Stack and Queue ADT

Lecture 16

by Marina Barsky

Recap: Abstract Data Types (ADT)

ADT includes:

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

The difference between **specification** and **implementation** can be best explained on the example of *Stack* and *Queue* ADTs

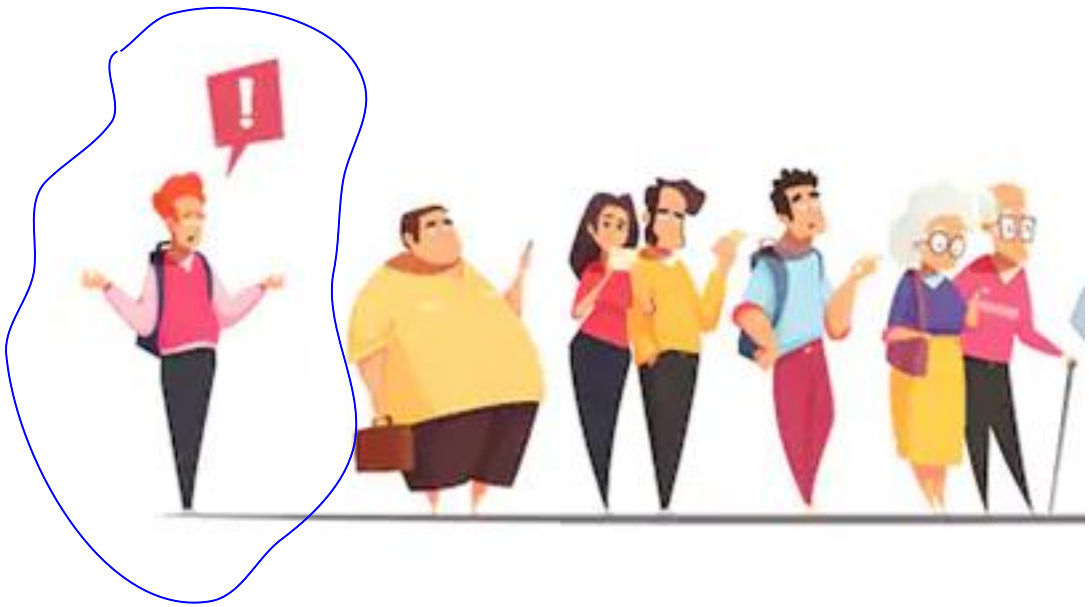
Example 1:
Abstraction for HR roster

We want to model the maintenance of the list of company employees

- ➢ When the company grows - we should be able to add a new employee

Example 1: HR roster

- When the company grows - we should be able to **add** a new employee



Example 1: HR roster

- When the company grows - we should be able to add a new employee
- When the company downsizes we should be able to **remove** the last-added employee (seniority principle)

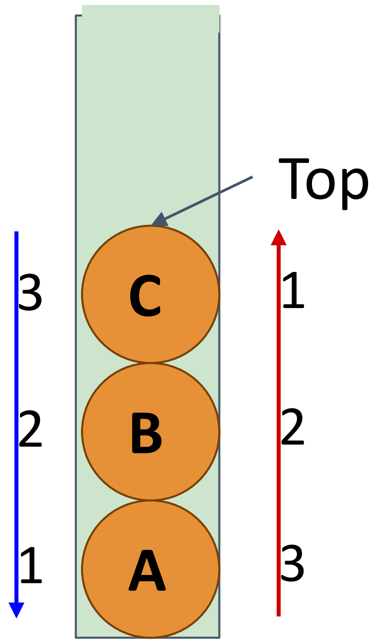
Example 1: HR roster

Requirements:

- ➢ When the company grows - we should be able to add a new employee
- ➢ When the company downsizes we should be able to remove the last-added employee (seniority principle)

Abstraction of HR roster: *Stack*

- If these are the only important requirements to the HR roster, then we can model it using ***Stack*** Abstract Data Type
- Stack stores a sequence of elements and allows only 2 operations: **adding a new element on top** of the stack and **removing the element from the top** of the stack
- Thus, the elements are sorted by the time stamp - from recent to older
- Stack is also called a **LIFO** queue (**L**ast **I**n - **F**irst **O**ut)



Specification

Stack: Abstract data type which stores dynamic sequence and supports following operations:

➔*Push(e)*: adds element to collection

➔*Peek() [Top()]*: returns most recently-added element

➔*Pop()*: removes and returns most recently-added element

➔Boolean *IsEmpty()*: are there any elements?

➔Boolean *IsFull()*: is there any space left?

ADT: Specification vs. implementation

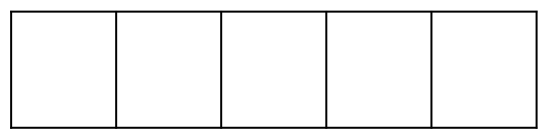
Specification and **implementation** have to be disjoint:

- **One** specification
- **One or more** implementations
  - **Using different data structures (Array? Linked List?)**
  - **Using different algorithms**

Stack Implementation with Array

size: 0

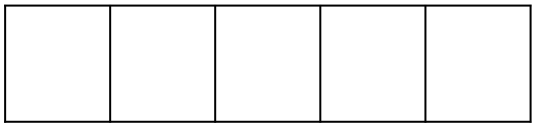
capacity: 5



Stack Implementation with Array

size: 0

capacity: 5



Push(a)

Stack Implementation with Array

size: 1

capacity: 5

a				

Stack Implementation with Array

size: 1

capacity: 5

a				

Push(b)

Stack Implementation with Array

size: 2

capacity: 5

a	b			

Stack Implementation with Array

size: 2

capacity: 5

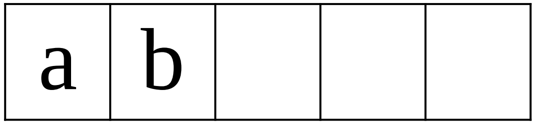
a	b			

Peek() → *b*

Stack Implementation with Array

size: 2

capacity: 5



Push(c)

Stack Implementation with Array

size: 3

capacity: 5

a	b	c		

Stack Implementation with Array

size: 3

capacity: 5

a	b	c		

Pop( )

Stack Implementation with Array

size: 2

capacity: 5

a	b			

Pop() → *c*

Stack Implementation with Array

size: 2

capacity: 5

a	b			

Push(d)

Stack Implementation with Array

size: 3

capacity: 5

a	b	d		

Stack Implementation with Array

size: 3

capacity: 5

a	b	d		

Push(e)

Stack Implementation with Array

size: 4

capacity: 5

a	b	d	e	

Stack Implementation with Array

size: 4

capacity: 5

a	b	d	e	

Push(f)

Stack Implementation with Array

size: 5

capacity: 5

a	b	d	e	f

Stack Implementation with Array

size: 5

capacity: 5

a	b	d	e	f

Push(g)

Stack Implementation with Array

size: 5

capacity: 5

a	b	d	e	f

ERROR

isFull() → True

Stack Implementation with Array

size: 5

capacity: 5

a	b	d	e	f

Pop( )

Stack Implementation with Array

size: 4

capacity: 5

a	b	d	e	

IsEmpty → *False*

Stack Implementation with Array

size: 4

capacity: 5

a	b	d	e	

Pop( )

Stack Implementation with Array

size: 3

capacity: 5

a	b	d		

Pop( )

Stack Implementation with Array

size: 2

capacity: 5

a	b			

Stack Implementation with Array

size: 2

capacity: 5

a	b			

Pop( )

Stack Implementation with Array

size: 1

capacity: 5

a				

Stack Implementation with Array

size: 1

capacity: 5

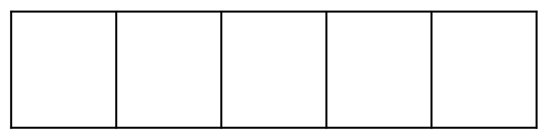
a				

Pop( )

Stack Implementation with Array

size: 0

capacity: 5

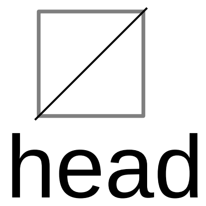


IsEmpty() → *True*

Stack ADT: cost of operations

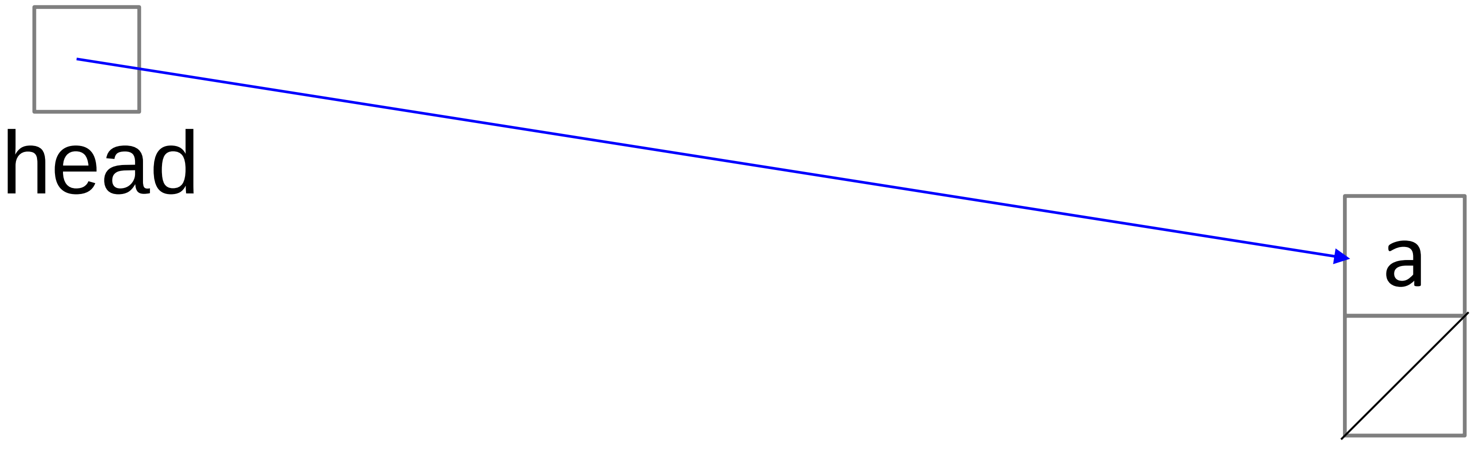
	Array Impl.	
Push(e)	O(1) if no resize is needed	
Peek()	O(1)	
Pop()	O(1)	
IsEmpty()	O(1)	
IsFull()	O(1)	

Stack Implementation with Linked List

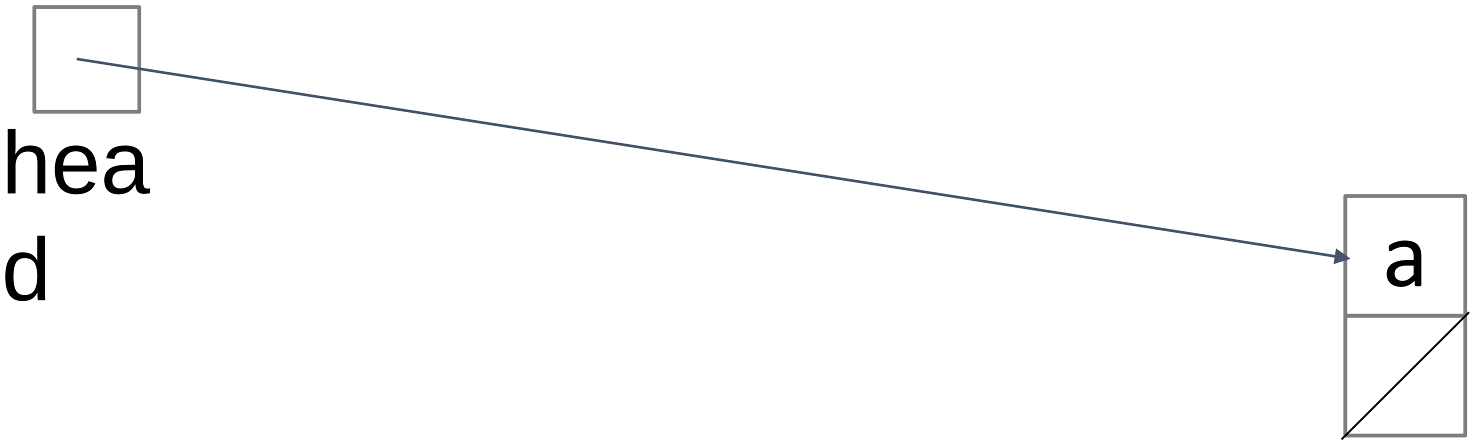


Push(a)

Stack Implementation with Linked List

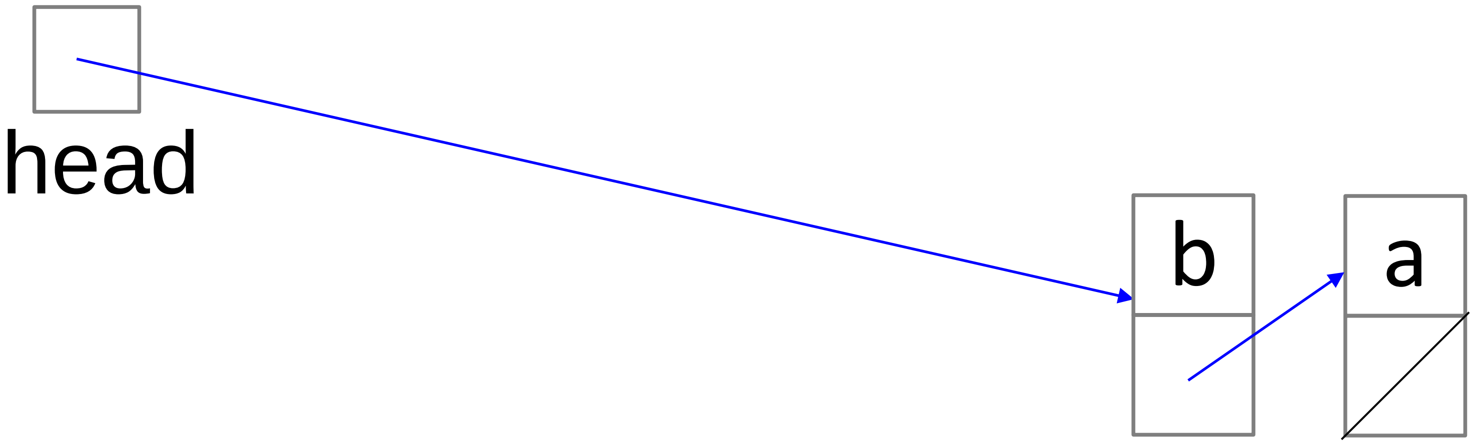


Stack Implementation with Linked List

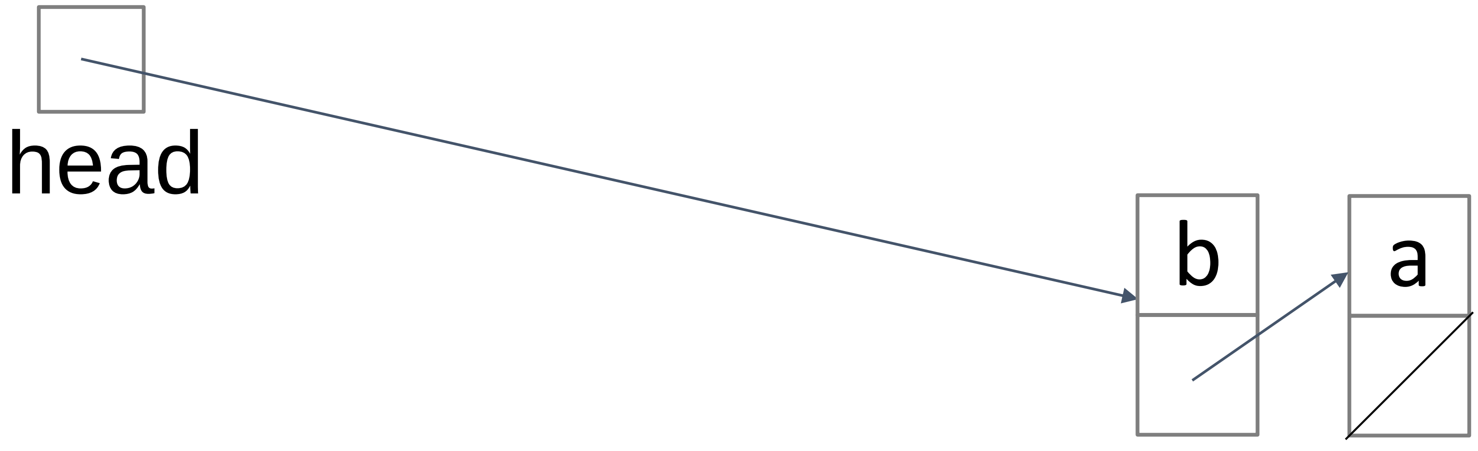


Push(b)

Stack Implementation with Linked List

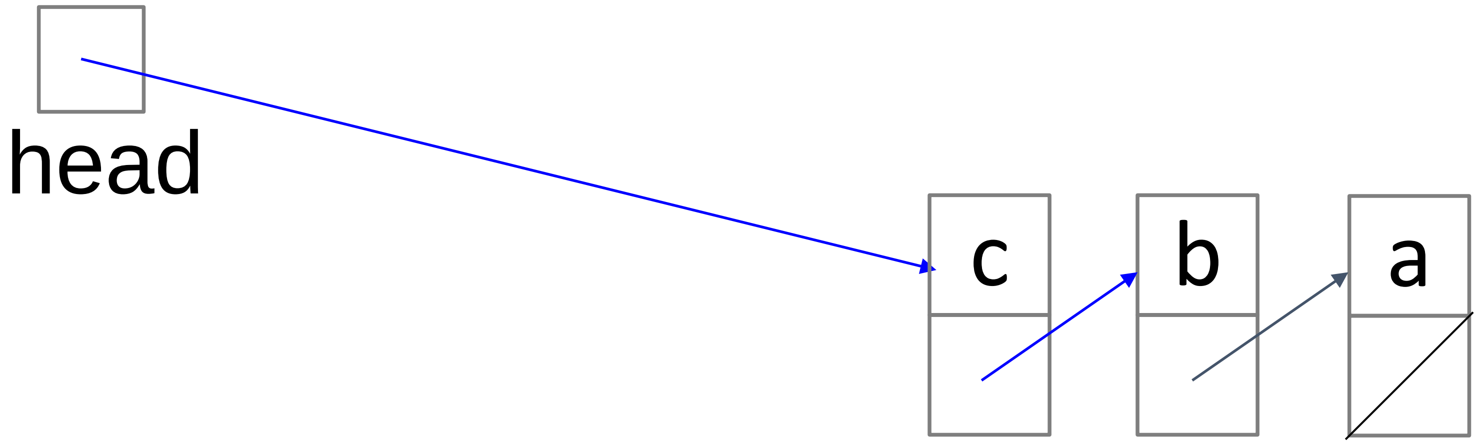


Stack Implementation with Linked List

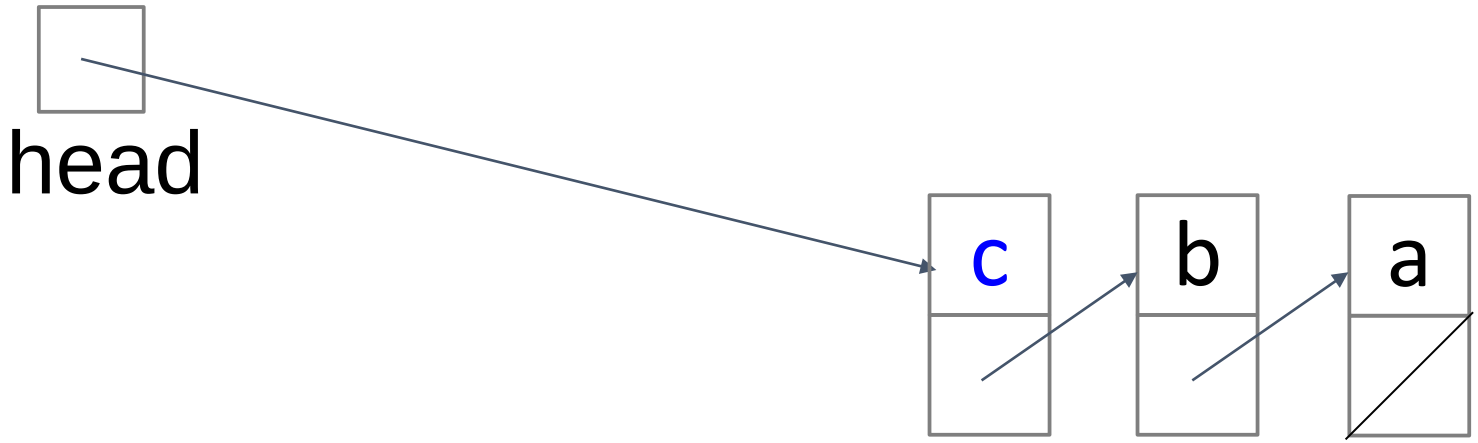


Push(c)

Stack Implementation with Linked List

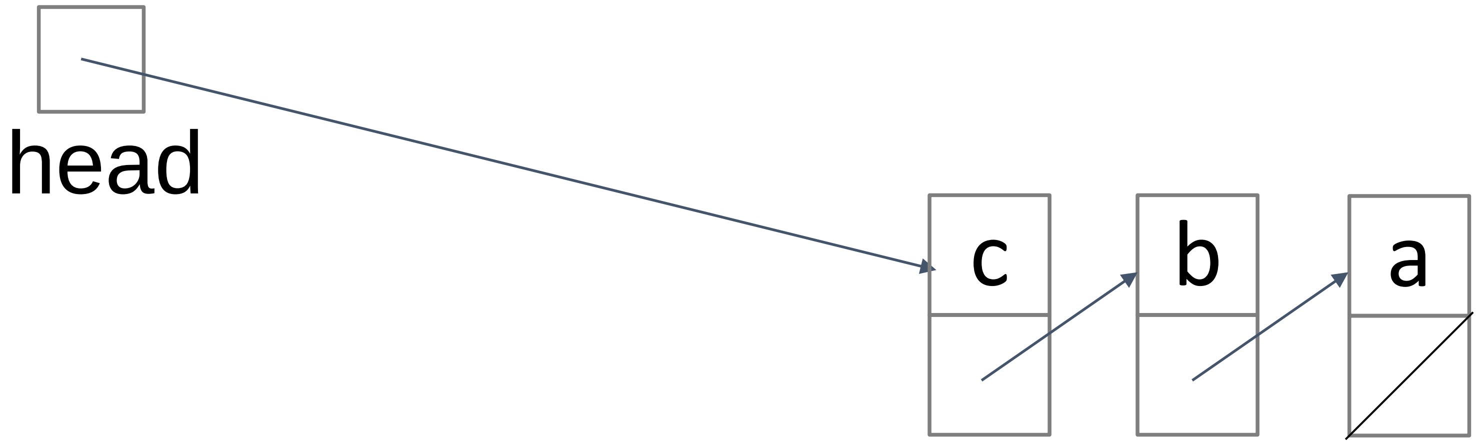


Stack Implementation with Linked List



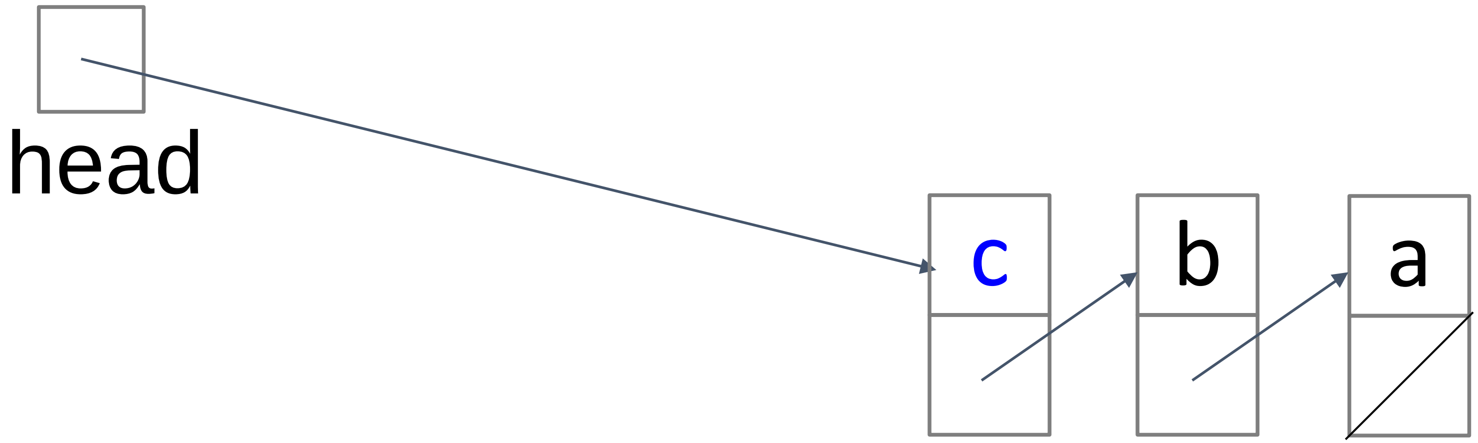
Peek()

Stack Implementation with Linked List



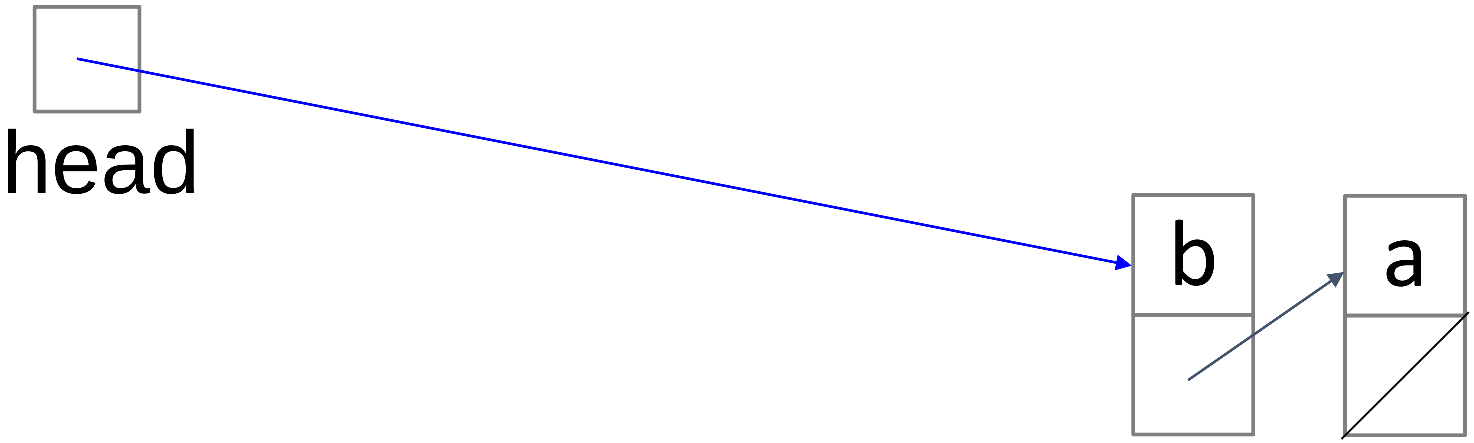
Peek() → *c*

Stack Implementation with Linked List



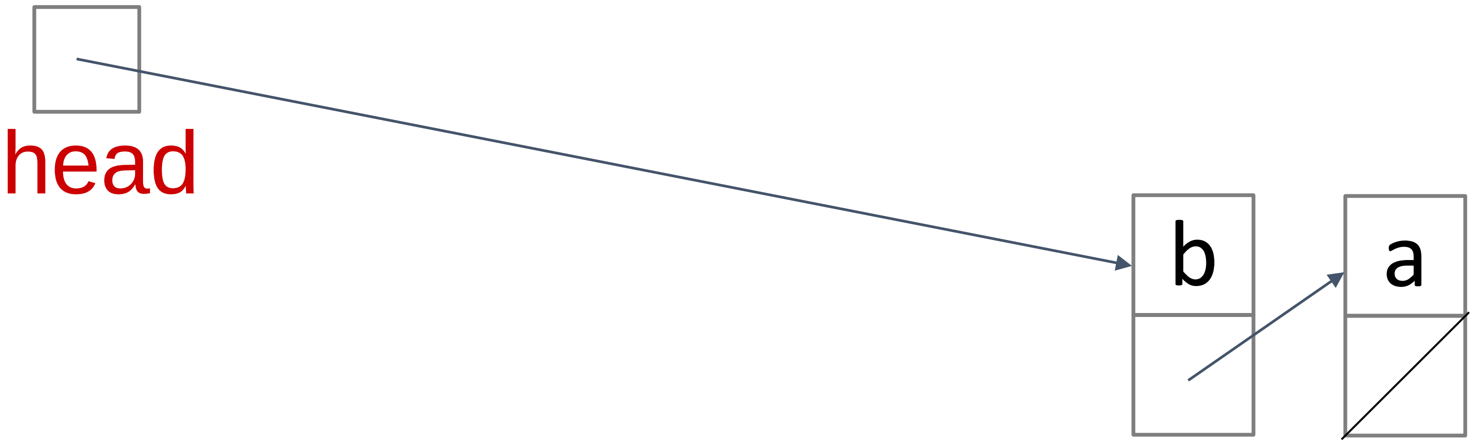
Pop()

Stack Implementation with Linked List



$Pop() \rightarrow c$

Stack Implementation with Linked List



IsEmpty() → False

Stack ADT: cost of operations

	Array Impl.	Link. List Impl.
Push(e)	O(1)	O(1)
Peek()	O(1)	O(1)
Pop()	O(1)	O(1)
IsEmpty()	O(1)	O(1)
IsFull()	O(1)	O(1)

Stack: Summary

- ➔ **ADT *Stack*** can be implemented with either an *Array* or a *Linked List* Data structure
- ➔ Each stack operation is $O(1)$: `Push, Pop, Peek, IsEmpty`
- ➔ Considerations:
  - ◆ Linked Lists have storage overhead
  - ◆ Arrays need to be resized when full

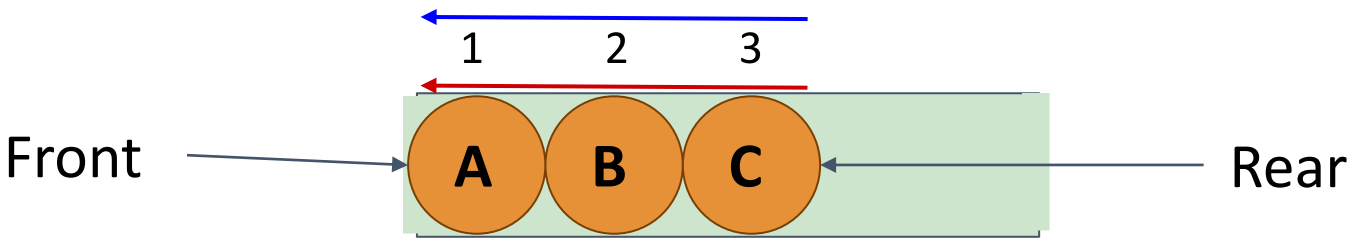
Example 2:
Abstraction for the Doctor Queue

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**
- As in the Stack ADT, the elements in the Queue are also sorted by timestamp, but in a different order: from the earlier to the later
- This ADT is called a *FIFO Queue* (First In First Out)



Specification

Queue: Abstract Data Type which stores dynamic data and supports the following operations:

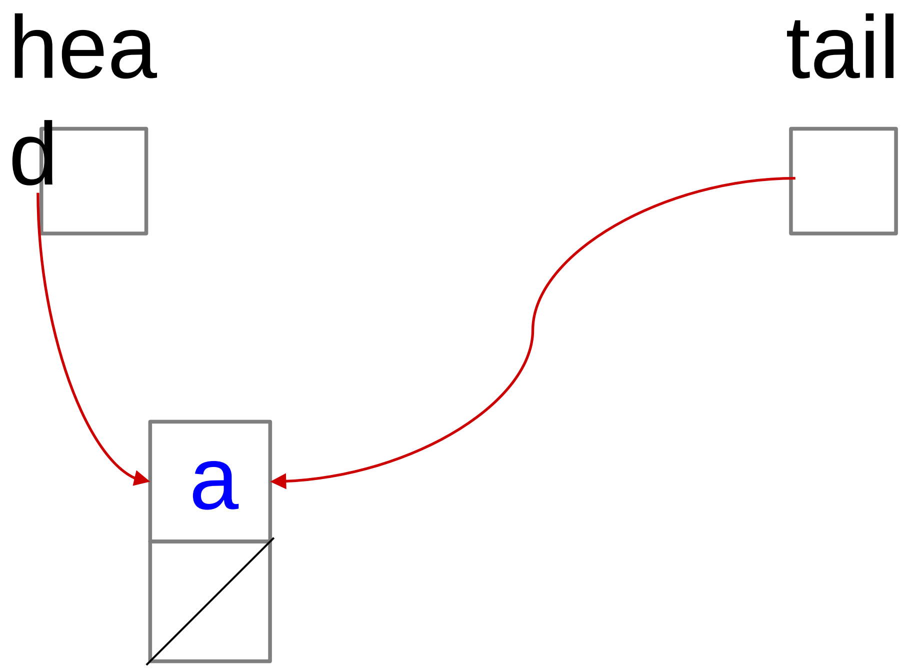
- ➔ ***Enqueue(e)***: adds element *e* to collection
- ➔ ***Peek()[Front()]***: returns least recently-added (the oldest) key
- ➔ ***Dequeue()***: removes and returns least recently-added key
- ➔ Boolean ***IsEmpty( )***: are there any elements?
- ➔ Boolean ***IsFull()***: is there any space left?

Queue Implementation with Linked List

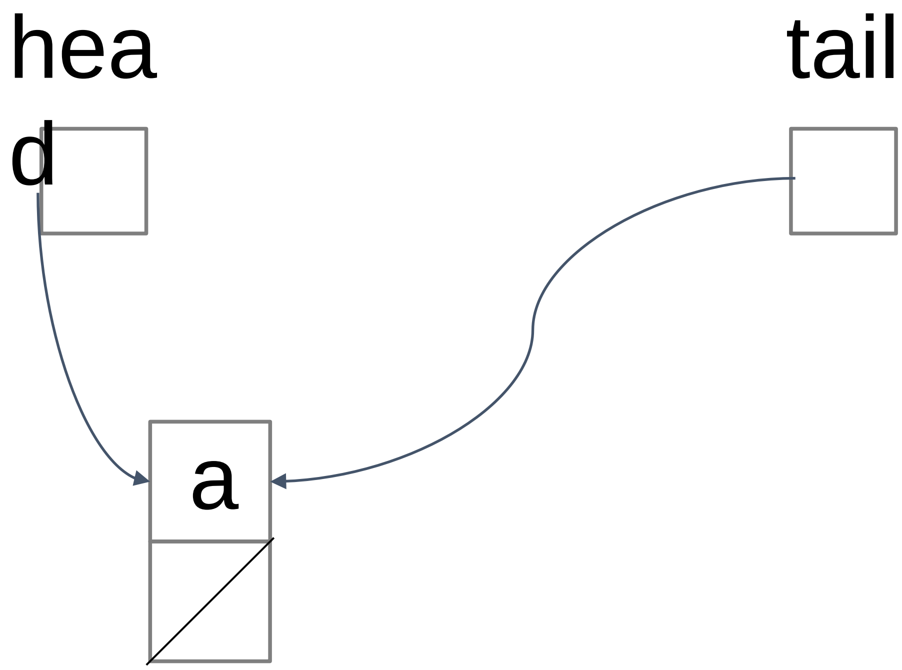
Queue Implementation with Linked List

Enqueue(a)

Queue Implementation with Linked List

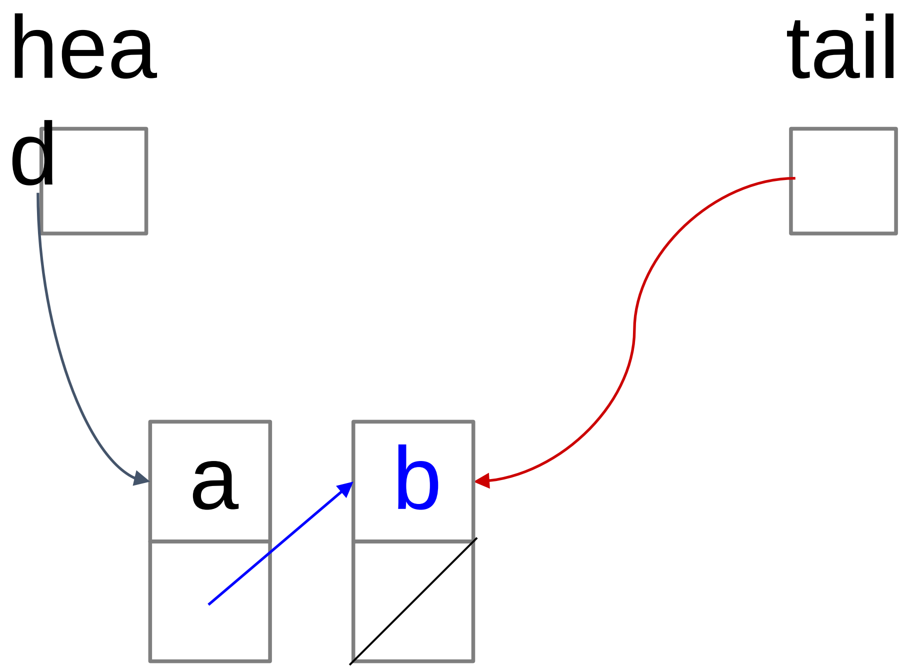


Queue Implementation with Linked List

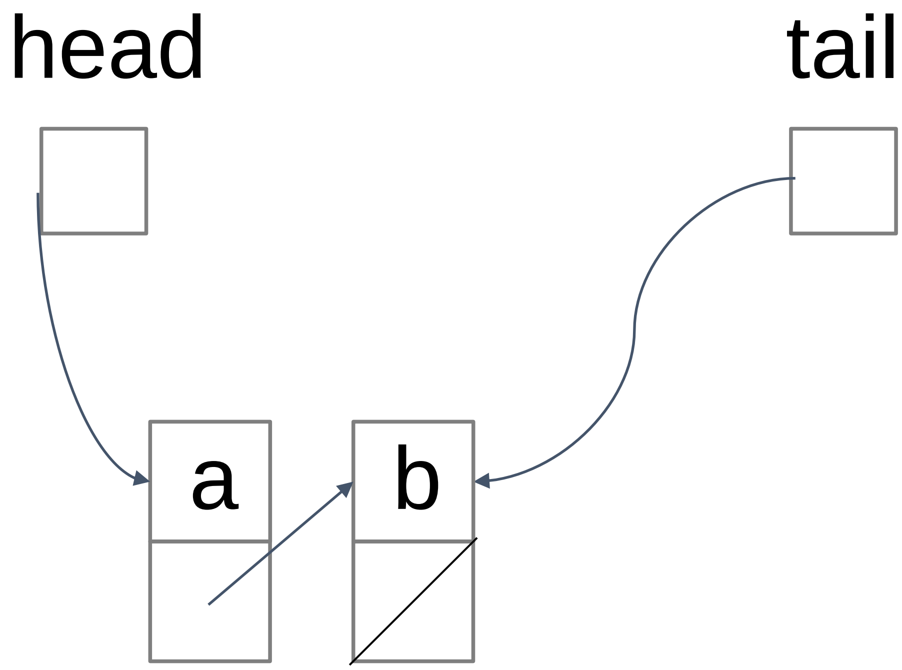


Enqueue(b)

Queue Implementation with Linked List

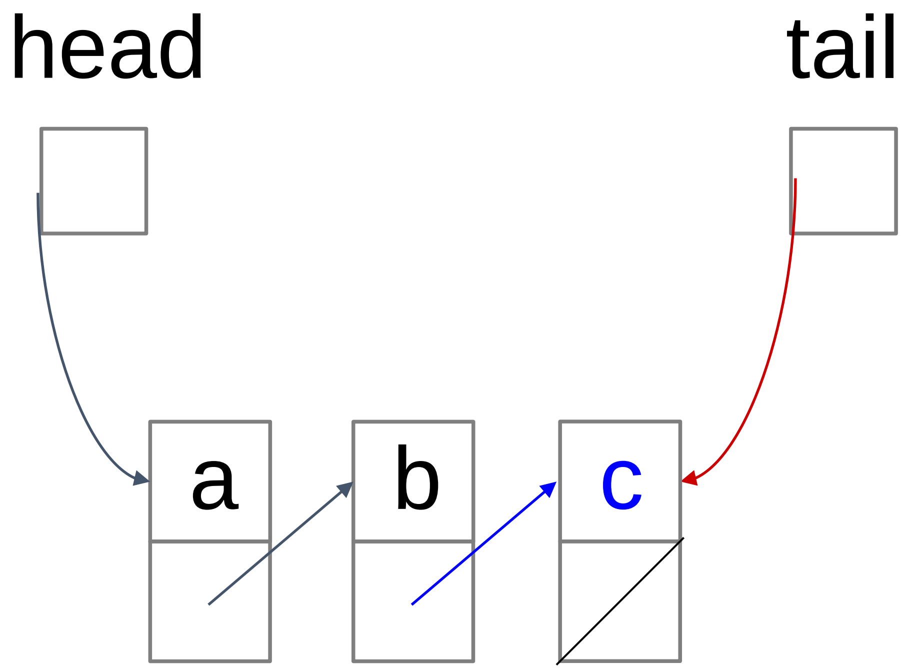


Queue Implementation with Linked List

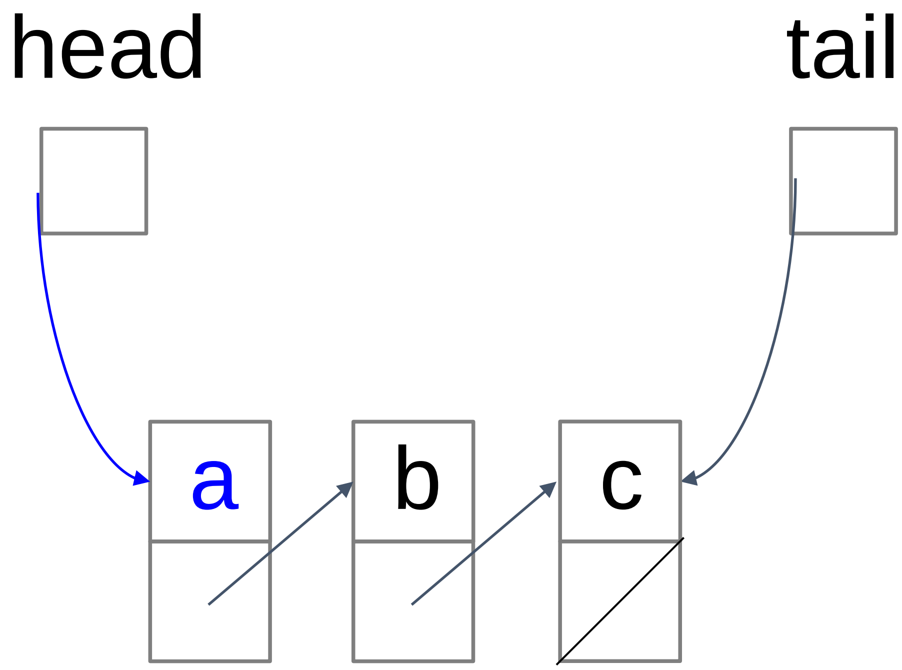


Enqueue(*c*)

Queue Implementation with Linked List

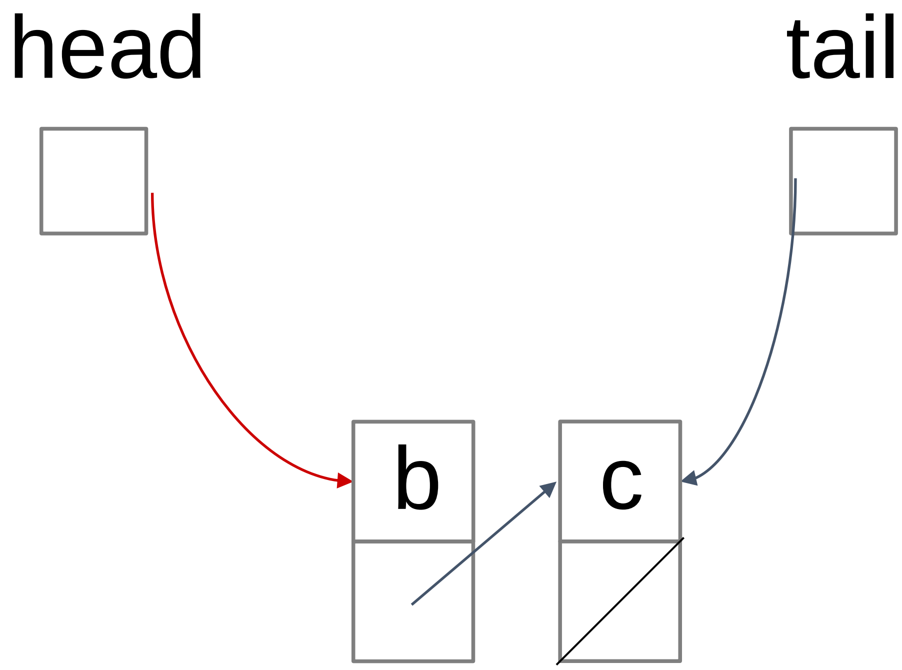


Queue Implementation with Linked List



Dequeue()

Queue Implementation with Linked List



Dequeue() → a

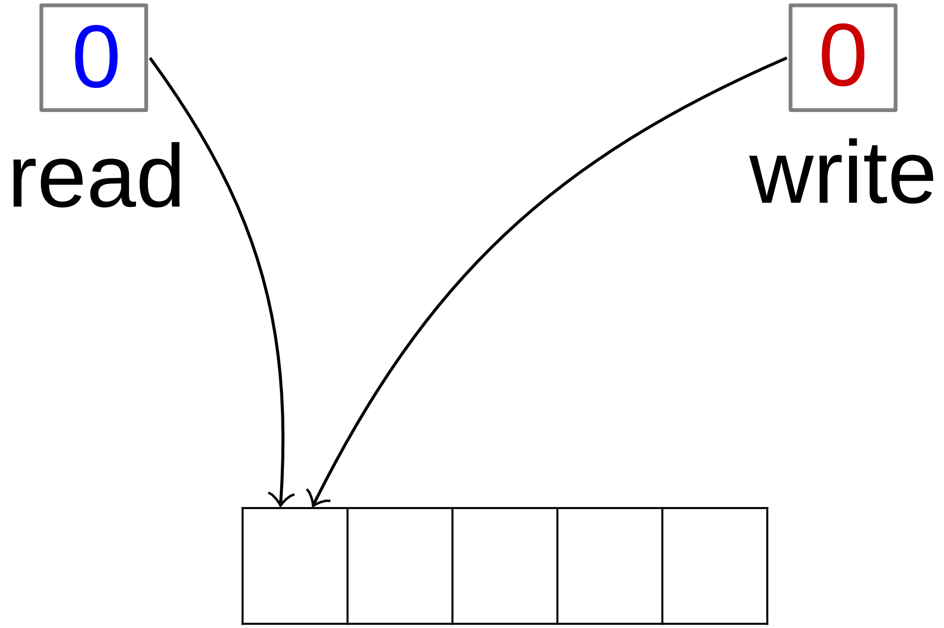
Queue Implementation with Linked List

- ➔ Use Linked List augmented with the *tail* pointer
- ➔ For *Enqueue(e)* use *list.add(e)* - which adds an element at the end
- ➔ For *Dequeue()* use *list.removeFirst()*
- ➔ For *IsEmpty()* use (*list.head* == NULL?)

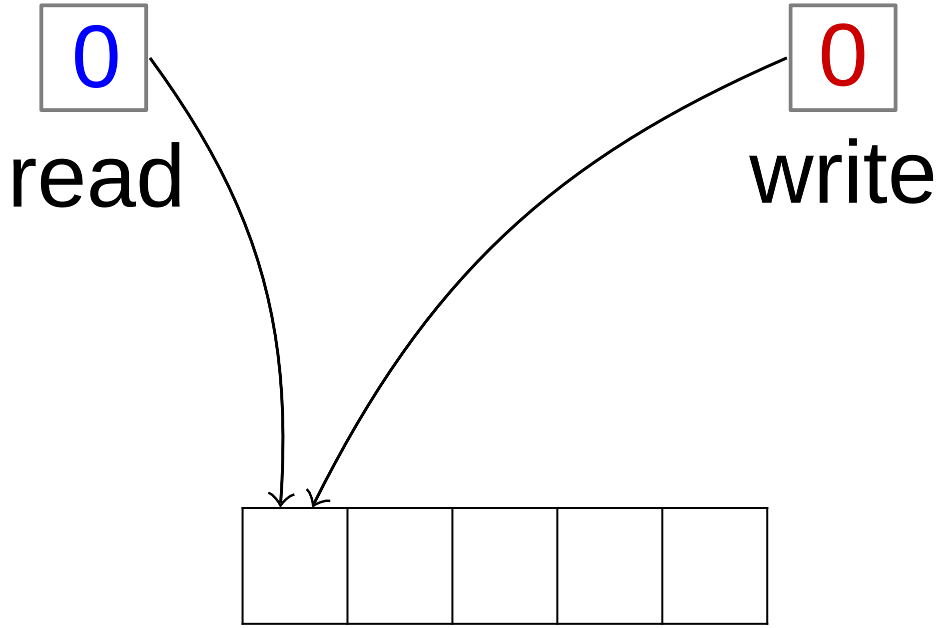
Queue ADT: cost of operations

	Link. List Impl. with tail	Array Impl.
Enqueue (e)	O(1)	
Dequeue()	O(1)	
IsEmpty()	O(1)	

Queue Implementation with Array

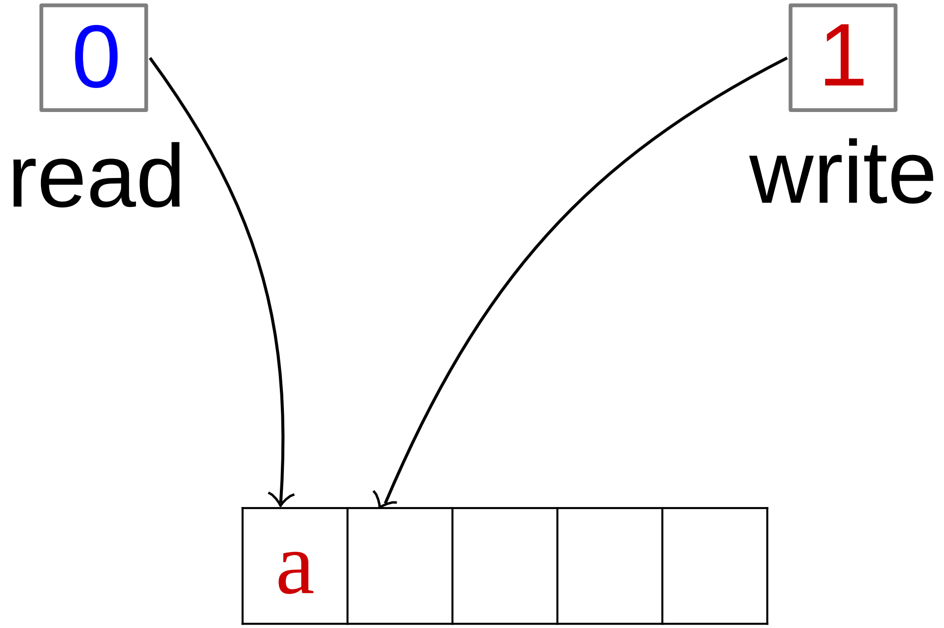


Queue Implementation with Array

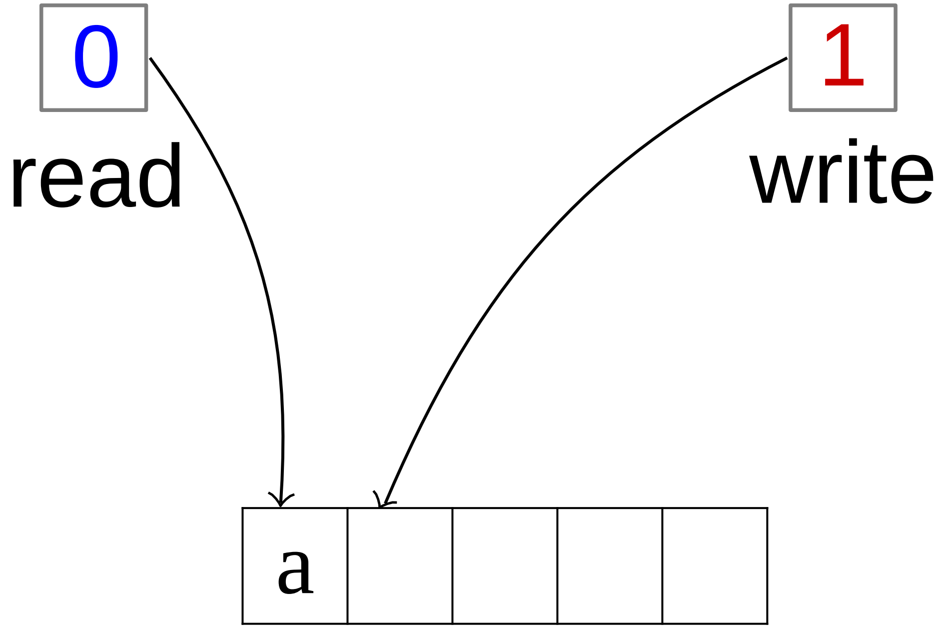


Enqueue(*a*)

Queue Implementation with Array

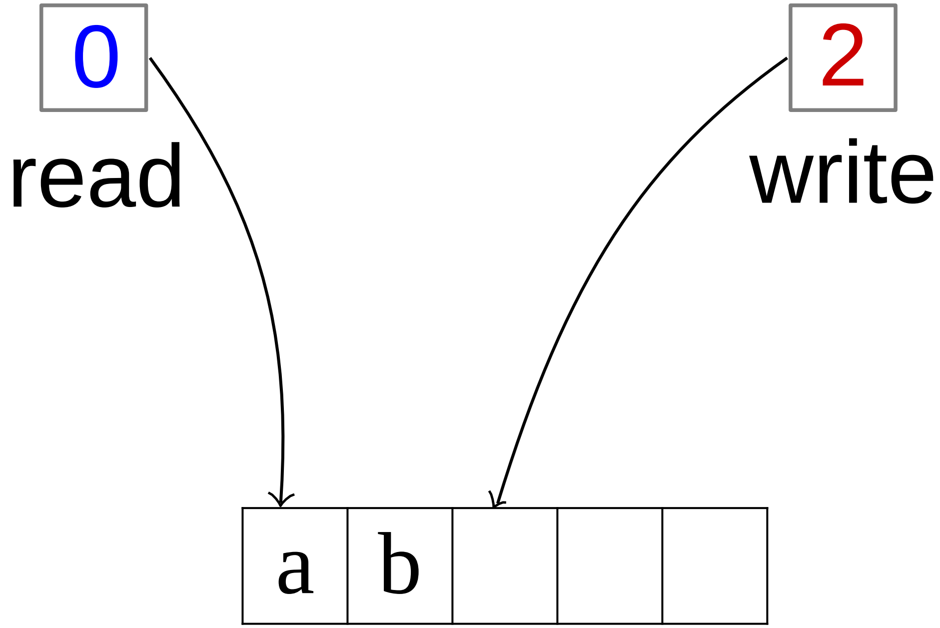


Queue Implementation with Array

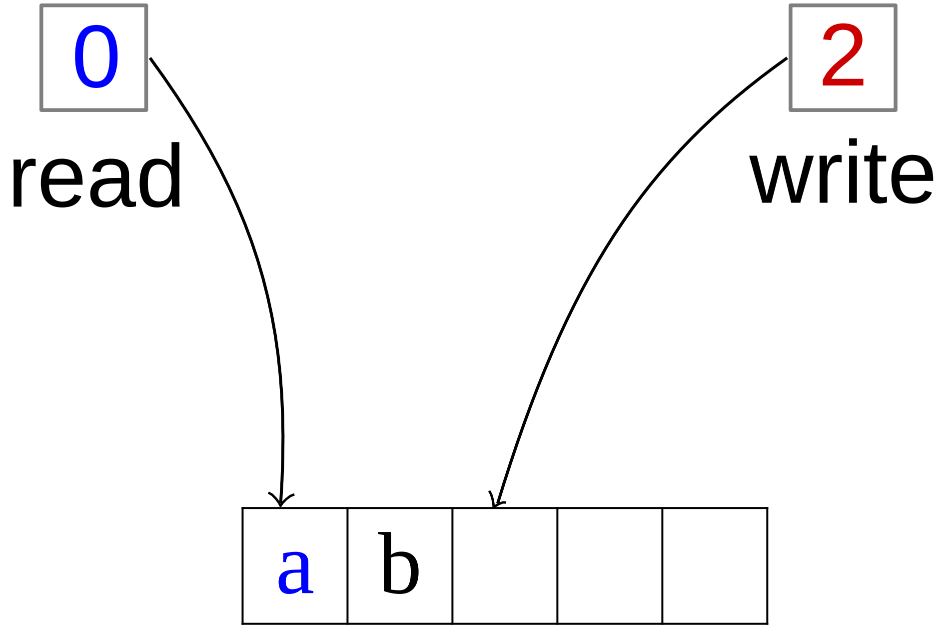


Enqueue(b)

Queue Implementation with Array

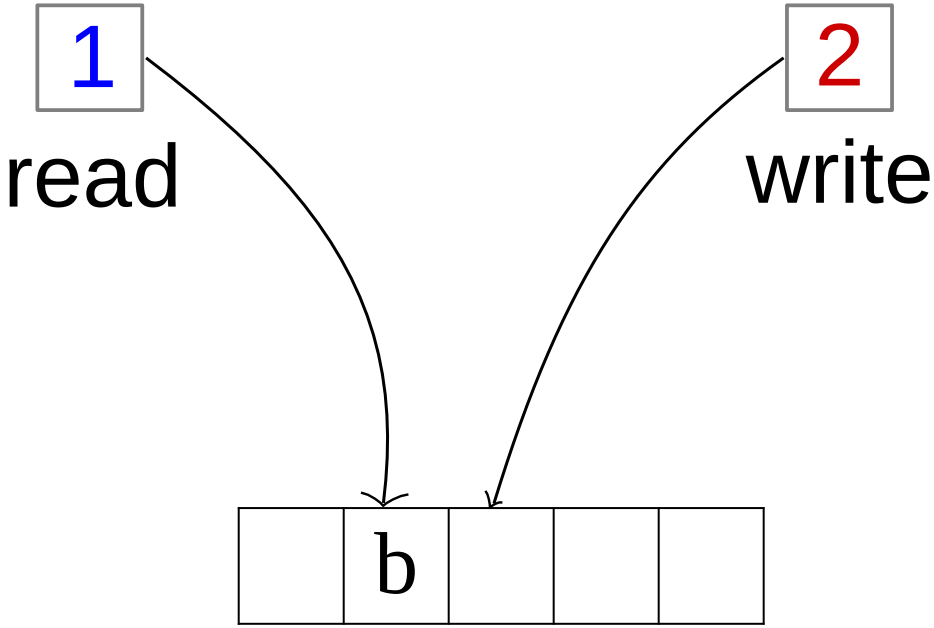


Queue Implementation with Array



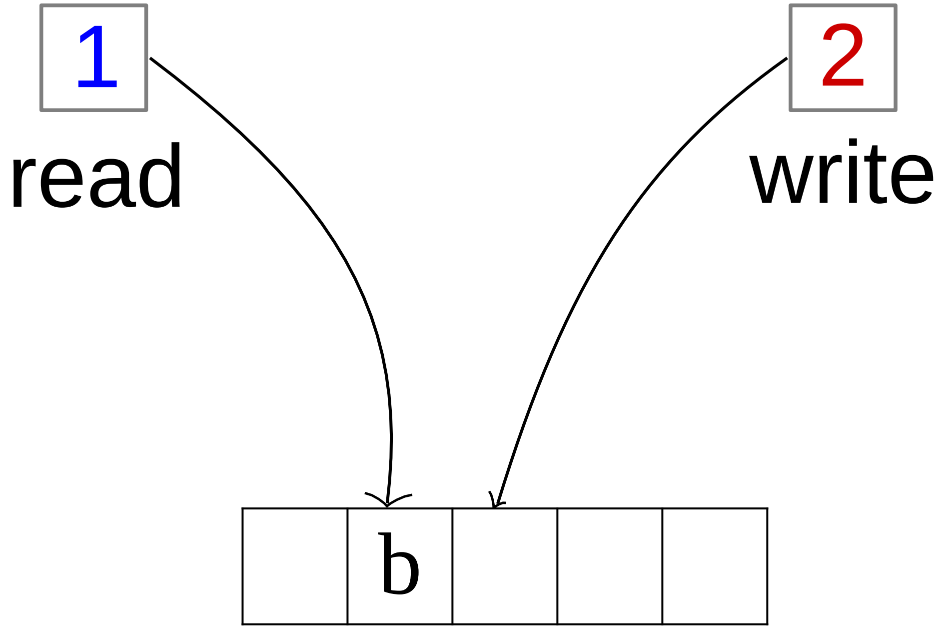
Dequeue()

Queue Implementation with Array



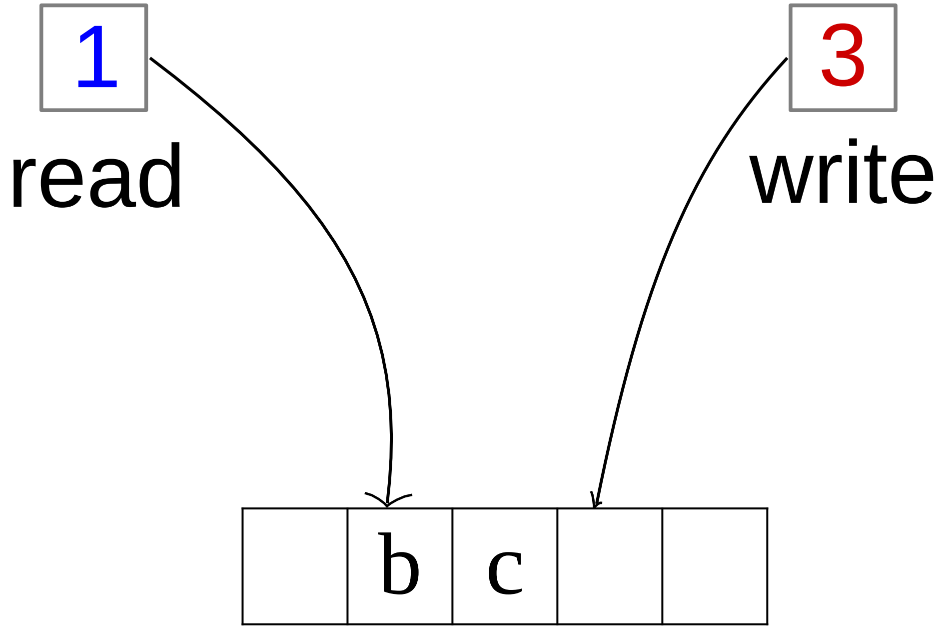
Dequeue() → a

Queue Implementation with Array

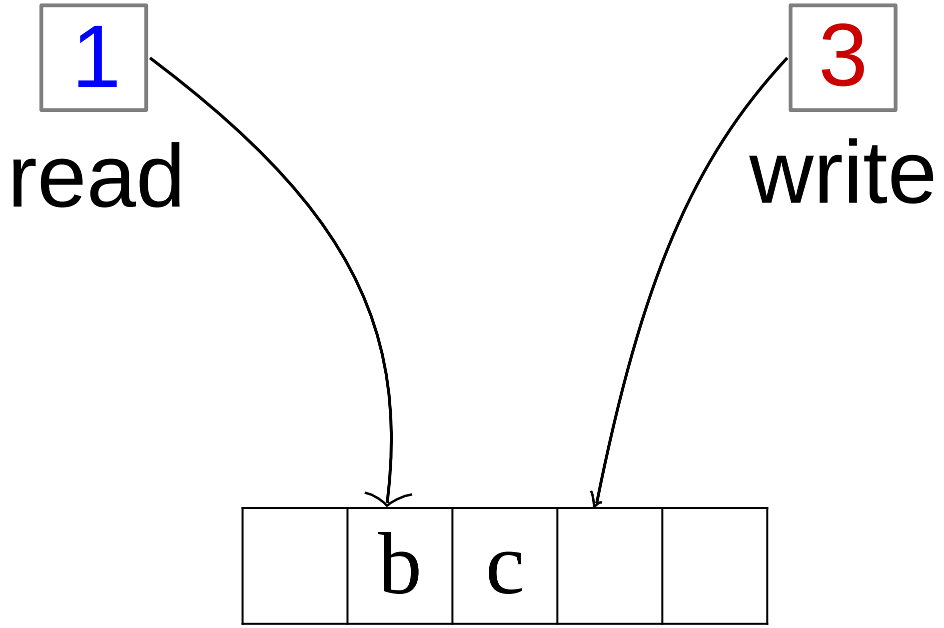


Enqueue(c)

Queue Implementation with Array

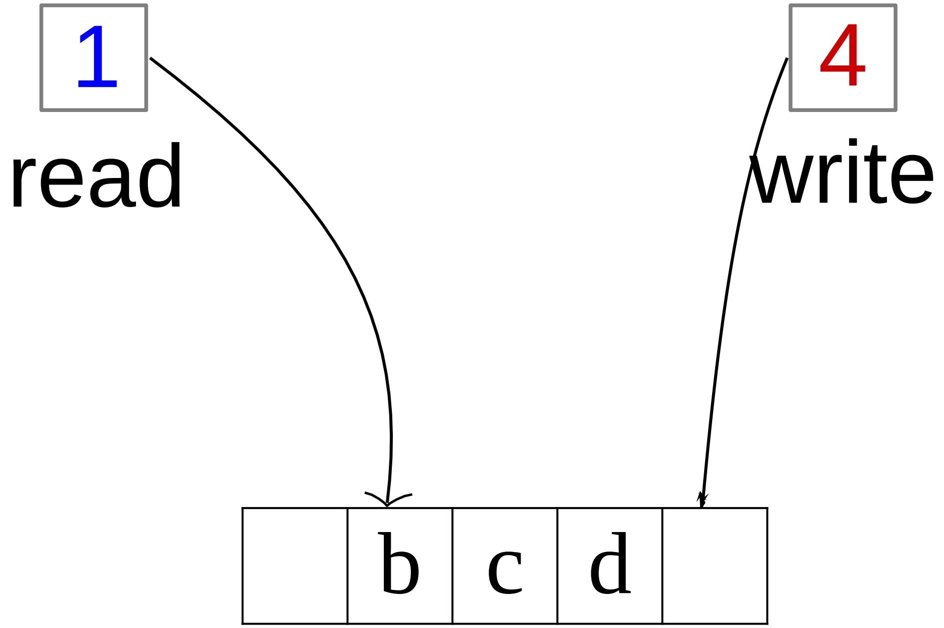


Queue Implementation with Array

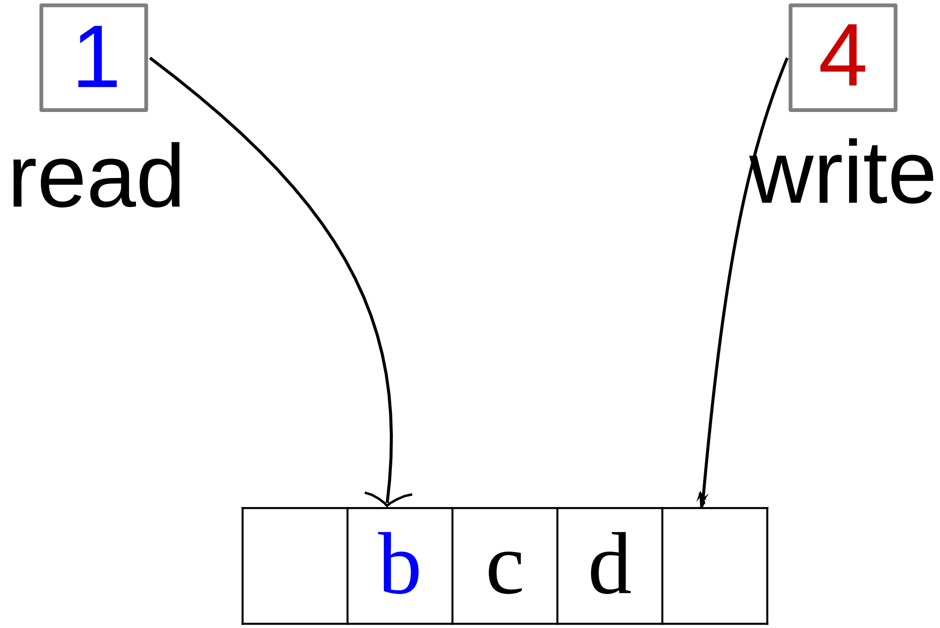


Enqueue(d)

Queue Implementation with Array

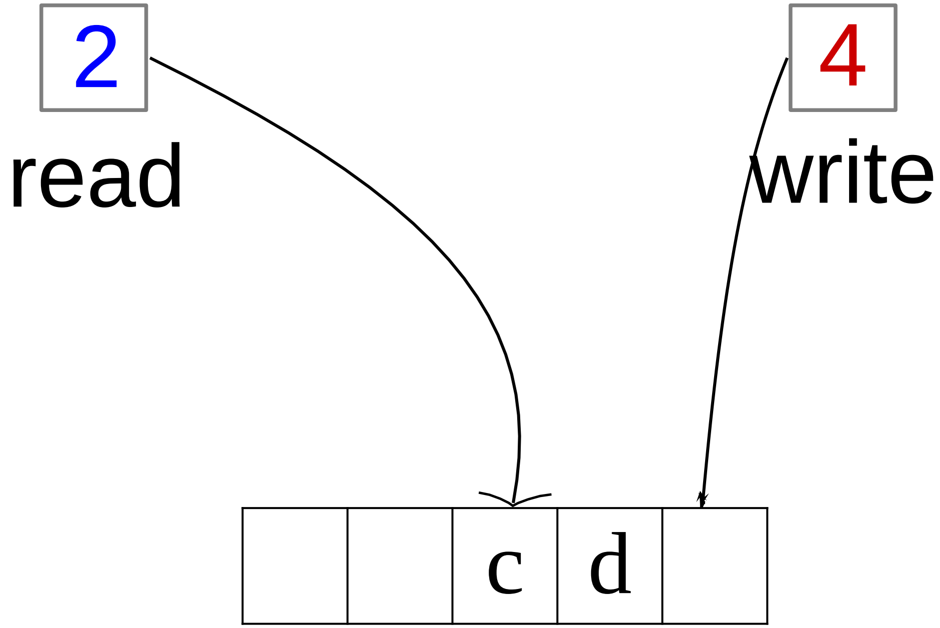


Queue Implementation with Array



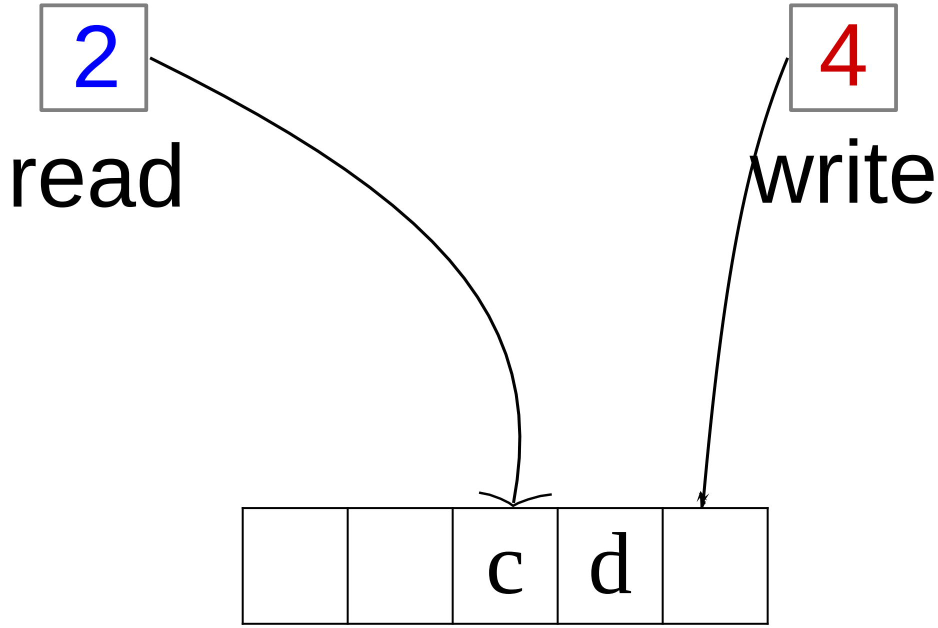
Dequeue()

Queue Implementation with Array



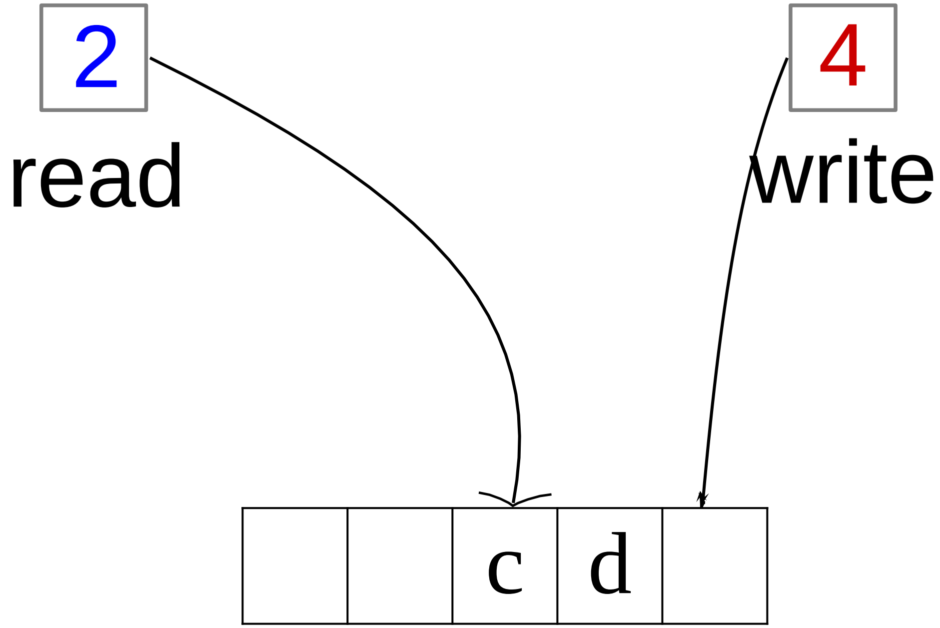
Dequeue() → *b*

Queue Implementation with Array



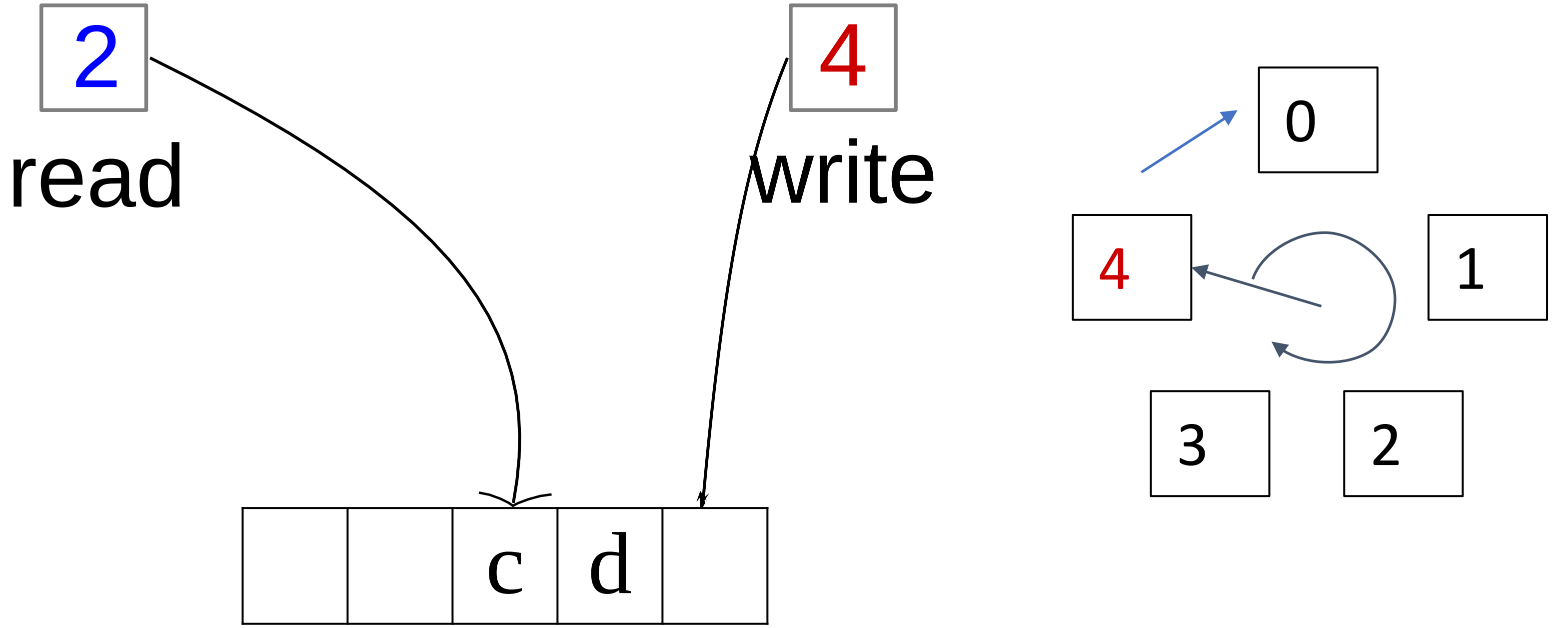
Enqueue(e)

Concept of a Circular Array



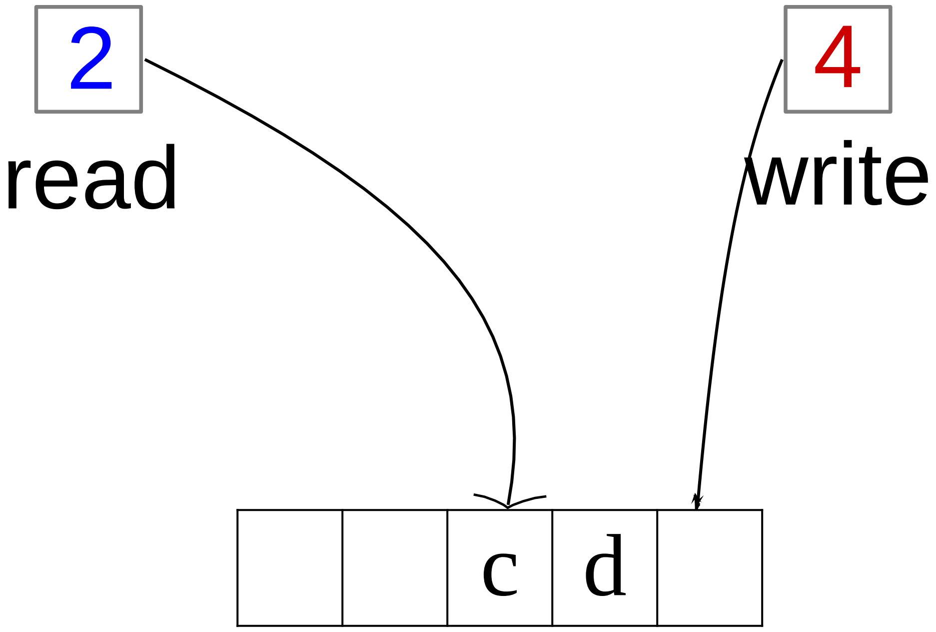
Enqueue(e)

Concept of a Circular Array



Enqueue(e)

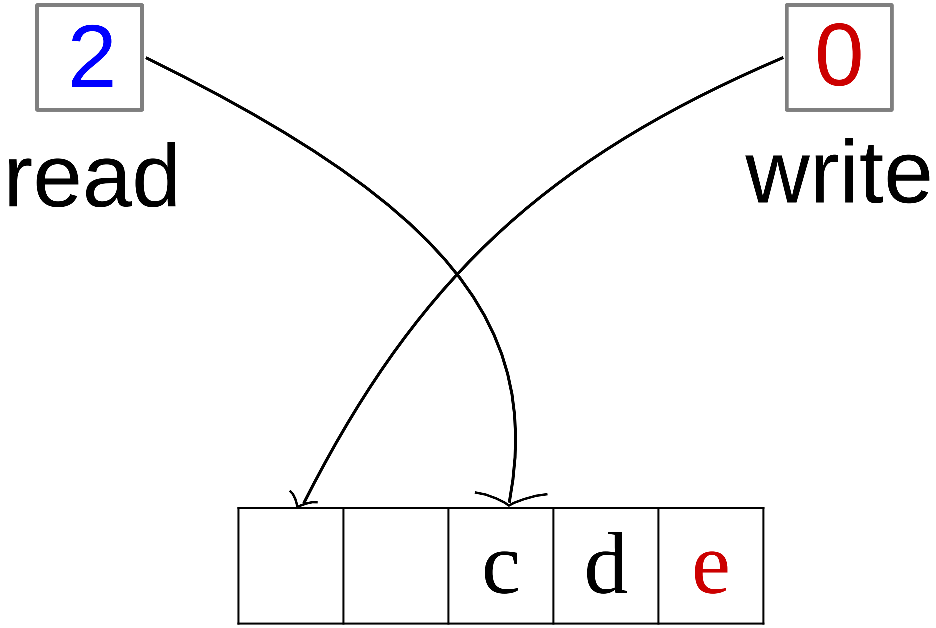
What will be the value of the read and write pointers after the operation is completed?

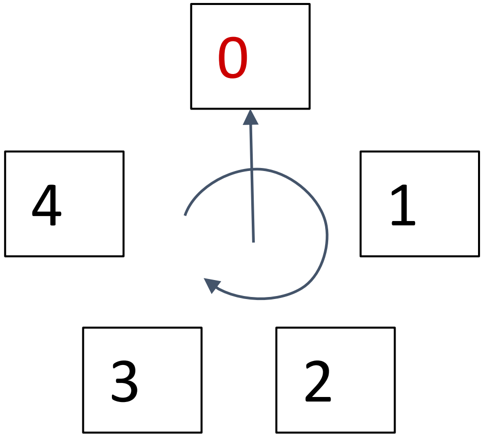


Enqueue(e)

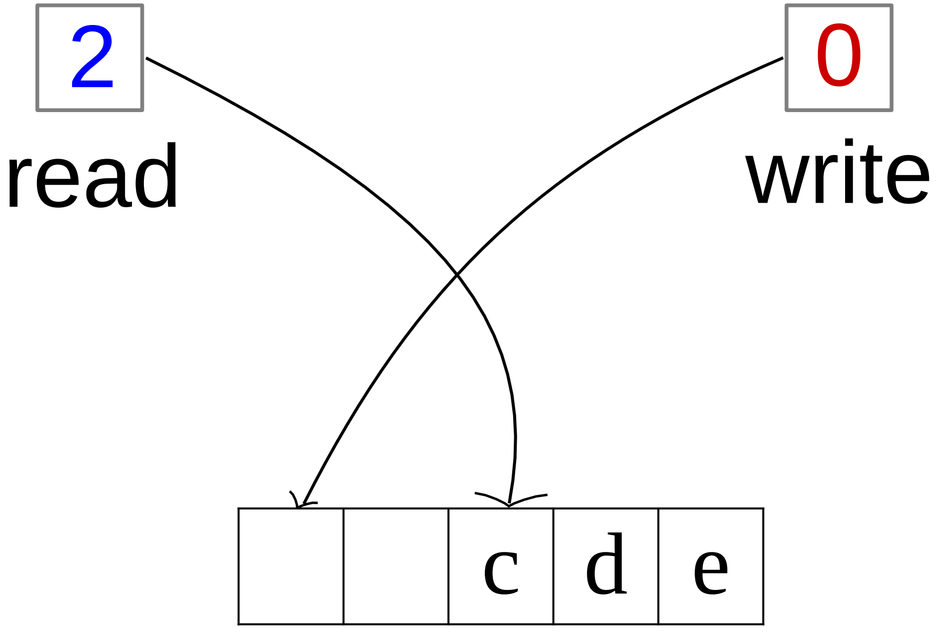
A. read=2, write=5
B. read=2, write=0
C. read=0, write=0
D. read=2, write=1
E. none of the above

Queue Implementation with Array



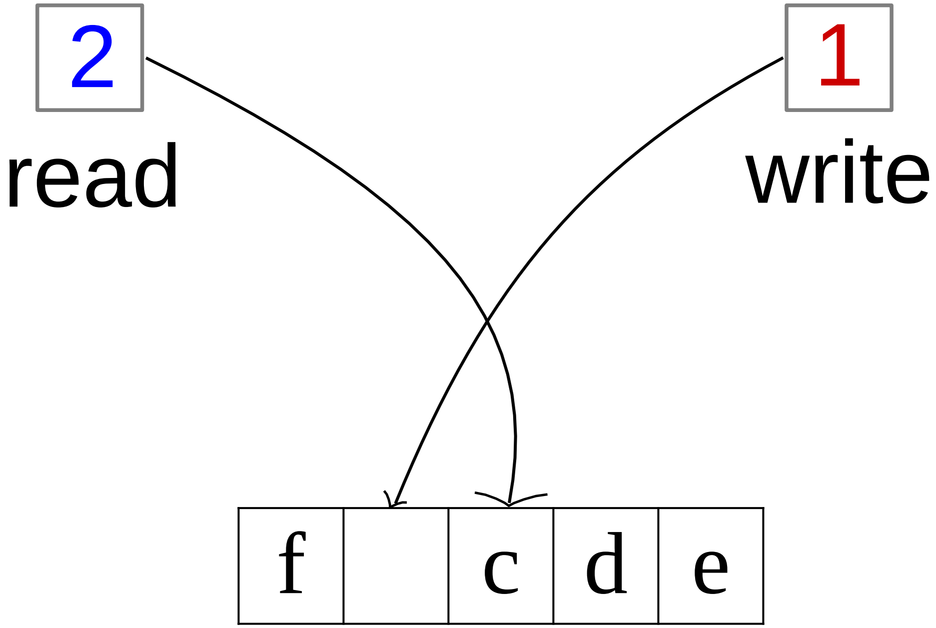


Queue Implementation with Array

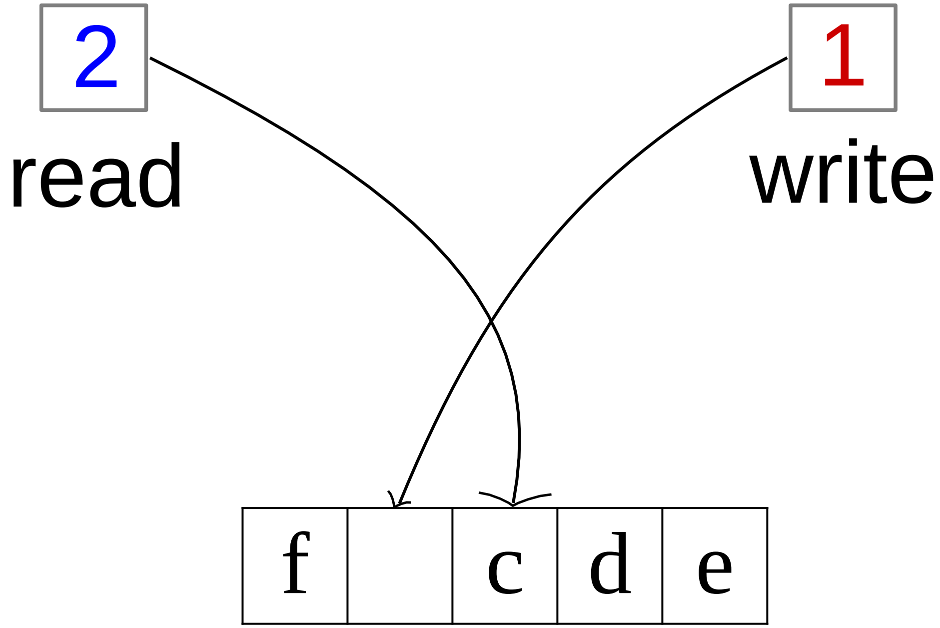


Enqueue(f)

Queue Implementation with Array

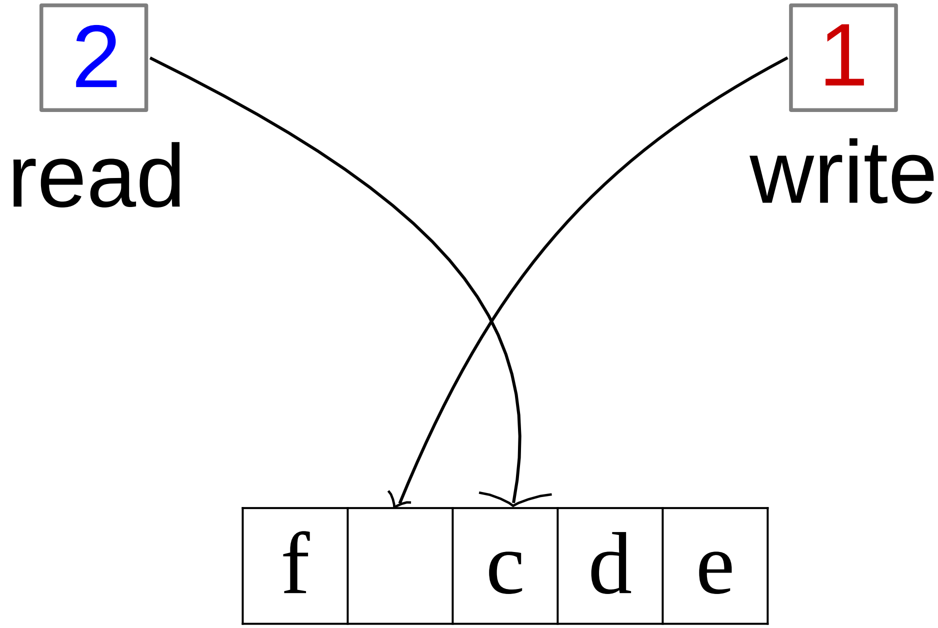


Queue Implementation with Array



Enqueue(g)

Queue Implementation with Array

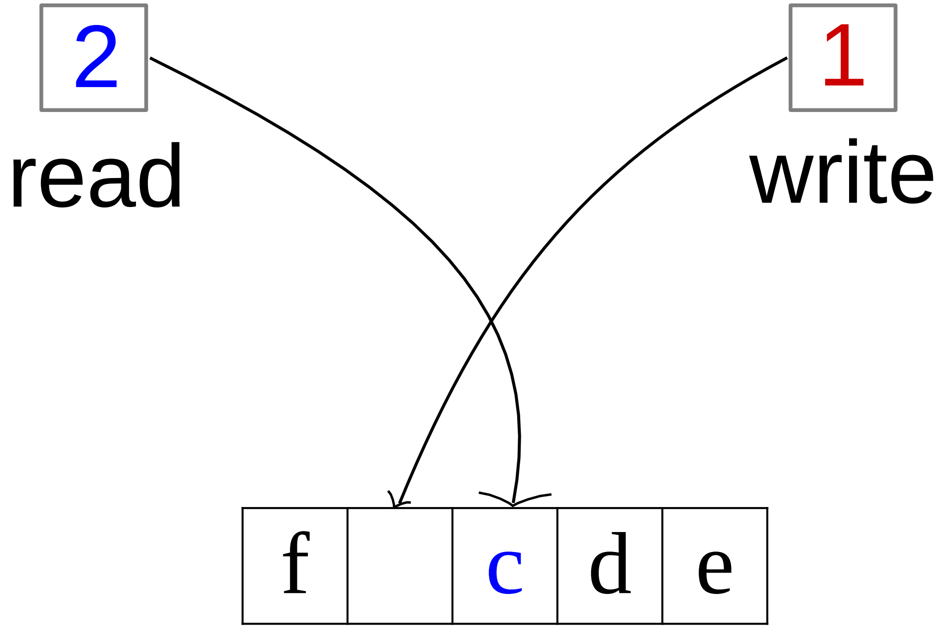


Enqueue(g) → ERROR

Cannot set read = write

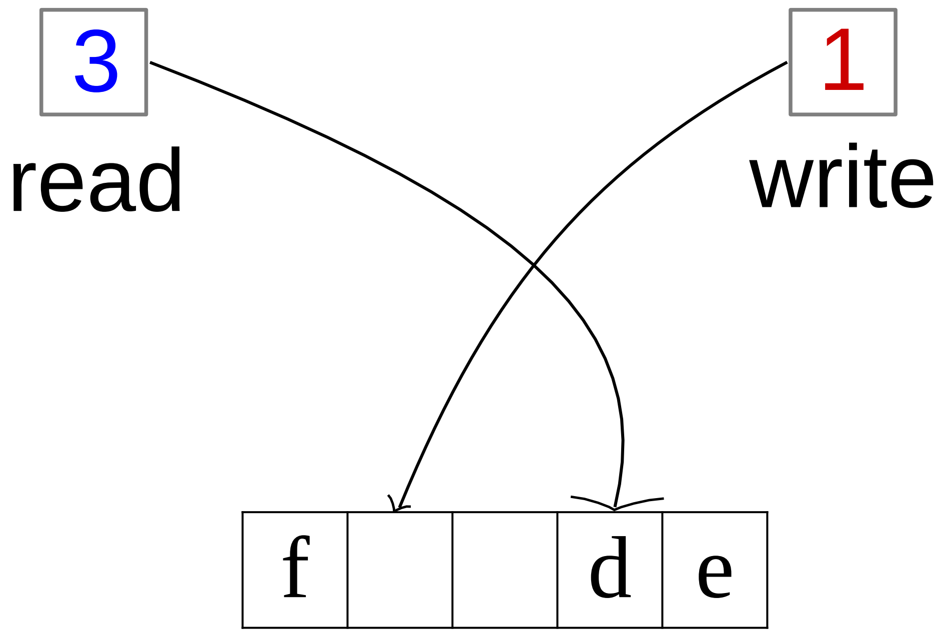
isFull() → True

Queue Implementation with Array



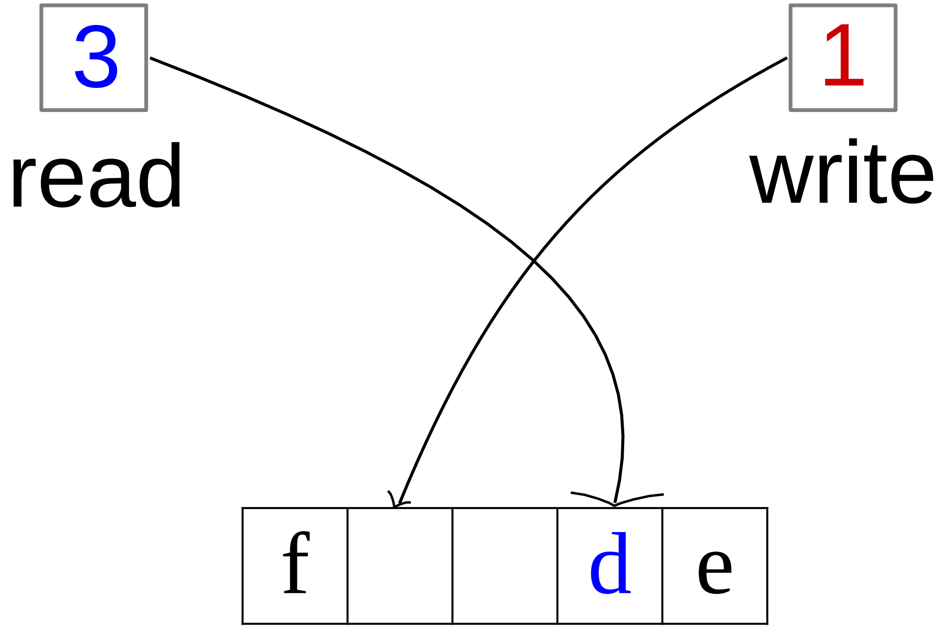
Dequeue()

Queue Implementation with Array



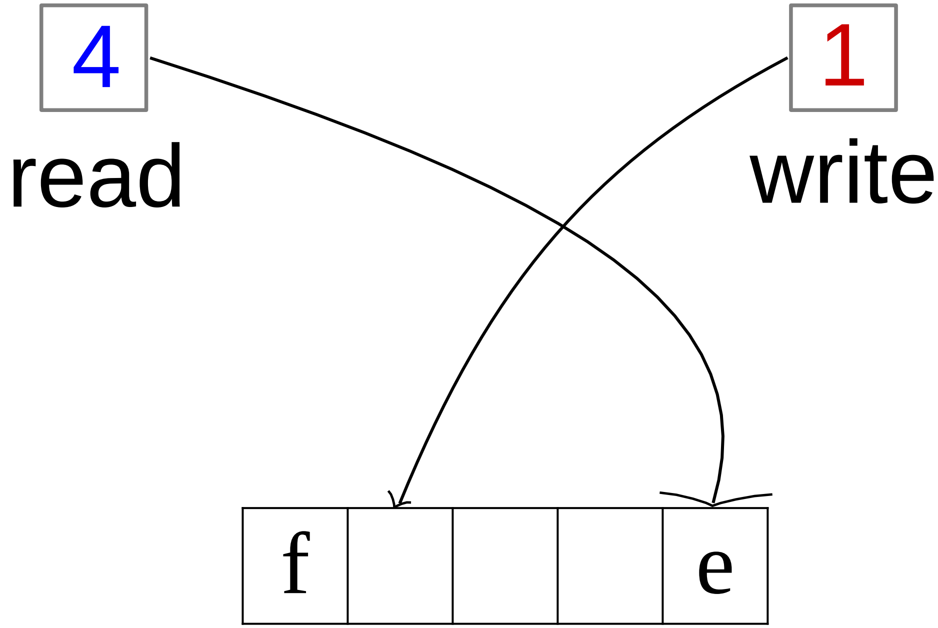
Dequeue() → *c*

Queue Implementation with Array



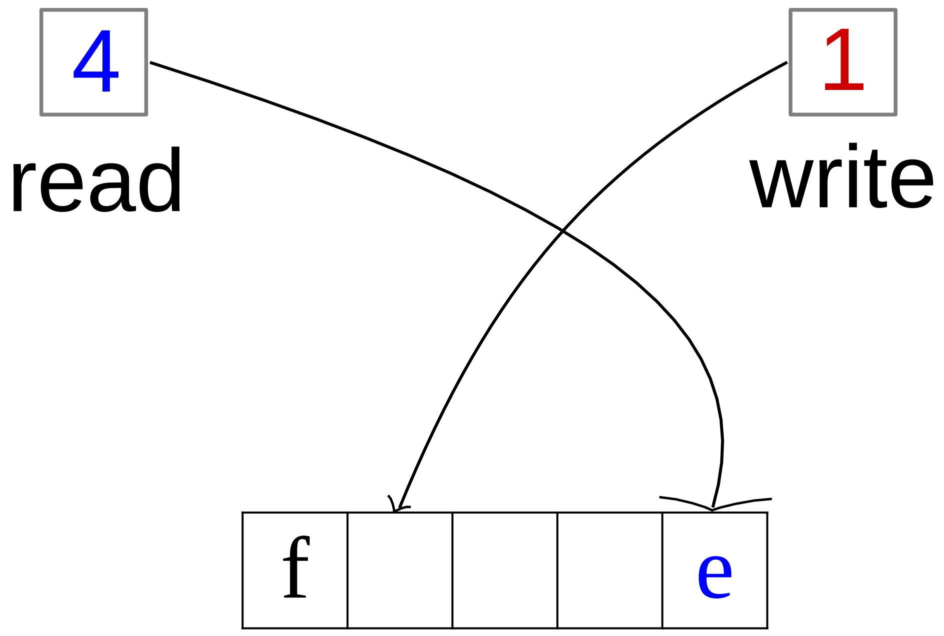
`Dequeue()`

Queue Implementation with Array



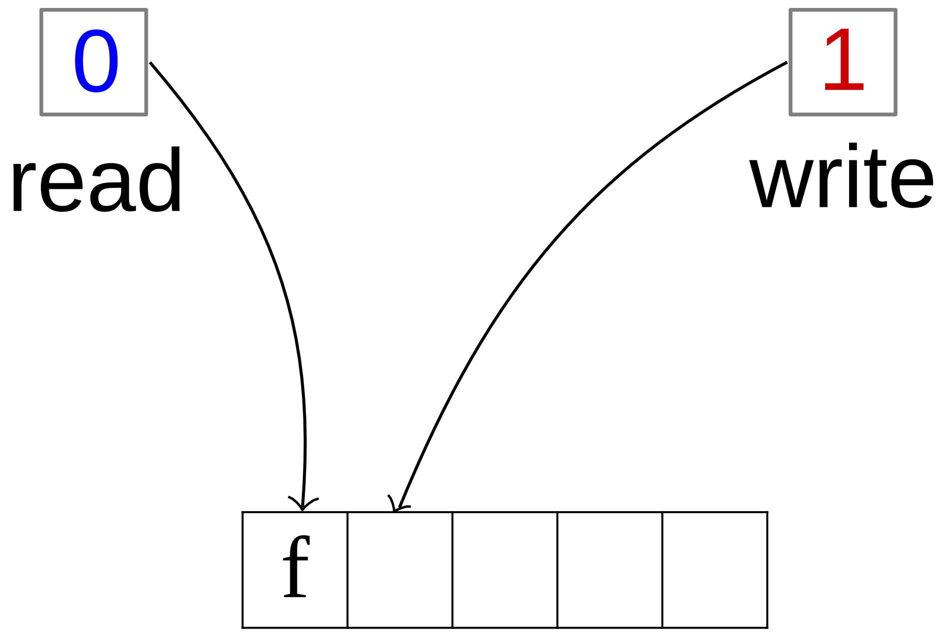
Dequeue() → *d*

Queue Implementation with Array



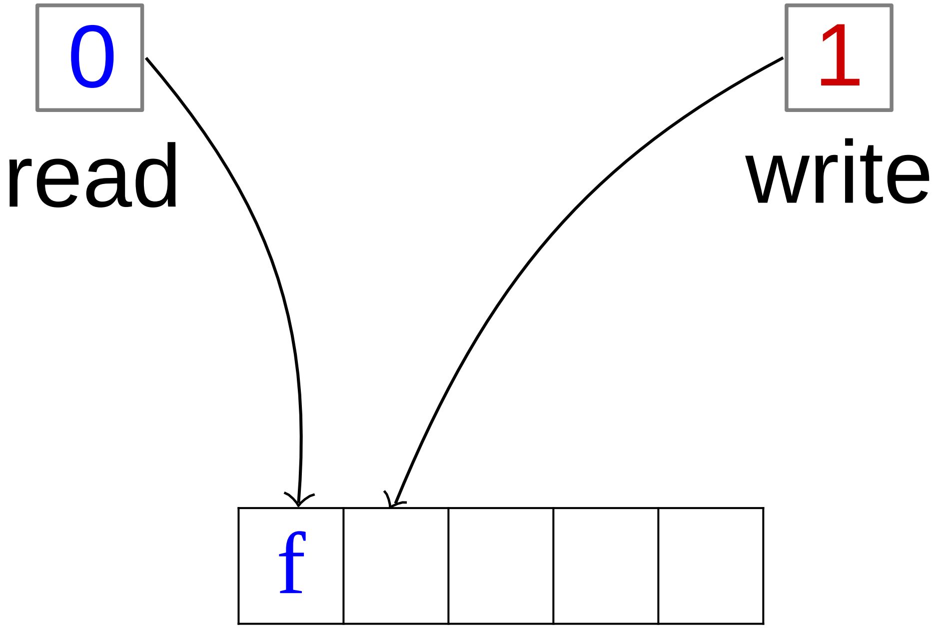
Dequeue()

Queue Implementation with Array



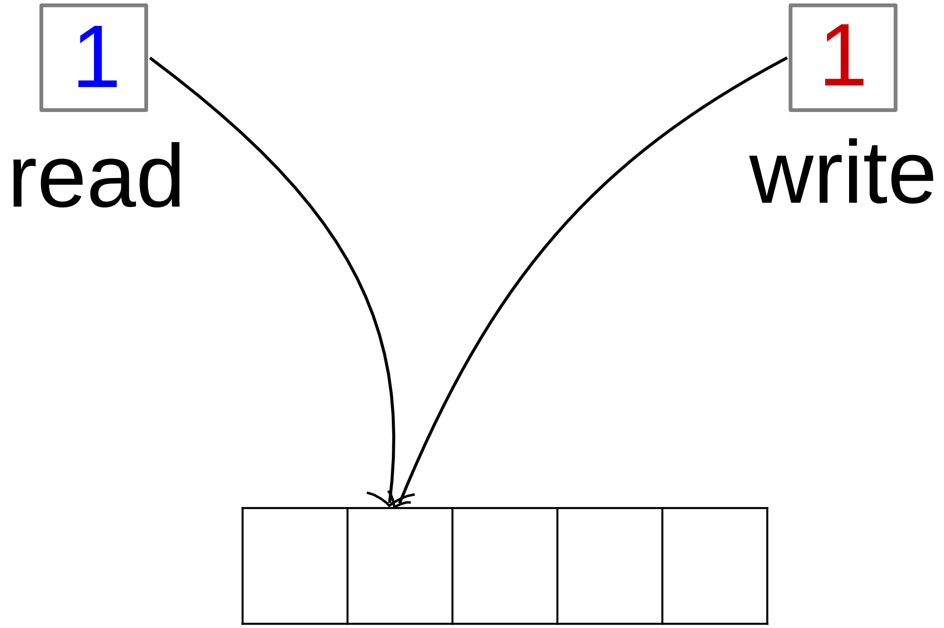
Dequeue() → e

Queue Implementation with Array



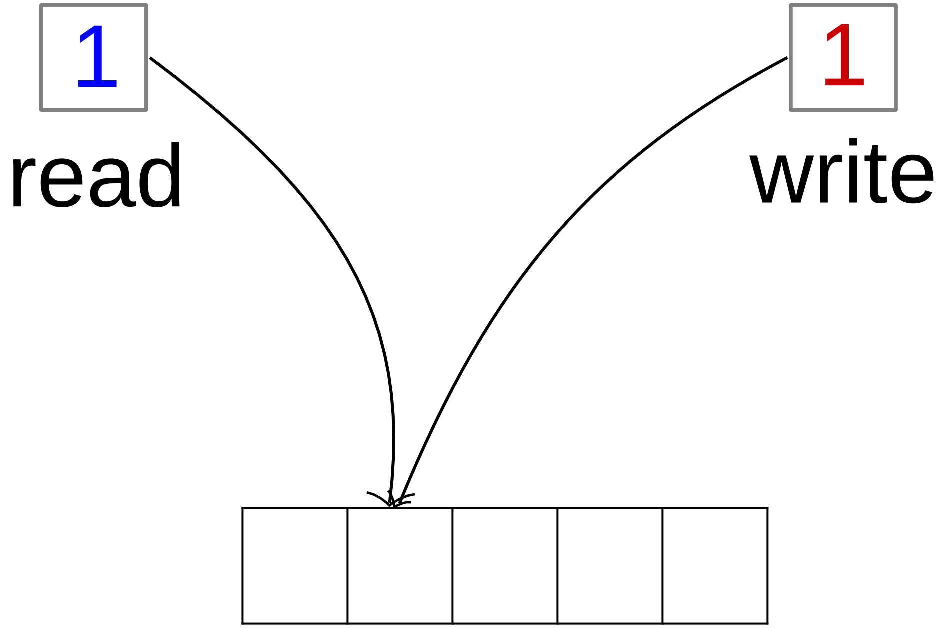
Dequeue()

Queue Implementation with Array



Dequeue() → *f*

Queue Implementation with Array



read==write

IsEmpty() → True

Queue Implementation with Array

- *Queue* ADT can be implemented with a *circular* Array
- We need 2 pointers (indexes in 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

Queue ADT: cost of operations

	Link. List Impl. with tail	Array Impl. circular
Enqueue (e)	O(1)	O(1) amortized
Dequeue()	O(1)	O(1)
IsEmpty()	O(1)	O(1)

Queue: Summary

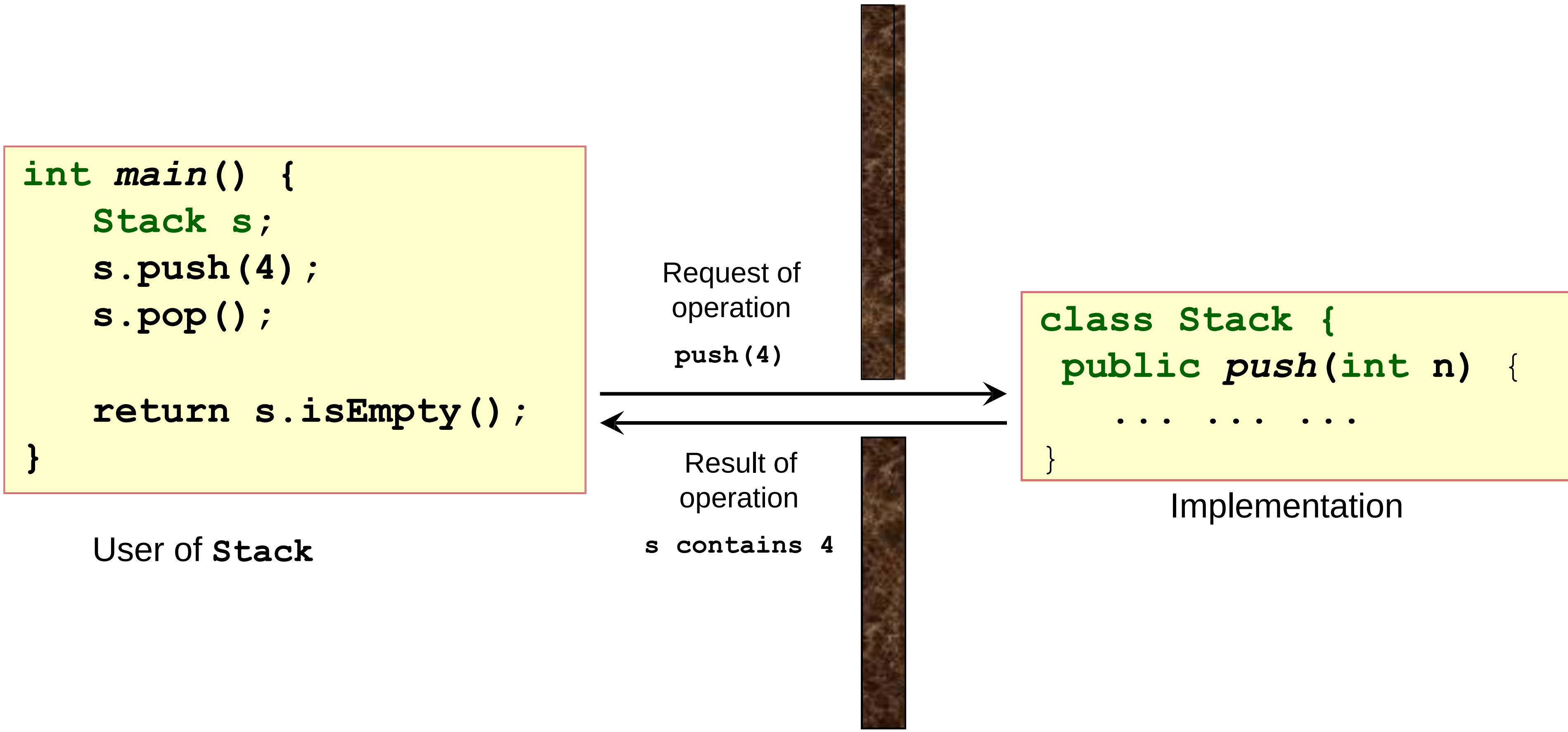
- **Queue ADT** can be implemented with either a *Linked List (with tail) or* an *Array (Circular)* Data structure
- Each queue operation is $O(1)$: `Enqueue, Dequeue, IsEmpty`
- Considerations:
  - Linked Lists have unlimited storage
  - Arrays need to be resized when full
  - Linked Lists have simpler maintenance

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 should **not** affect the users of ADT

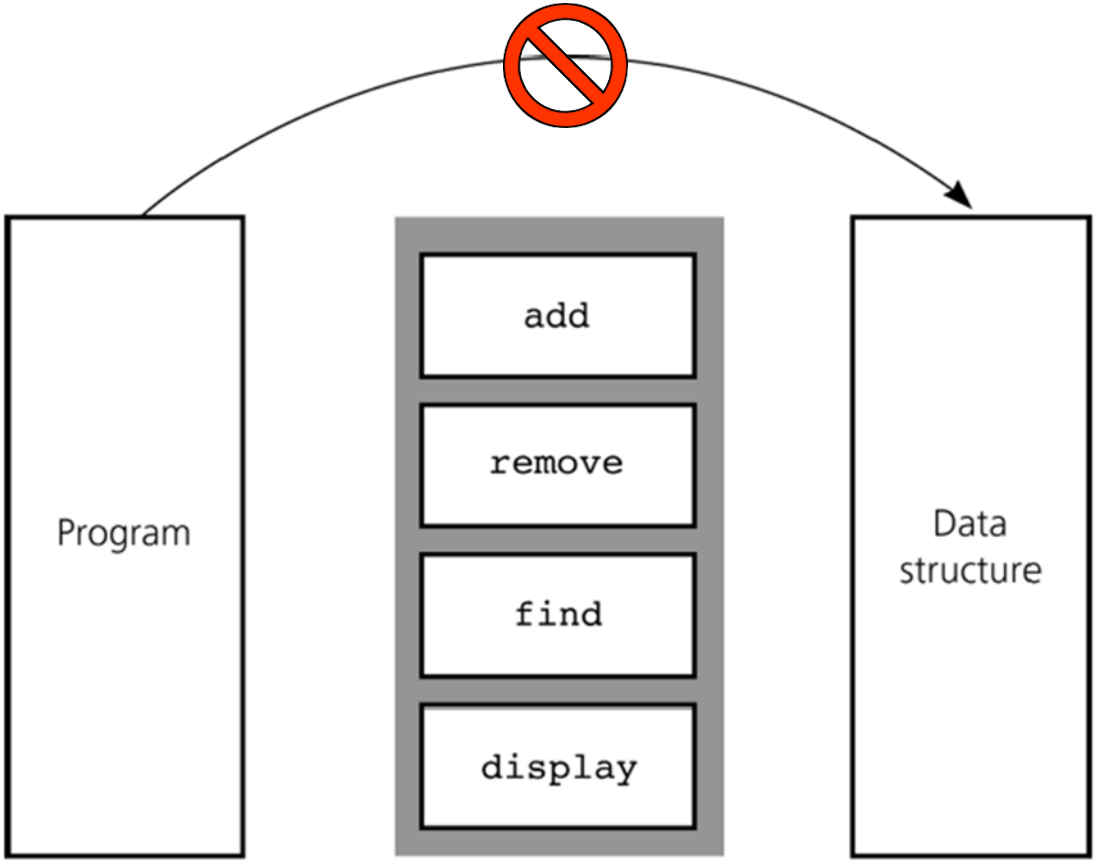
Specification as slit in the wall



- Users only depend on specifications (interface):
  - Method signature and return type

ADT and encapsulation

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



Sample application

Balanced Brackets Problem

Input: A string *str* consisting of '(', ')', '[', ']','{', '}' characters.

Output: Return whether or not the string's parentheses and brackets are balanced.

Examples

Balanced:

“([])[]()”,

“((([([])]))())”

Unbalanced:

“(]()”

“][”

“([)]”

“([]”

Which ADT can help us to solve the problem of balanced brackets?

Stack?

List?

Sorted list?

Queue?

...?

Is this solution correct?

```
stack = empty Stack()

for each character X in text:
   if X is one of ‘{‘, ‘[‘, ‘(‘
      push X to the stack
   if X is one of ‘}’,’]’,’)’
      Y = stack.pop()
      if X does not match Y
         return “Unbalanced”
return “Balanced”
```

A. Yes

B. No

Is this solution correct?

```
stack = empty Stack()
```

```
for each character X in text:
   if X is one of ‘{‘, ‘[‘, ‘(‘
      push X to the stack
   if X is one of ‘}’,’]’,’)’
      Y = stack.pop()
      if X does not match Y
         return “Unbalanced”
return “Balanced”
```

A. Yes

B. No

Try: text="[{ }"