A[i-1]
```

A[j]=9
size=6
capacity=7

Remove from the end

- Simply decrement `size`



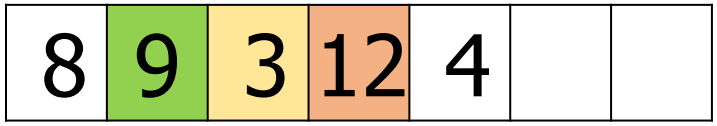
Remove in the middle (beginning)

- To remove element at position *j*, shift all elements from *j*+1 to *size*-1 to the left and decrement *size*

`size=6`
`capacity=7`

check that *j* is a valid position: *j*<*size*

remove(j=0)

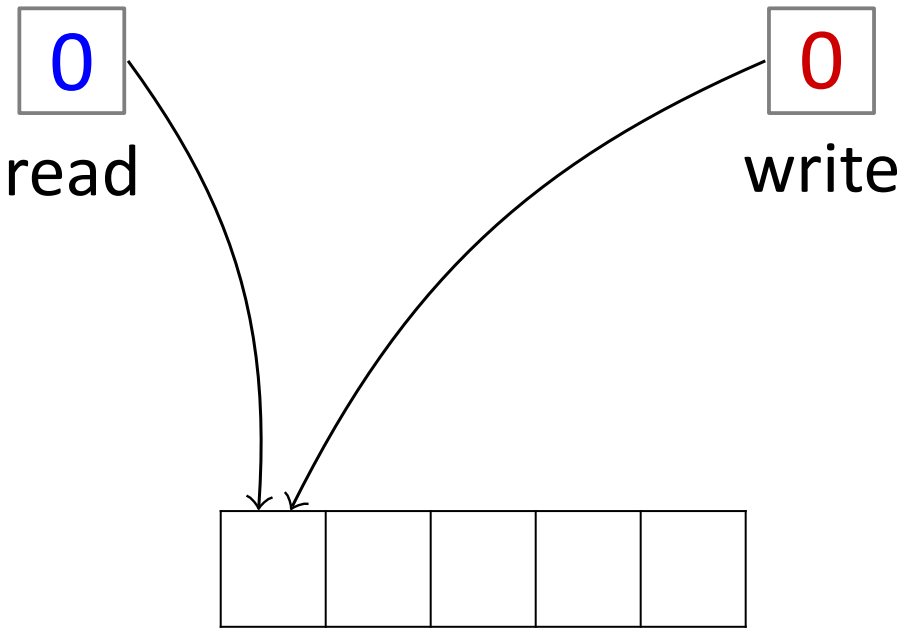


`size=5`
`capacity=7`

Shifting to the left

```
for (int i=j+1; i<size; i++)
       A[i-1] = A[i]
```

Queue ADT Implementation with Array



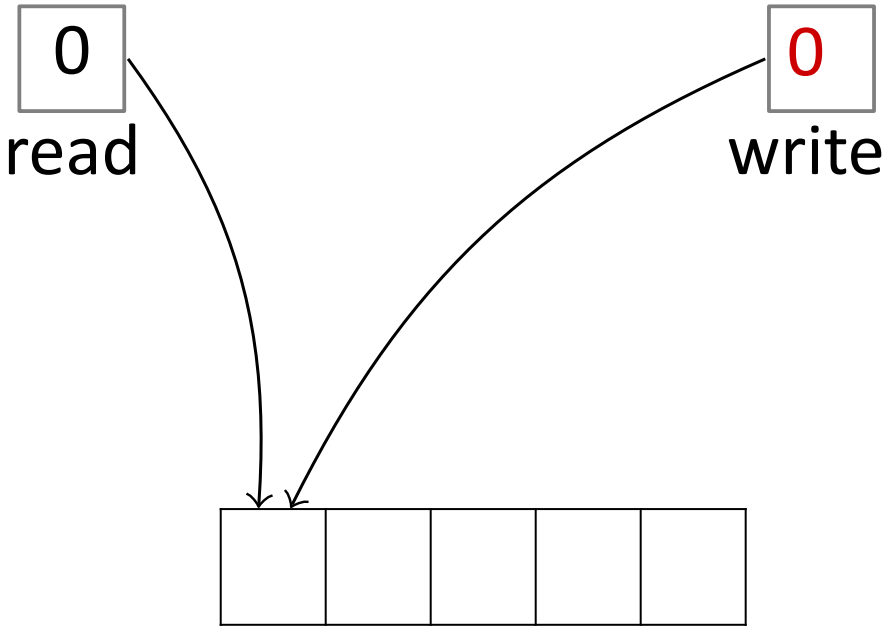
We have two pointers: *read* and *write*

read – index in A where the front element of the queue is located

write – index in A where the rear of the queue is located

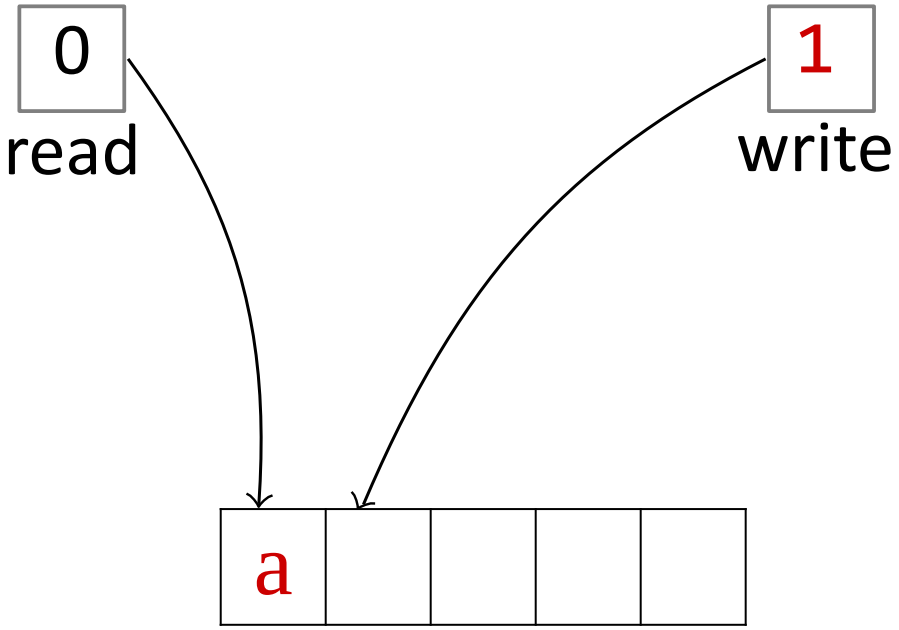
Initially, *read*=*write*=0

Queue Implementation with Array

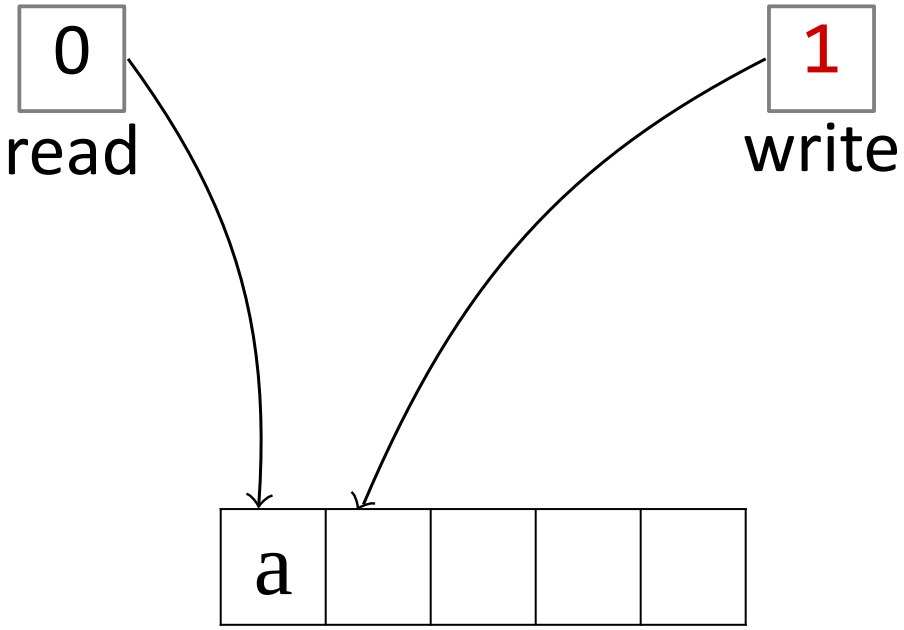


Enqueue(a)

Queue Implementation with Array

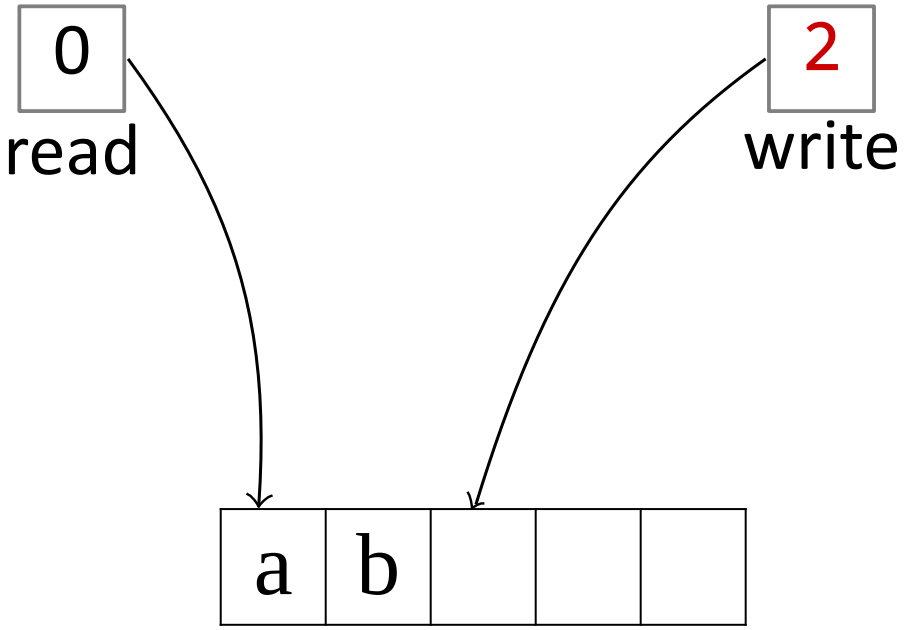


Queue Implementation with Array

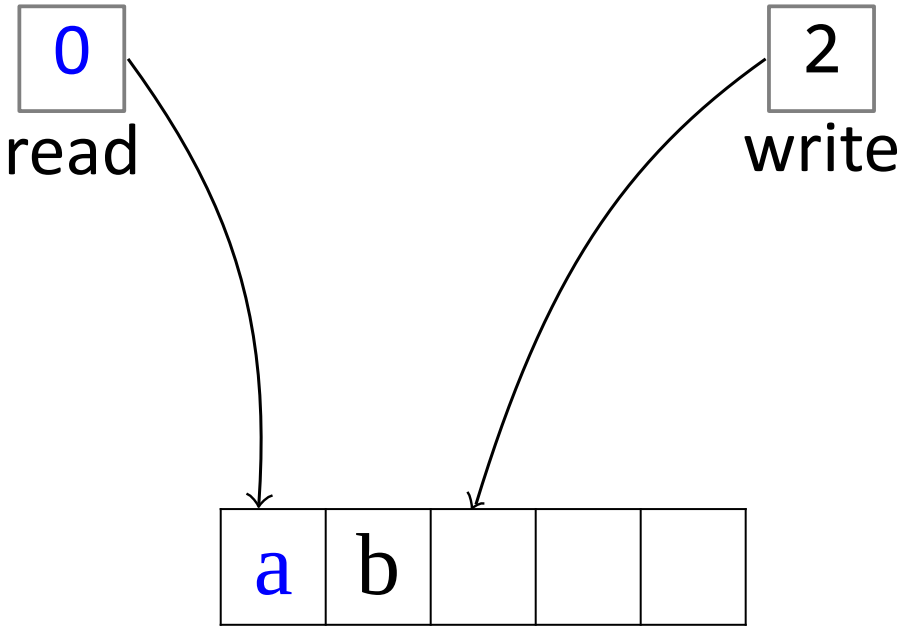


Enqueue(b)

Queue Implementation with Array

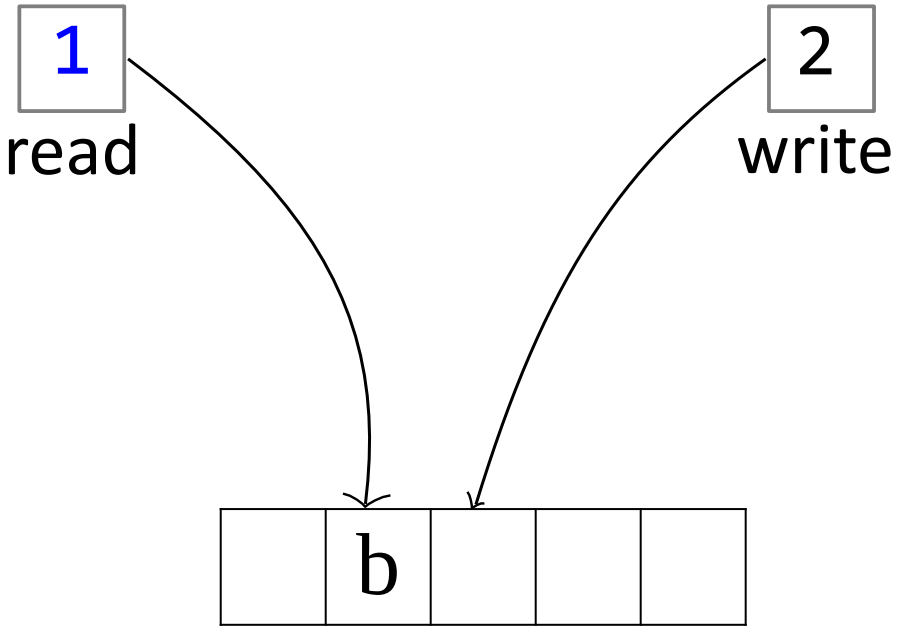


Queue Implementation with Array



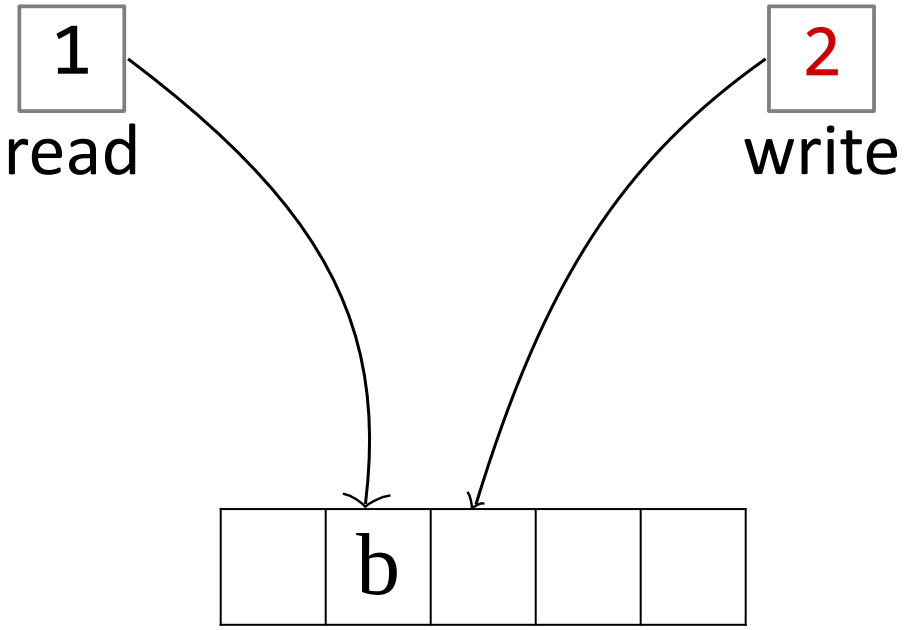
Dequeue()

Queue Implementation with Array



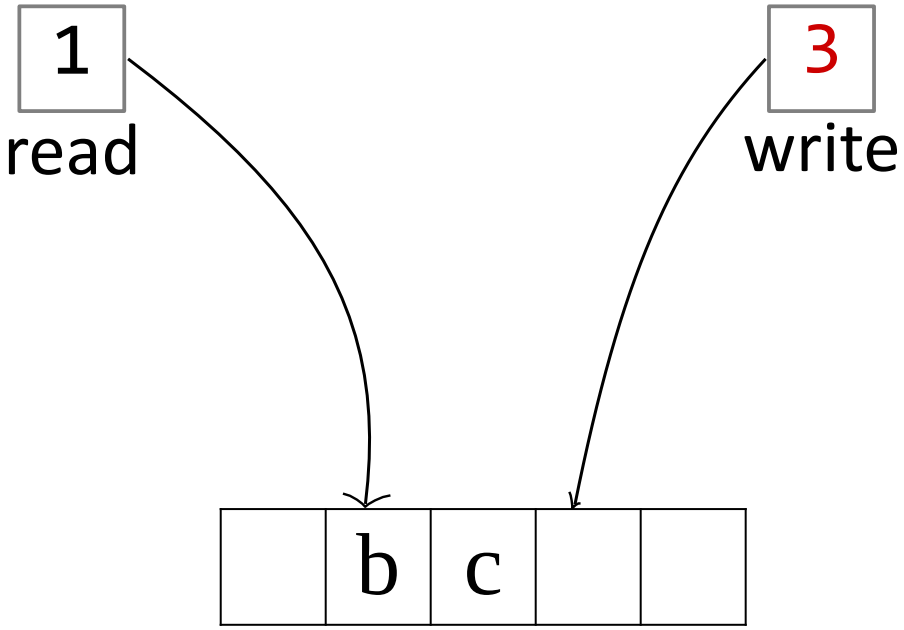
Dequeue() → a

Queue Implementation with Array

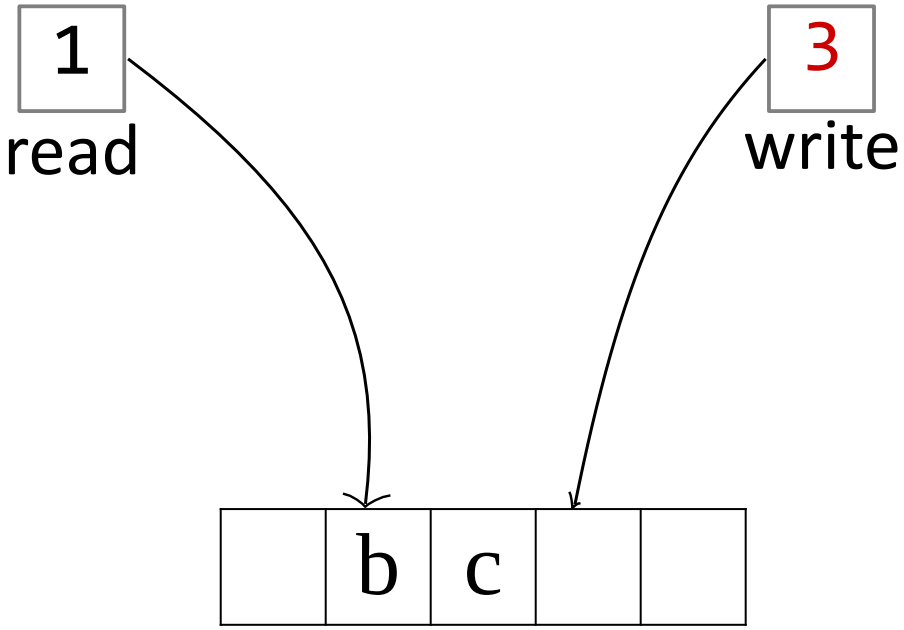


Enqueue(c)

Queue Implementation with Array

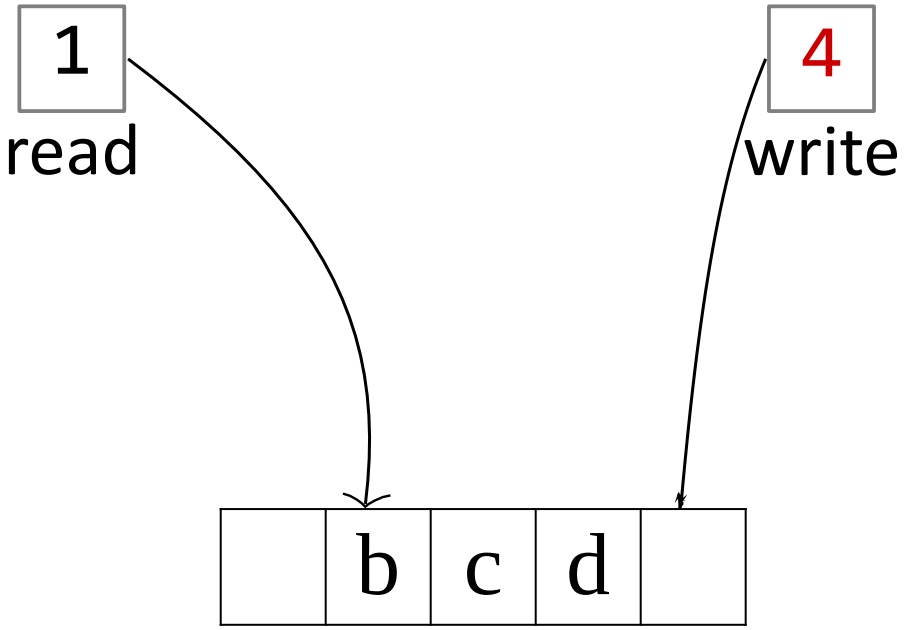


Queue Implementation with Array

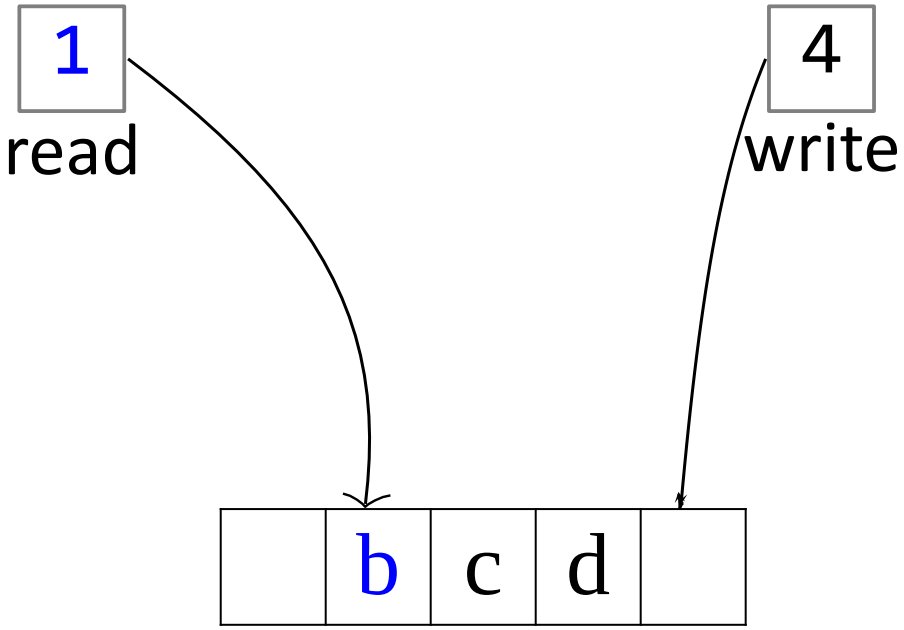


Enqueue(d)

Queue Implementation with Array

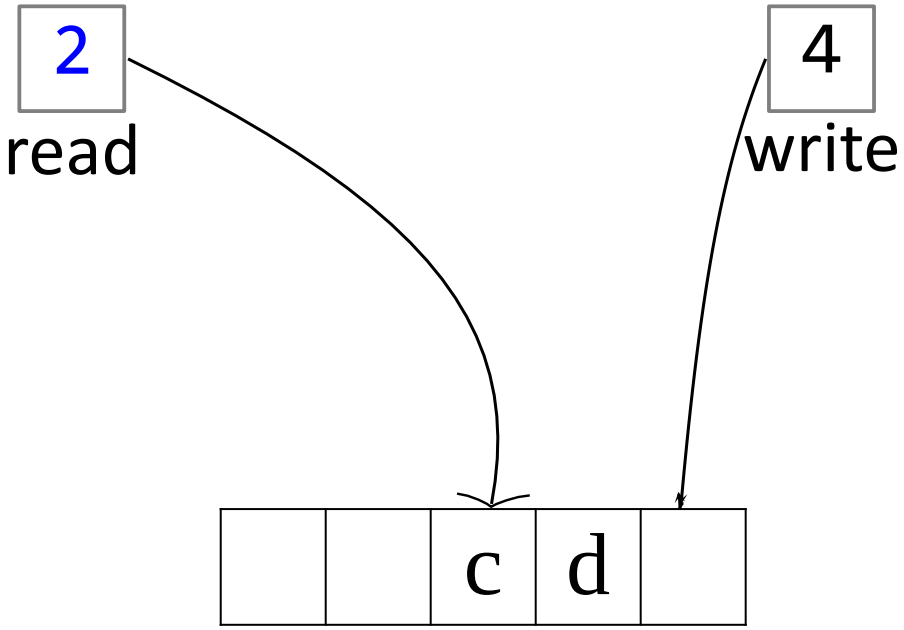


Queue Implementation with Array



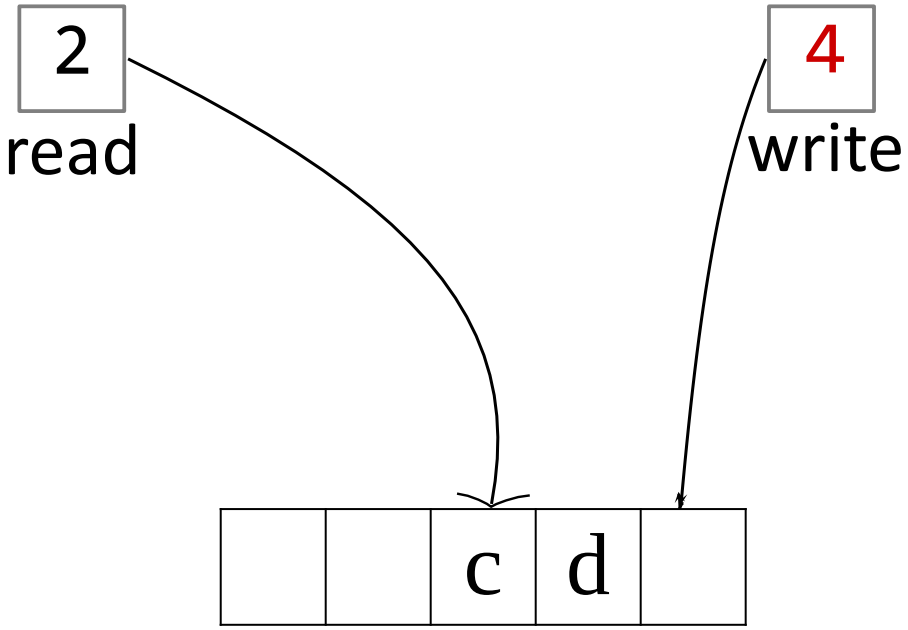
Dequeue()

Queue Implementation with Array



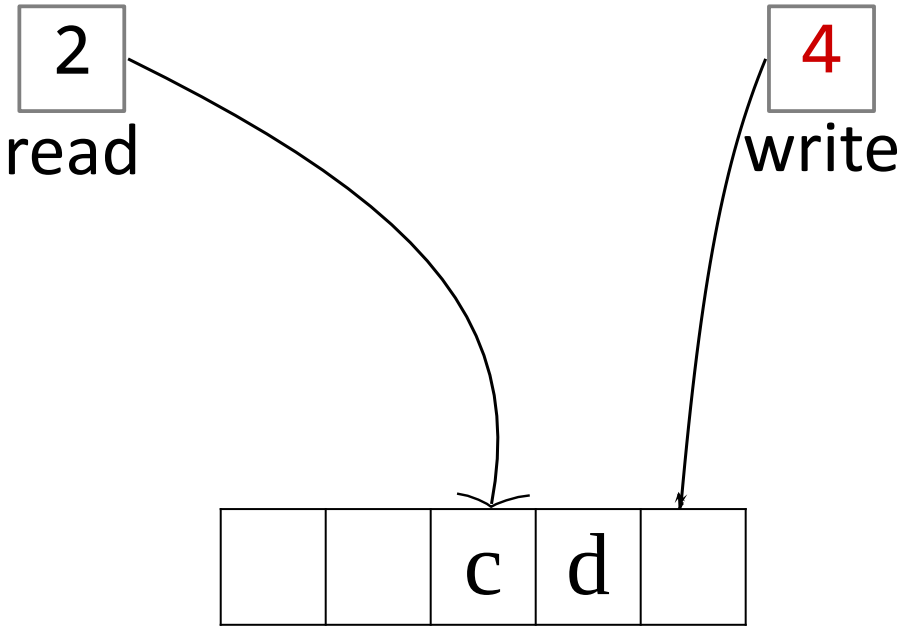
Dequeue() → *b*

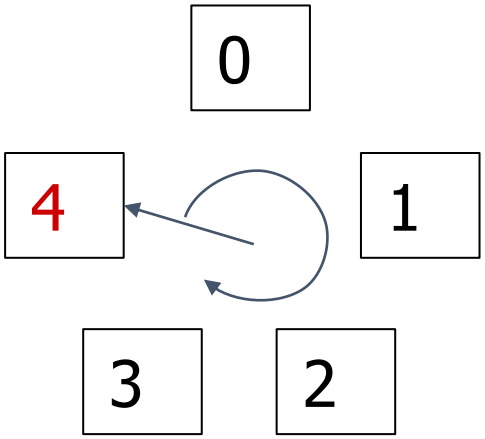
Queue Implementation with Array



Enqueue(e)

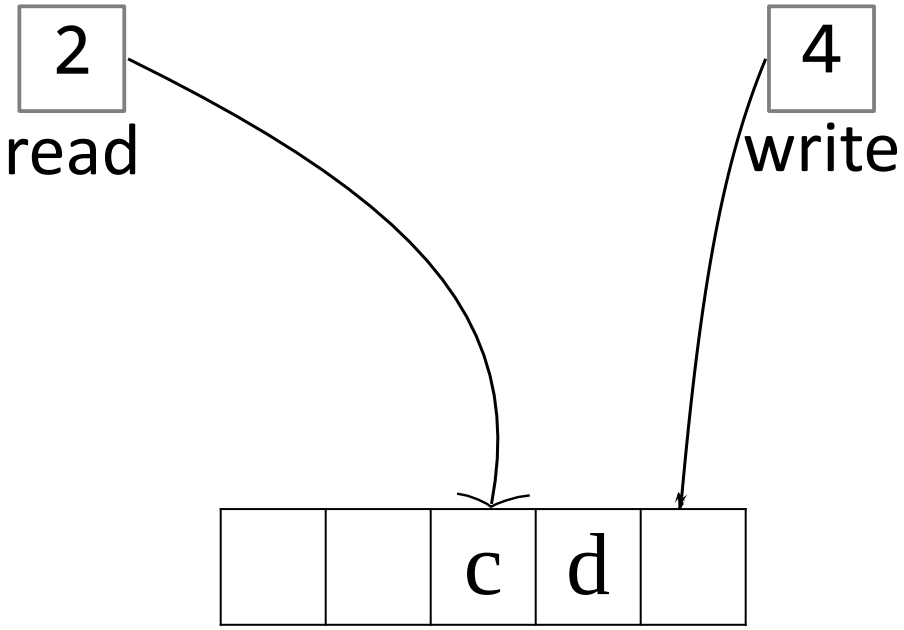
Concept of a Circular Array

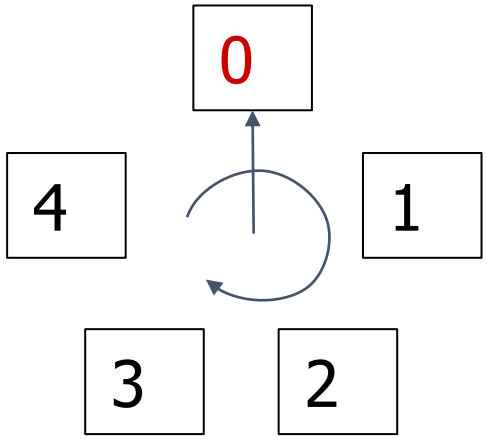




Enqueue(e)

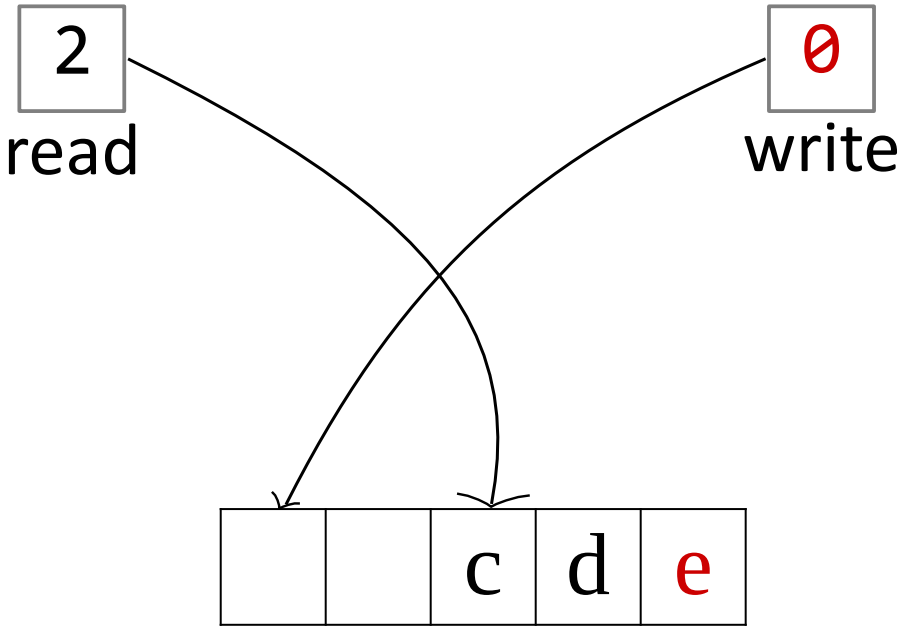
Concept of a Circular Array



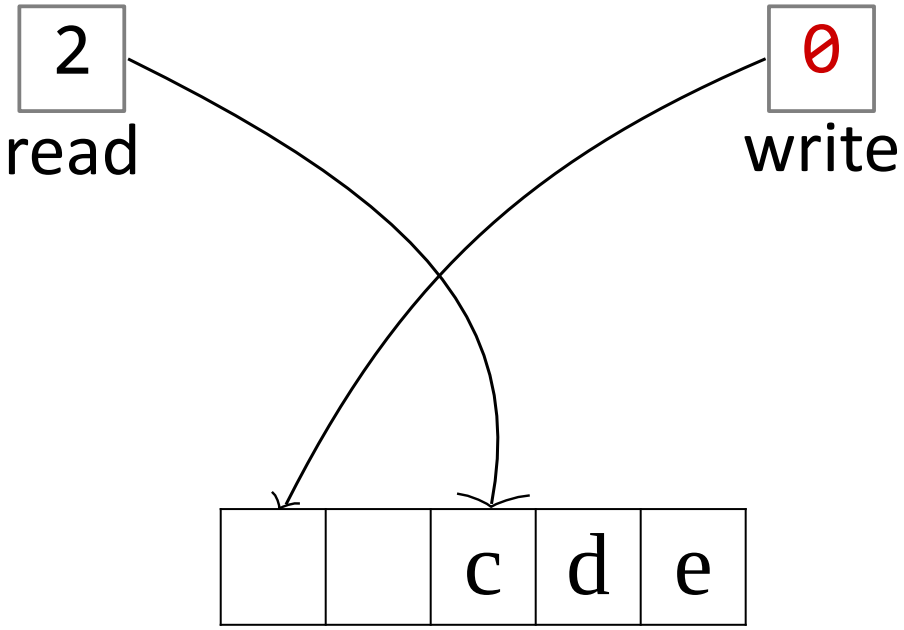


Enqueue(e)

Queue Implementation with a circular Array

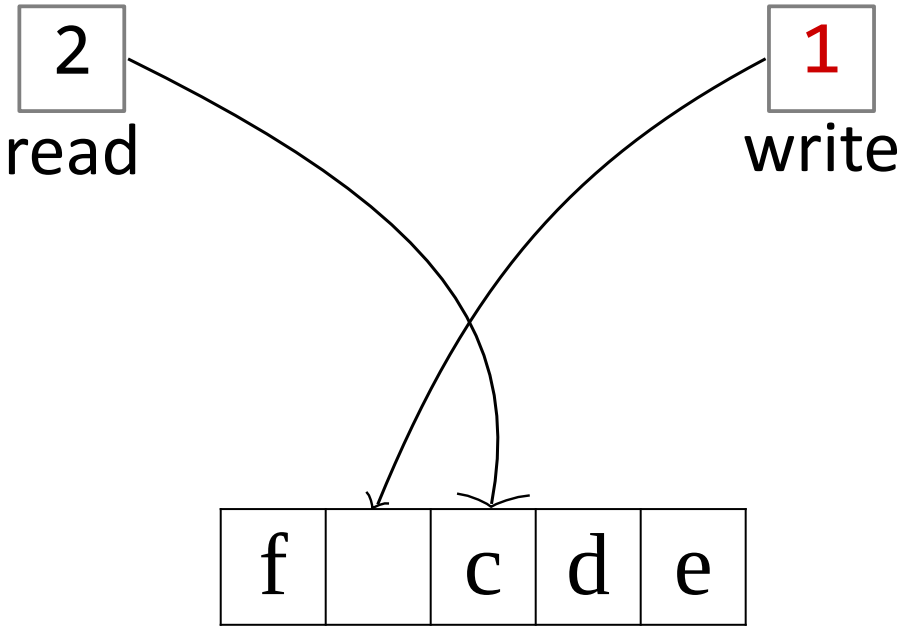


Queue Implementation with a circular Array

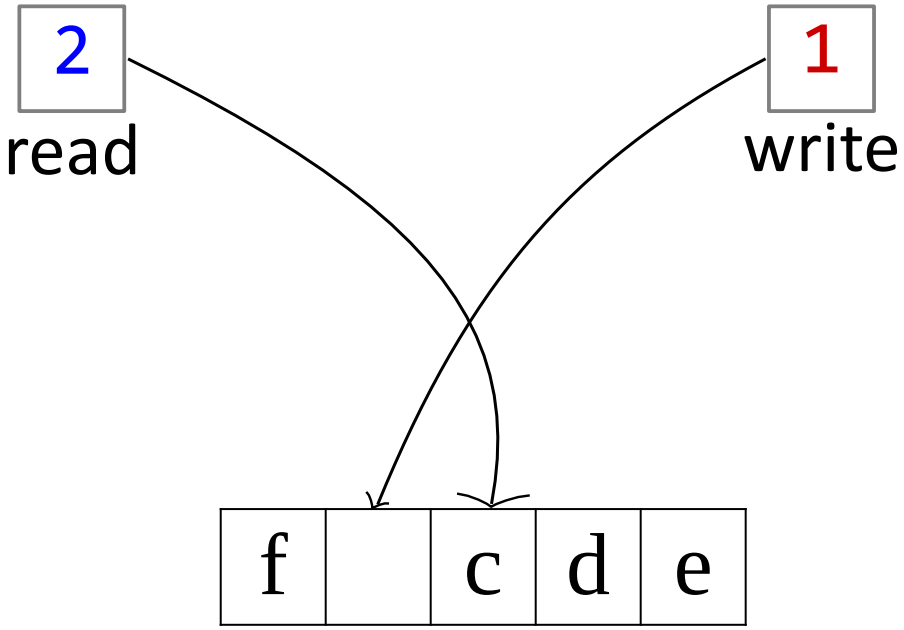


Enqueue(f)

Queue Implementation with a circular Array

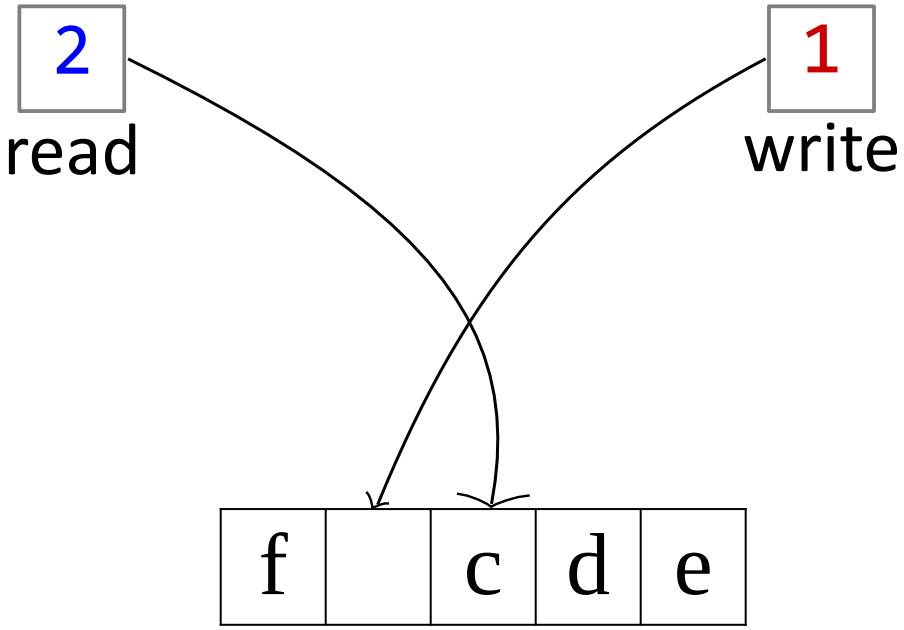


Queue Implementation with a circular Array



Enqueue(g)

Queue Implementation with a circular Array

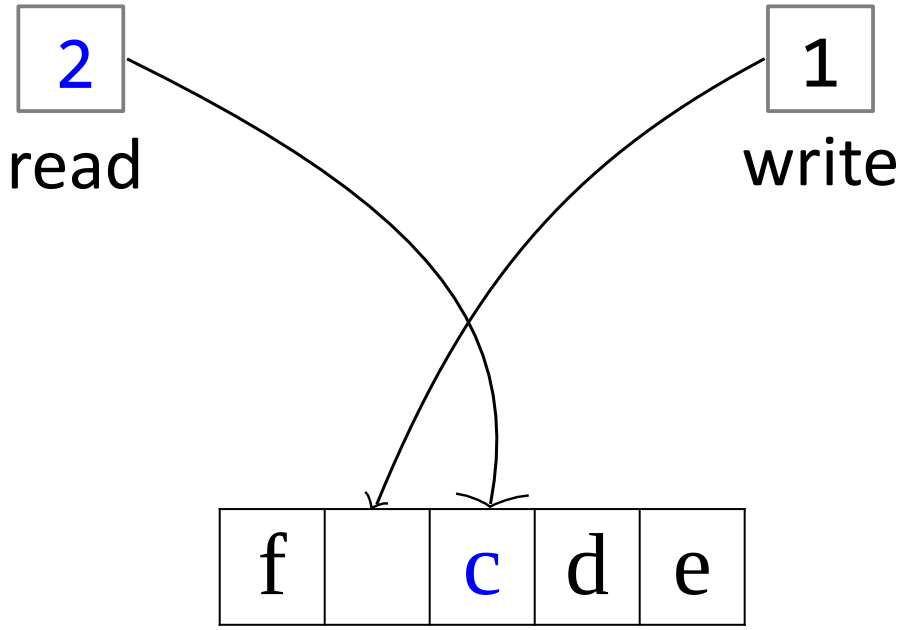


Enqueue(g) → ERROR

Cannot set read = write

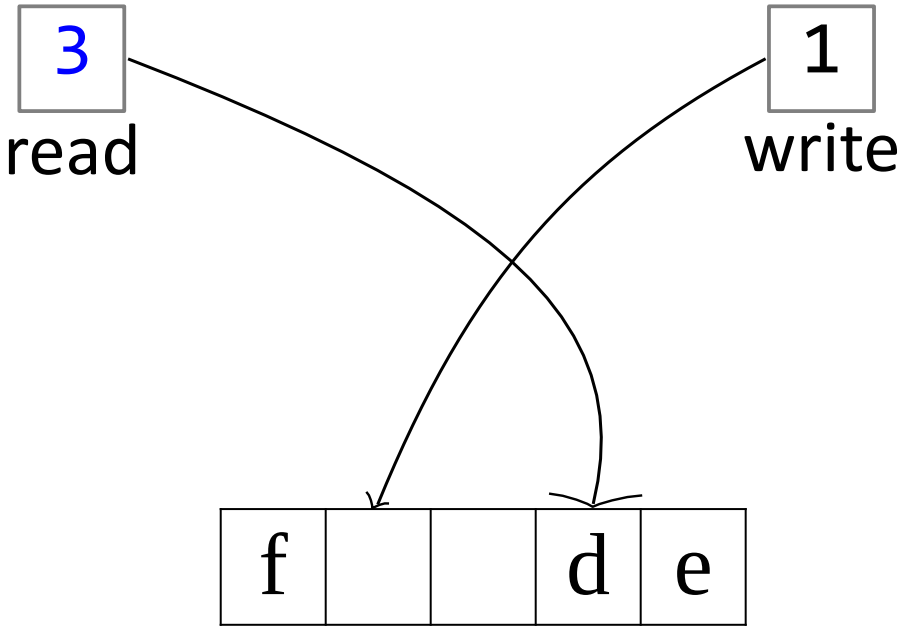
isFull() → True

Queue Implementation with a circular Array



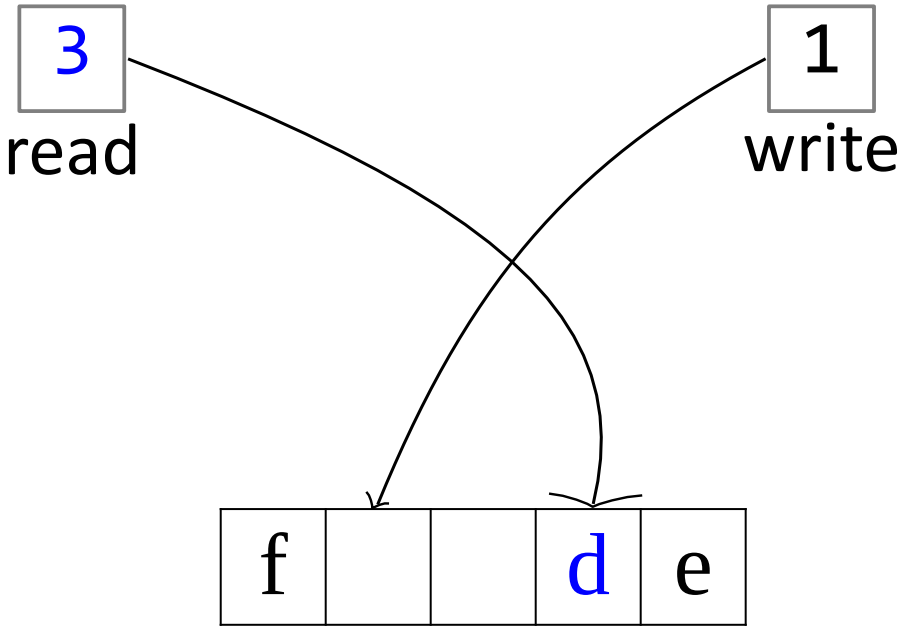
Dequeue()

Queue Implementation with a circular Array



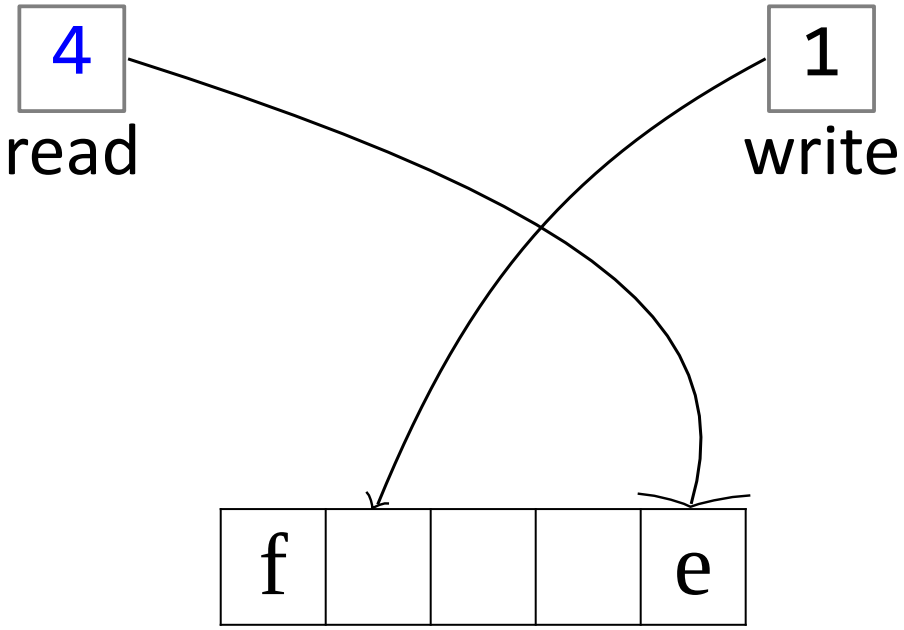
Dequeue() → *c*

Queue Implementation with a circular Array



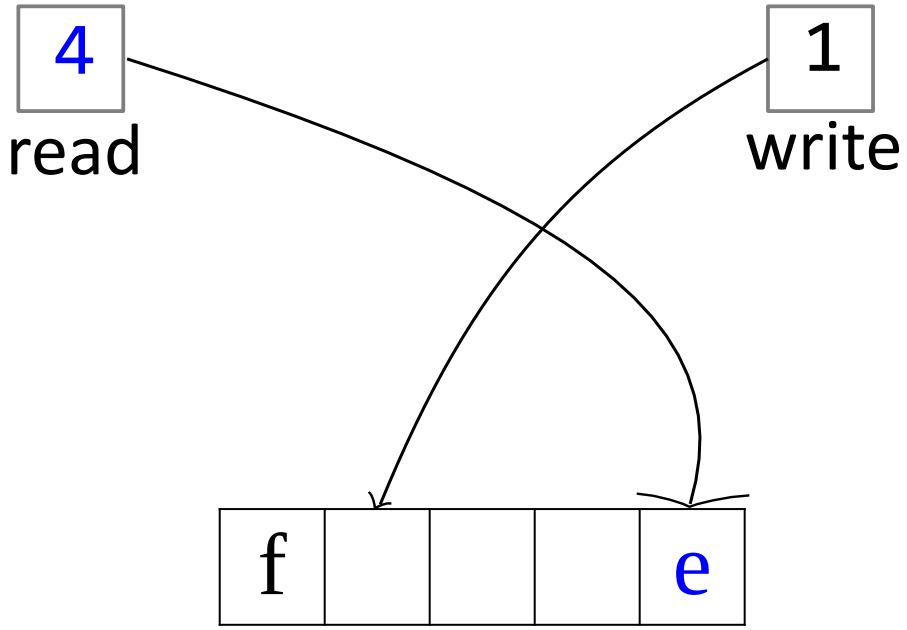
Dequeue()

Queue Implementation with a circular Array



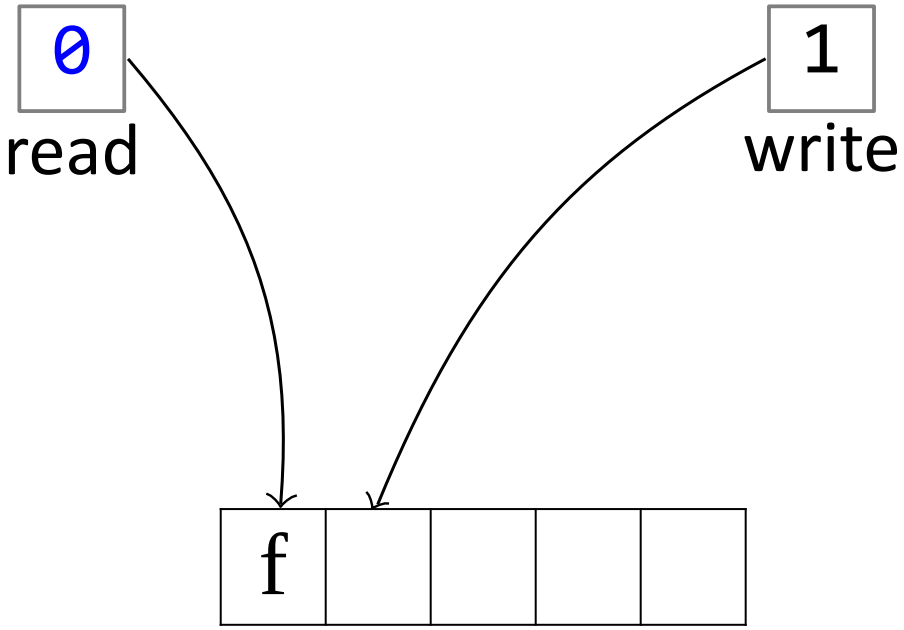
Dequeue() → *d*

Queue Implementation with a circular Array



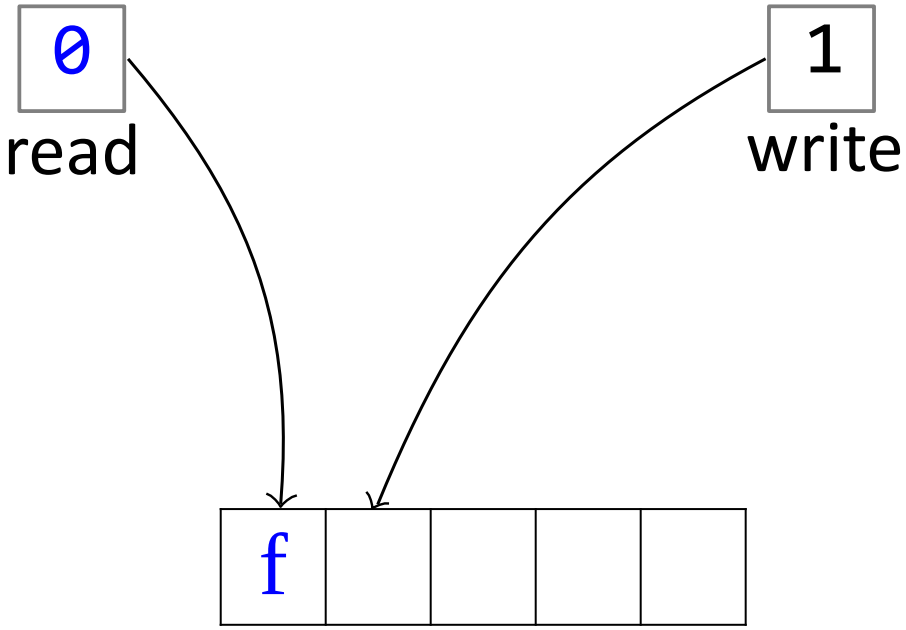
Dequeue()

Queue Implementation with a circular Array



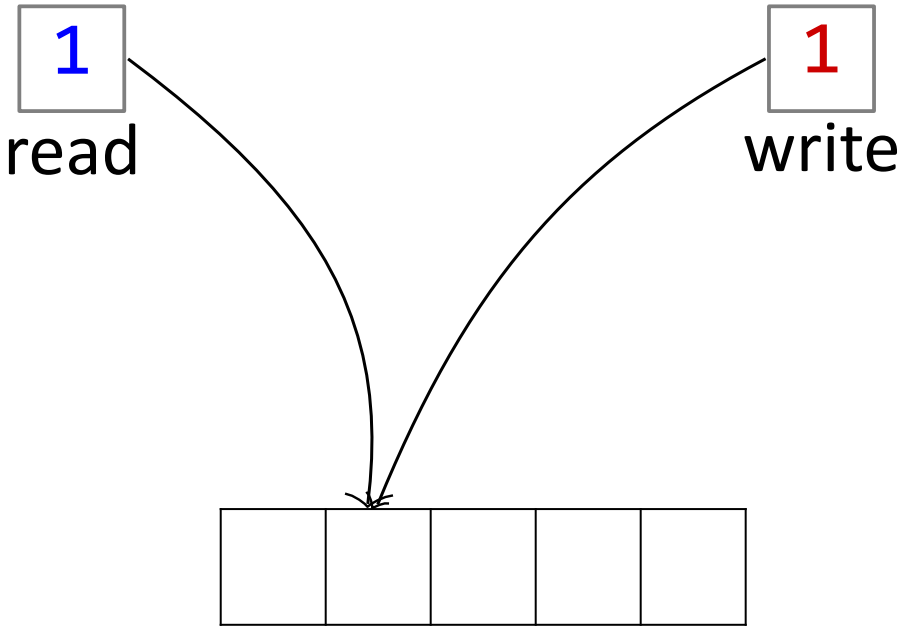
Dequeue() → *e*

Queue Implementation with a circular Array



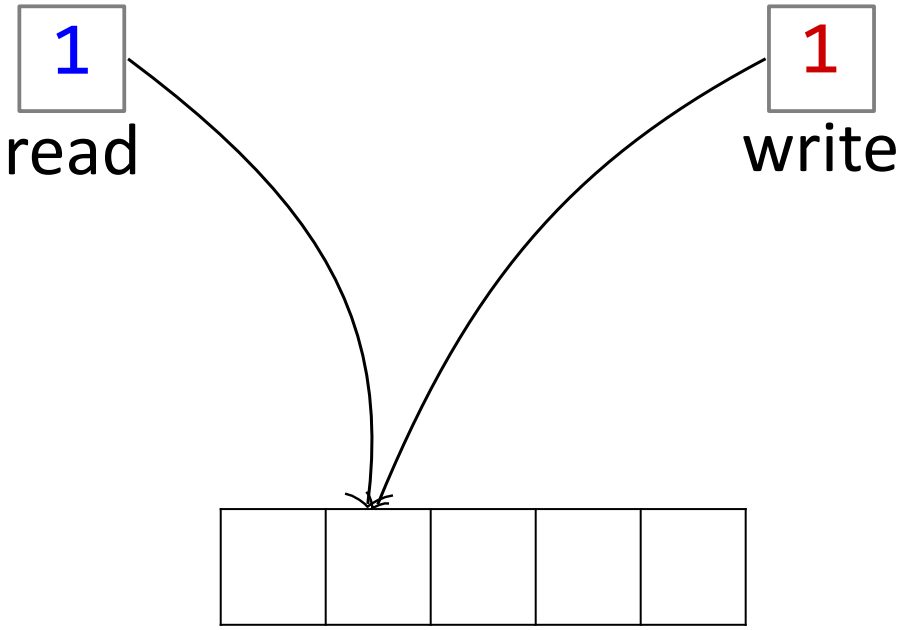
Dequeue()

Queue Implementation with a circular Array



Dequeue() → *f*

Queue Implementation with a circular Array



IsEmpty() → True

Queue Implementation using Array

- ➔ *Queue* ADT can be implemented with a *circular* Array
- ➔ We need 2 pointers (indexes of the array): *read* and *write*
- ➔ When we *enqueue(e)* we add *e* at position *write*, and increment *write*. If *write* was at the last position, it wraps around to position 0
- ➔ After *enqueue(e)* ***read*** **and** ***write*** **cannot be equal** - because next time you write you would erase the first element of the queue pointed to by *read*
- ➔ When we *dequeue()* we remove the element at position *read*, and increment *read*
- ➔ If *read*=*write* then the queue is empty

We hide implementation details from users of ADT

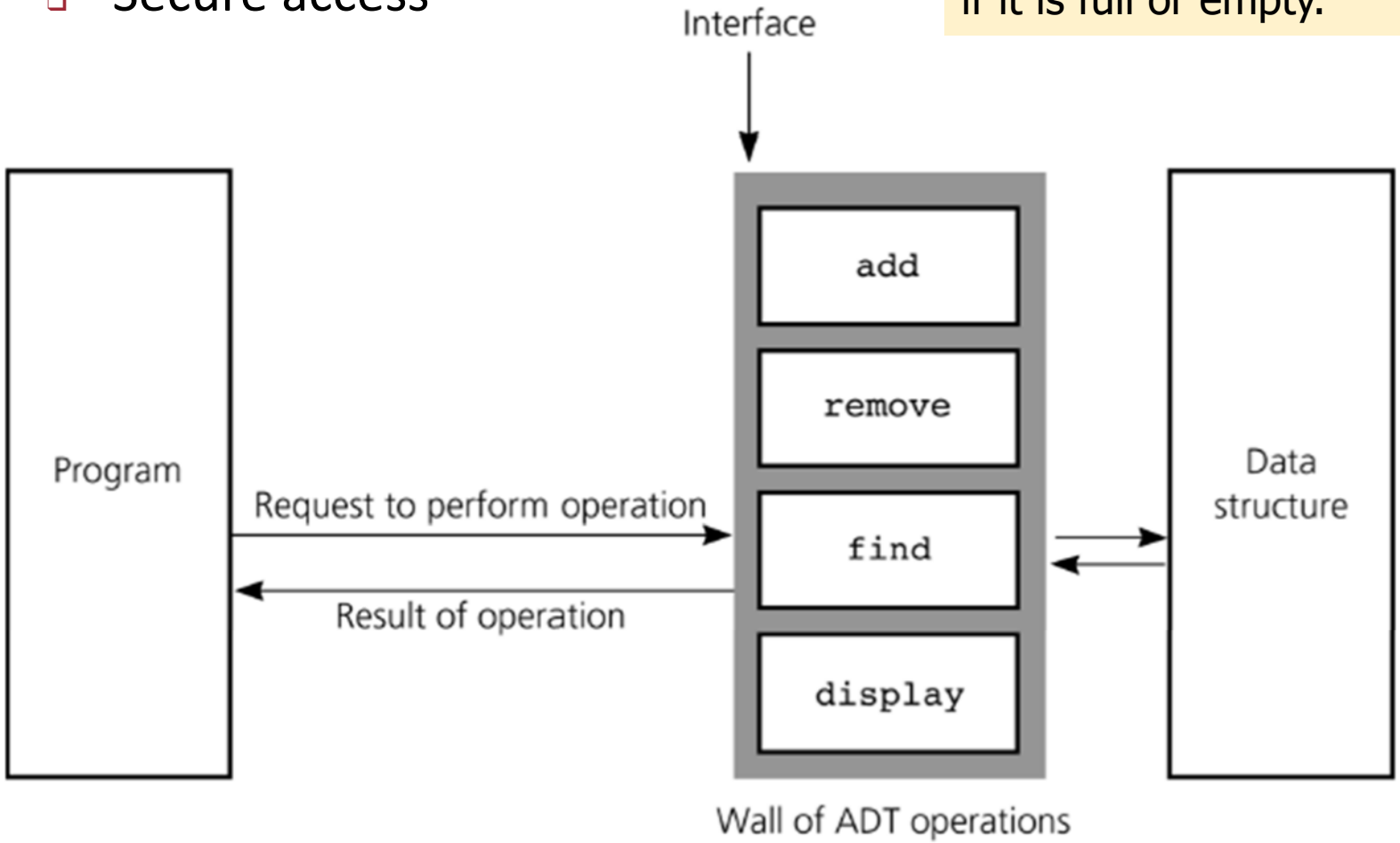
Users of ADT:

- Aware of the **specification only**
  - Usage only based on the specified operations
- Do not care / need **not** know about the actual **implementation**
  - i.e. Different implementations do **not** affect the users of ADT

A Wall of ADT

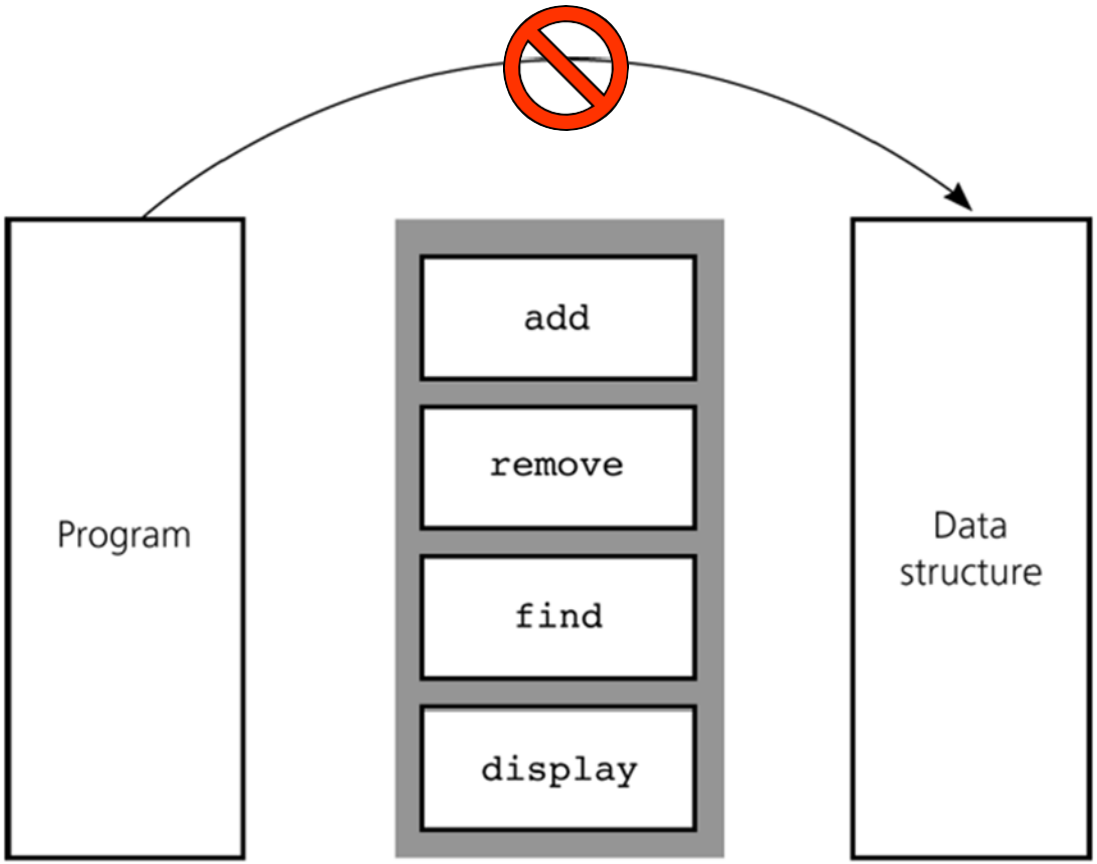
- ADT operations provide:
  - Interface to data structures
  - Secure access

Ex. The users of our Queue implementation do not know that there is an array, or what is a capacity of this array: only if it is full or empty.



Violating the abstraction

- User programs **should not**:
  - Use the underlying data structure directly
  - Depend on implementation details



Advantages of ADT

- Hide the implementation details by building walls around the data and operations
  - So that **changes in either will not affect** other program components that use them
- Functionalities are less likely to change
- Localize rather than globalize changes
- This helpx manage software complexity

Different Queue Implementations

- Queue ADT can be implemented using a **regular Array**, where the front is always at position 0, and rear at position size
  - This queue implementation requires shifting all the data to the left when we dequeue
- Queue ADT can be implemented using a **circular Array** – as demonstrated above
- Later in the course we will learn how to implement Queue ADT using **Linked List**

Comparing Different Queue Implementations

- Which implementation is the best?
  - It seems that we need some tools to analyze and compare the performance of different implementations

- This is the part of the course where we enter the Algorithms and study the tools for analyzing them