Abstract Data Types and Data structures

Lecture 02.04
by Marina Barsky

Abstraction

Abstraction in Programming

- *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 C++ or Java, use *class* construct to create a new ADT

ADT includes:

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

Example 1: HR roster

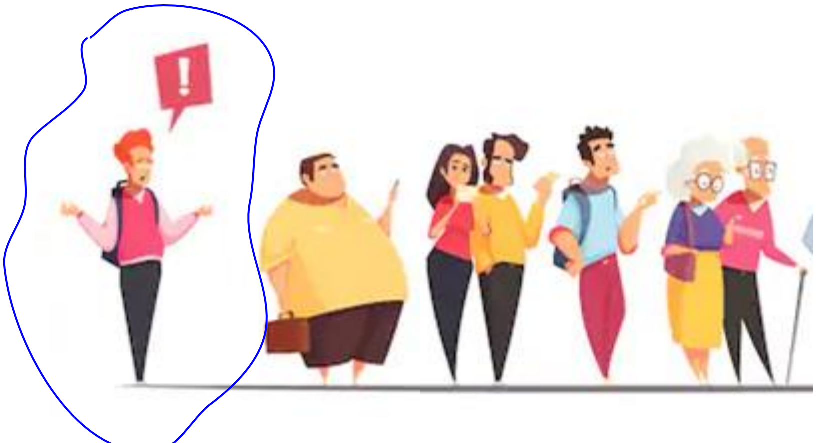
We want to model a list of company employees

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

Example 1: HR roster

We want to model a list of company employees

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



Example 1: HR roster

We want to model a list of company employees

- ➢ 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)

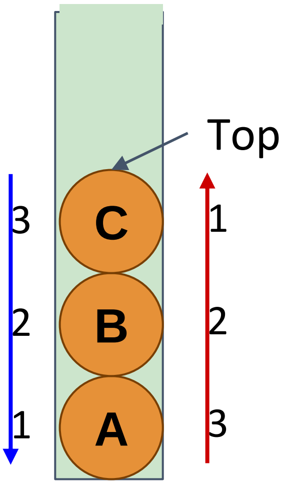
Example 1: HR roster

We want to model a list of company employees

- ➢ 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 solve this problem using ***Stack*** Abstract Data Type
- Stack stores a list 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 supports following operations:

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

→*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

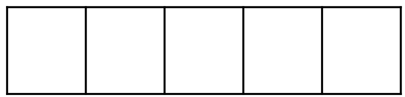
to be disjoint:

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

Stack Implementation with Array

size: 0

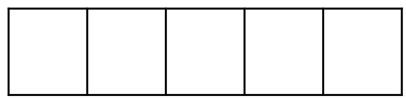
capacity: 5



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			

$$Top() \rightarrow b$$

Stack Implementation with Array

size: 2

capacity: 5

a	b			

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			

$Pop() \rightarrow c$

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() $\rightarrow$ *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

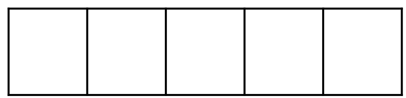
a				

Pop( )

Stack Implementation with Array

size: 0

capacity: 5

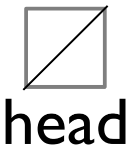


$IsEmpty() \rightarrow True$

Stack ADT: cost of operations

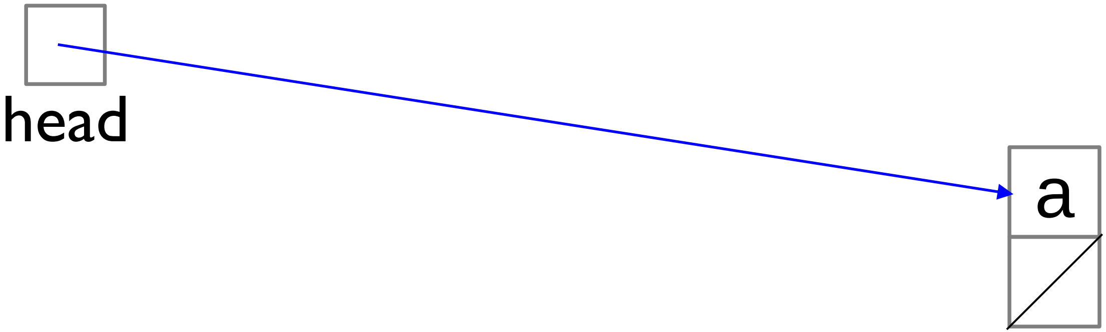
	Array Impl.	
Push(e)	O(1)	
Top()	O(1)	
Pop()	O(1)	
IsEmpty()	O(1)	
IsFull()	O(1)	

Stack Implementation with Linked List

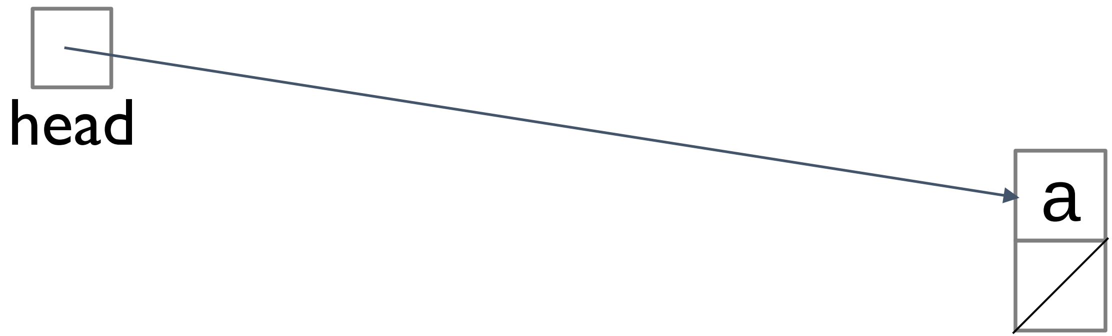


Push(a)

Stack Implementation with Linked List

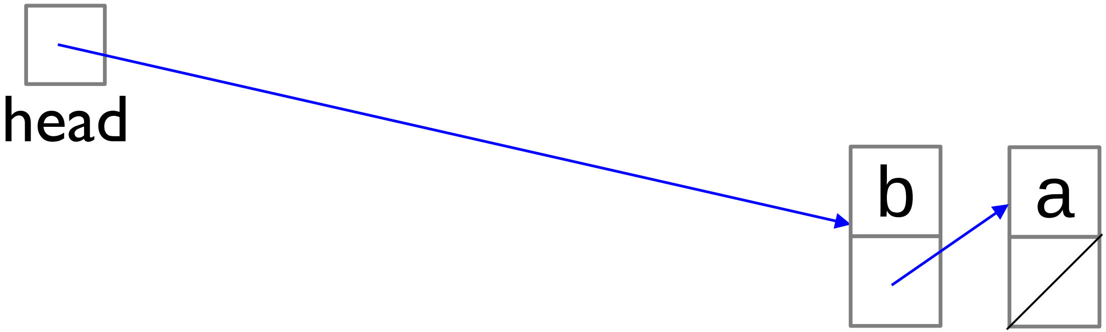


Stack Implementation with Linked List

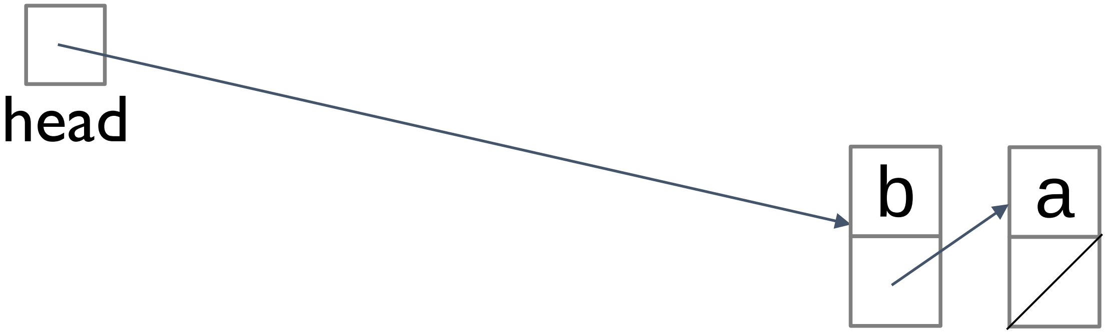


Push(b)

Stack Implementation with Linked List

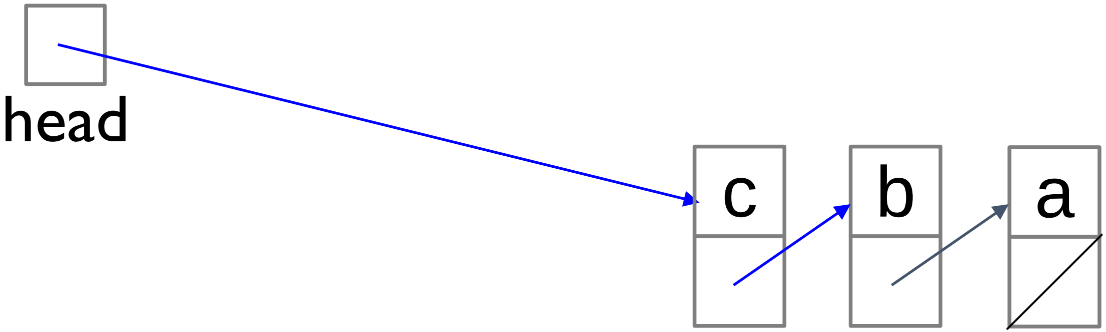


Stack Implementation with Linked List

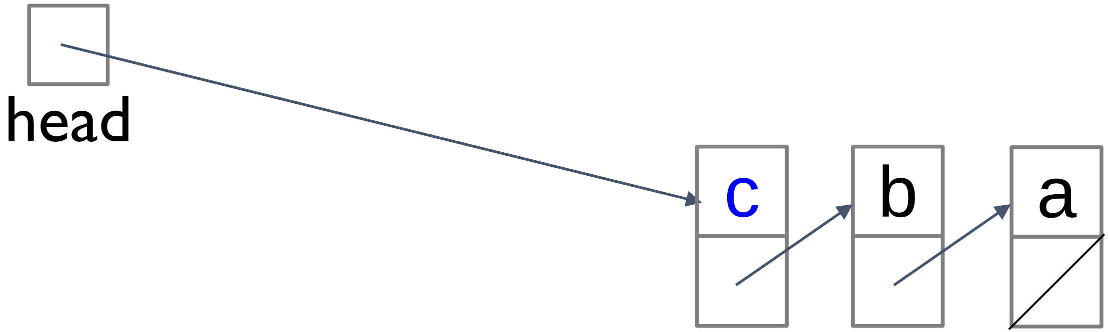


Push(c)

Stack Implementation with Linked List

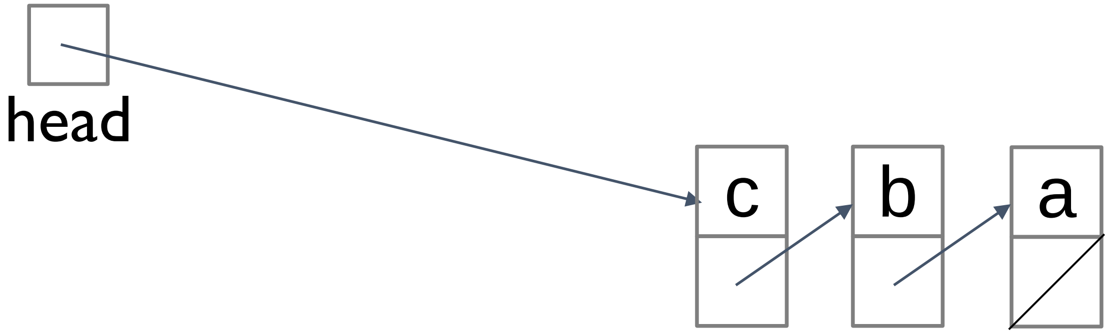


Stack Implementation with Linked List



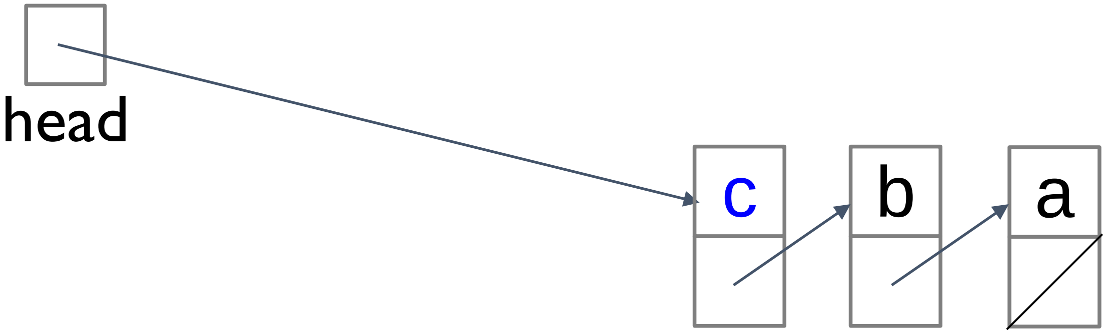
Top()

Stack Implementation with Linked List



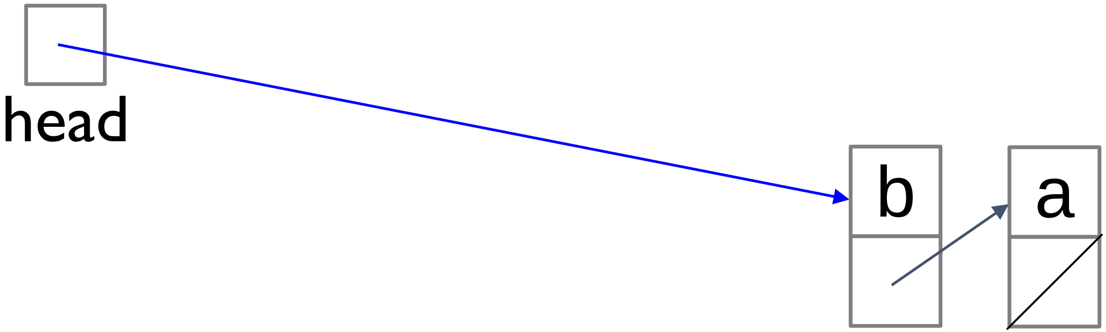
Top() → *c*

Stack Implementation with Linked List



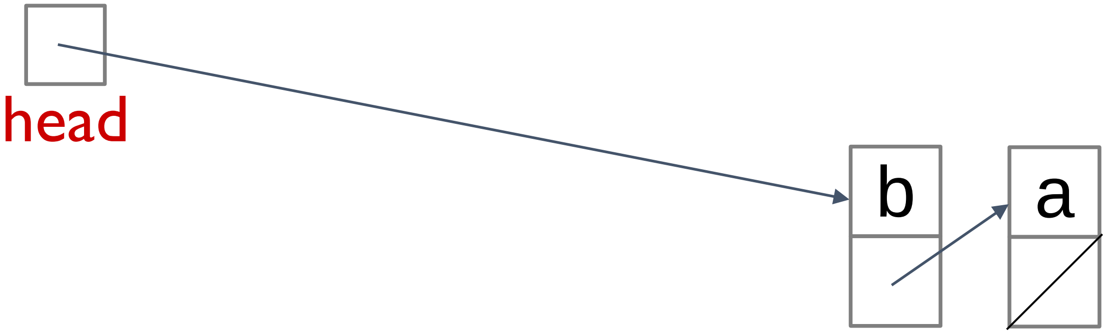
Pop( )

Stack Implementation with Linked List



$Pop() \rightarrow c$

Stack Implementation with Linked List



$IsEmpty() \rightarrow False$

Stack ADT: cost of operations

	Array Impl.	Link. List Impl.
Push(e)	O(1)	O(1)
Top()	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, Top, IsEmpty`*
- Considerations:
  - Linked Lists have storage overhead
  - Arrays need to be resized when full

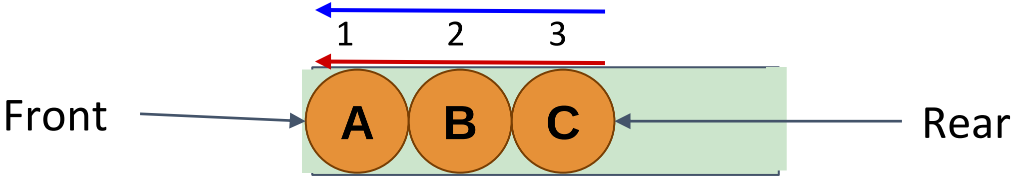
Example 2: Doctor queue

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 supports the following operations:

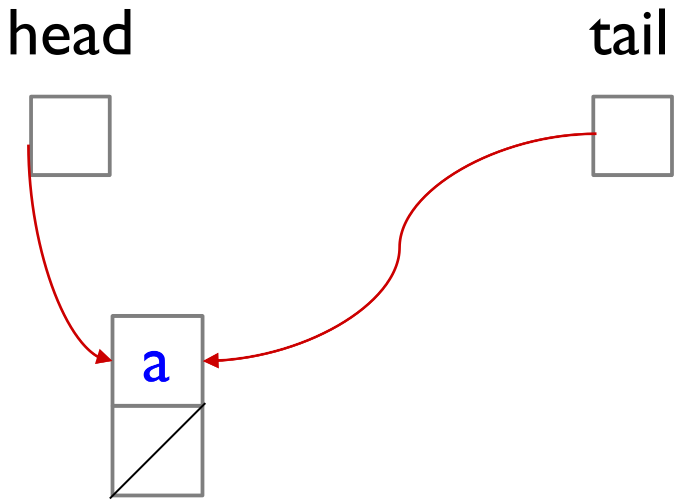
- ➔ ***Enqueue(e)***: adds element *e* to collection
- ➔ ***Dequeue( )***: removes and returns least recently-added key
- ➔ Boolean ***IsEmpty( )***: are there any elements?
- ➔ Boolean ***IsFull()***: is there any space left?

Queue Implementation with Linked List

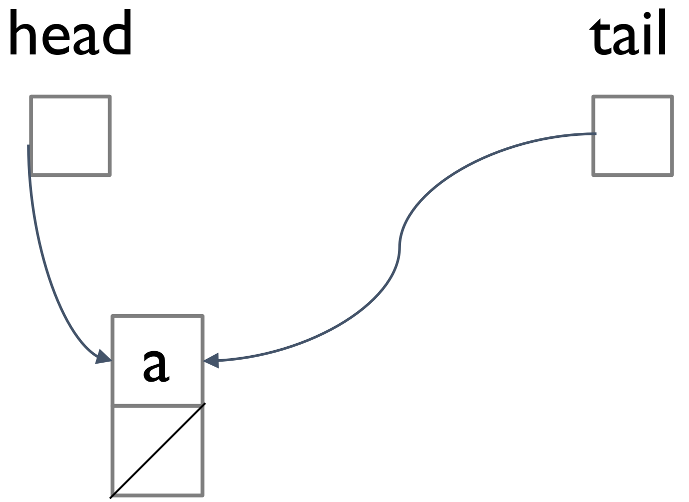
Queue Implementation with Linked List

Enqueue(a)

Queue Implementation with Linked List

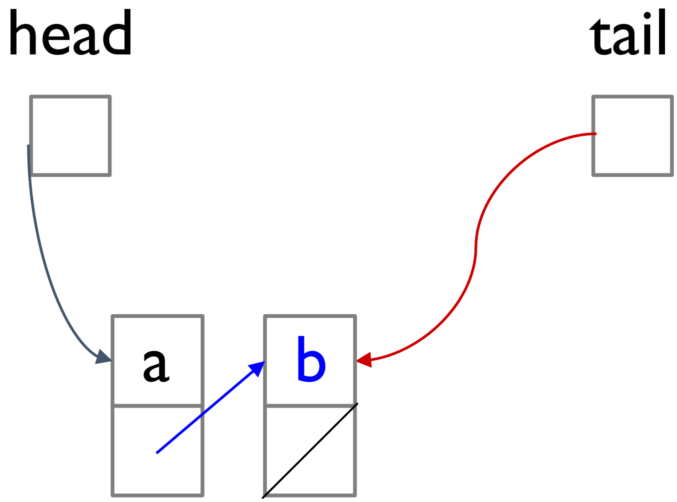


Queue Implementation with Linked List

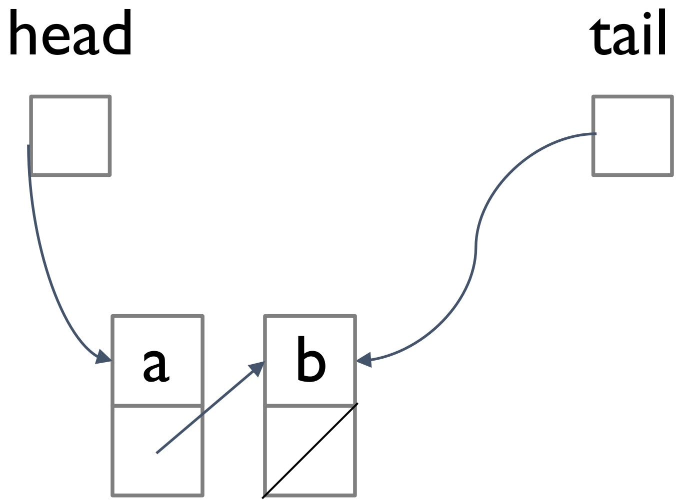


Enqueue(b)

Queue Implementation with Linked List

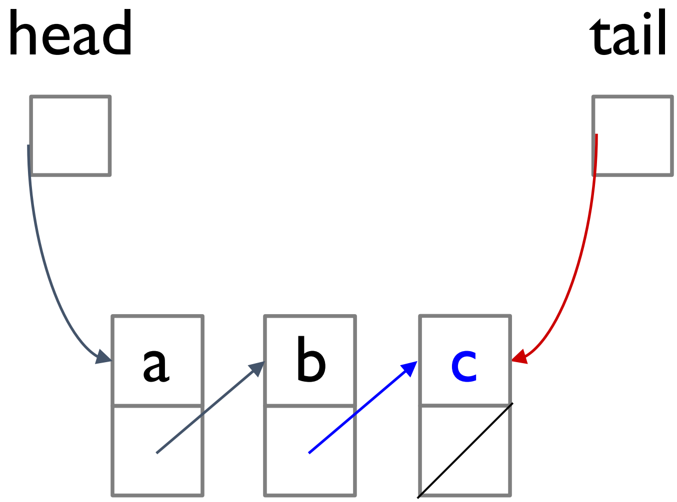


Queue Implementation with Linked List

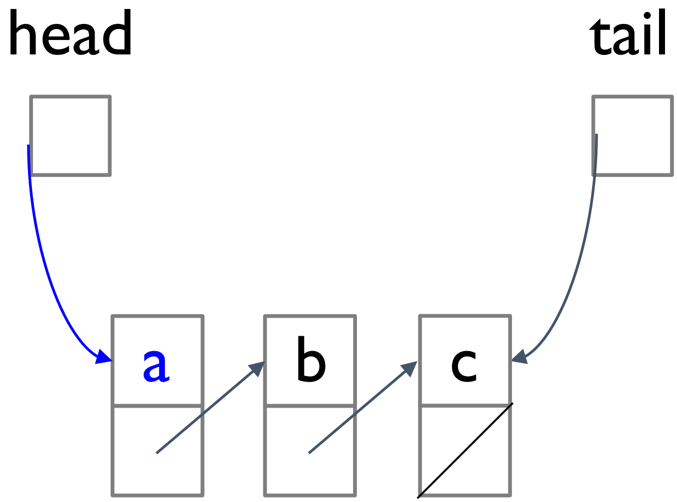


Enqueue(c)

Queue Implementation with Linked List

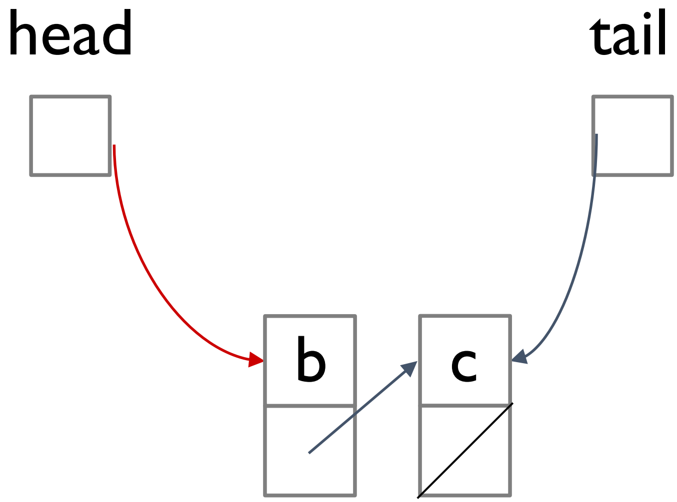


Queue Implementation with Linked List



Dequeue()

Queue Implementation with Linked List



Dequeue() → a

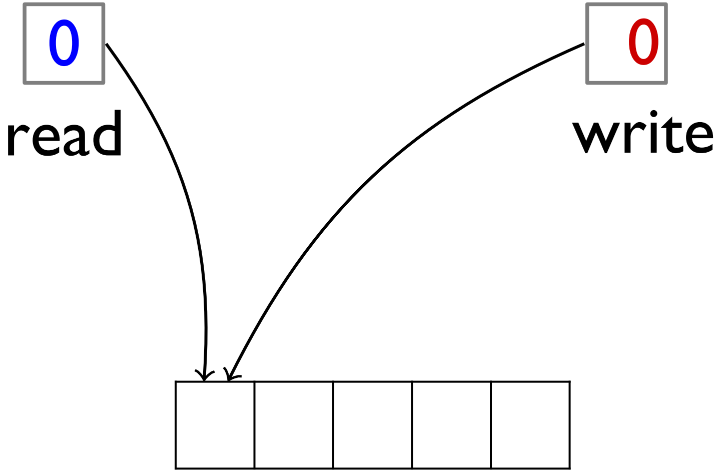
Queue Implementation with Linked List

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

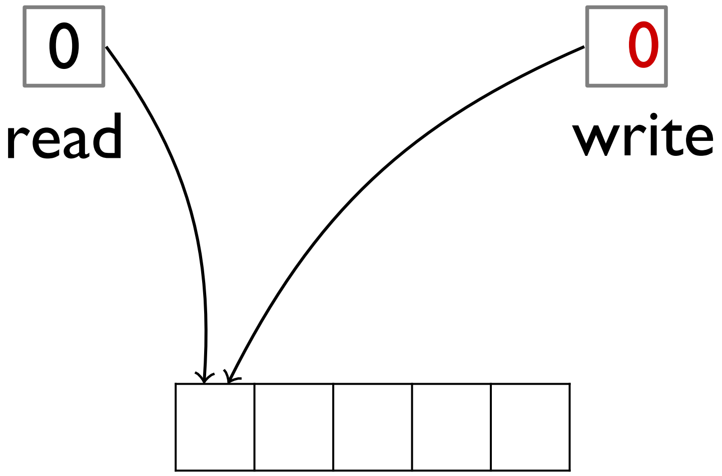
Queue ADT: cost of operations

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

Queue Implementation with Array

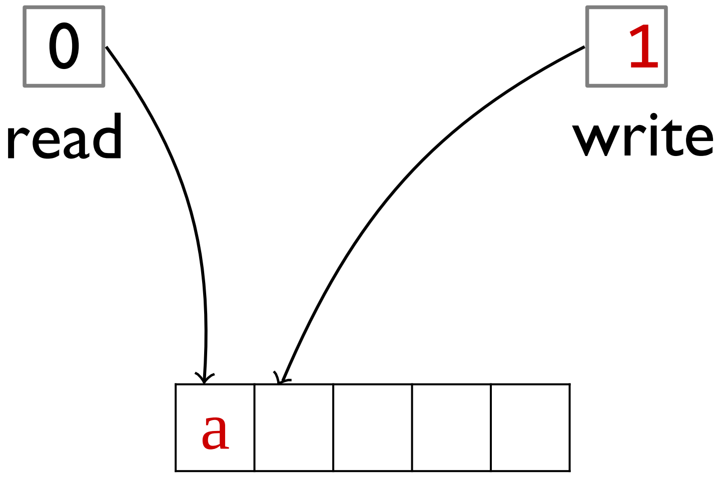


Queue Implementation with Array

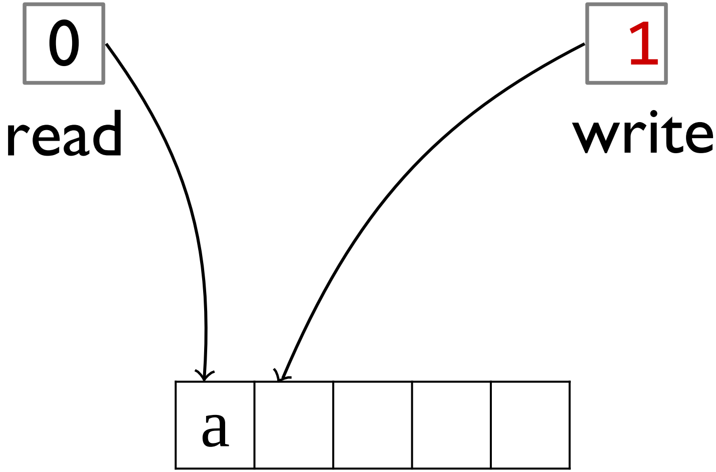


Enqueue(a)

Queue Implementation with Array

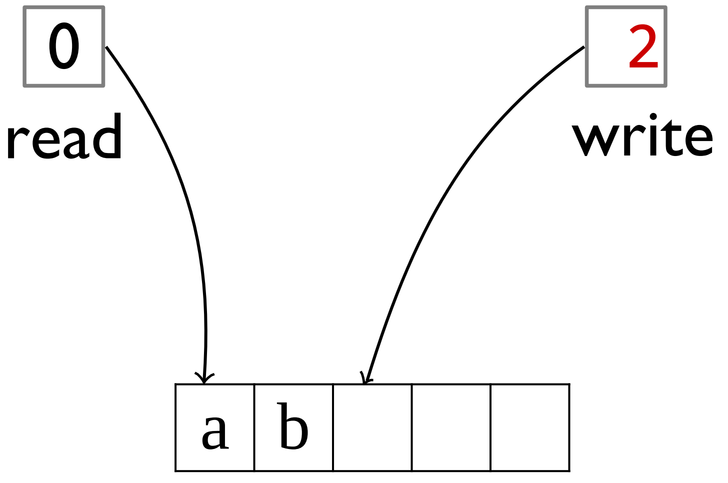


Queue Implementation with Array

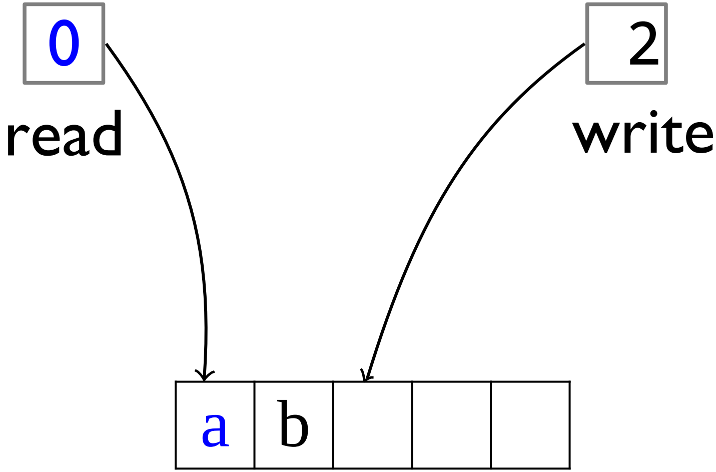


Enqueue(b)

Queue Implementation with Array

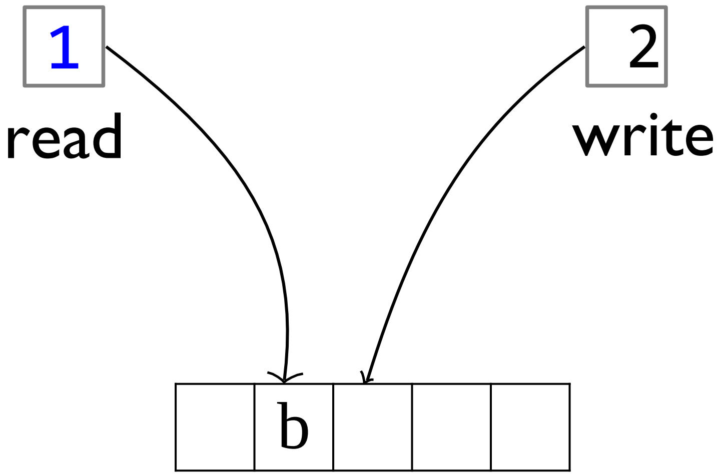


Queue Implementation with Array



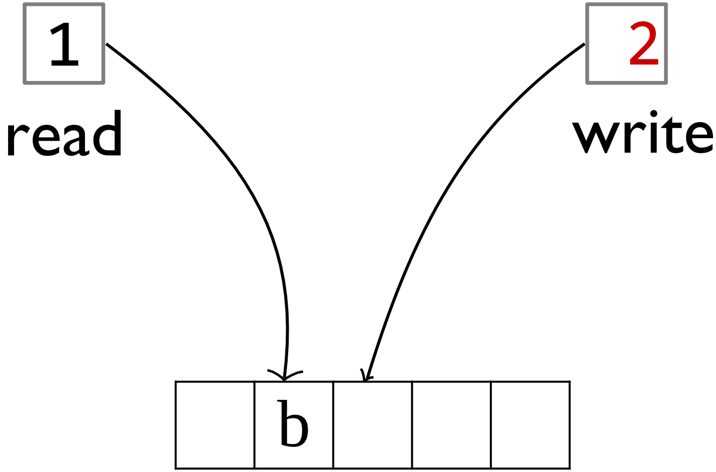
Dequeue()

Queue Implementation with Array



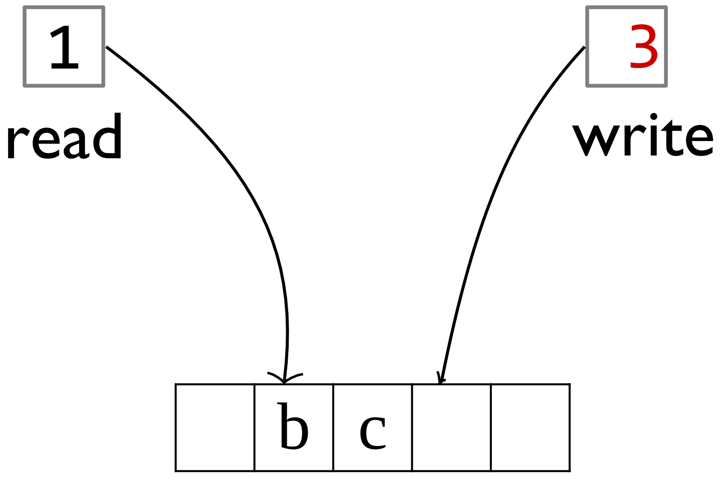
Dequeue() → a

Queue Implementation with Array

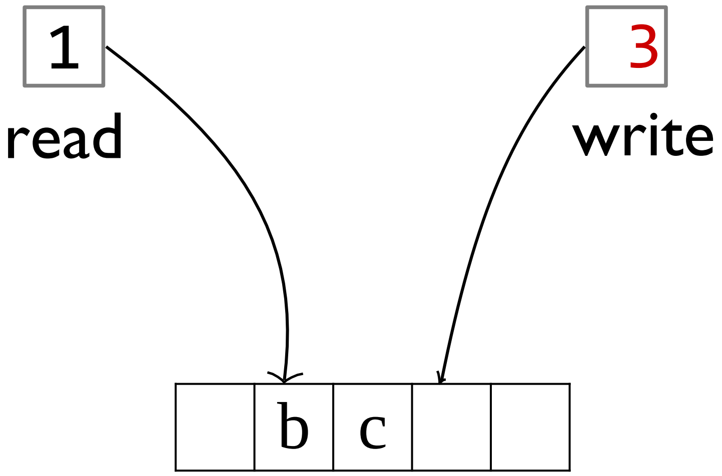


Enqueue(c)

Queue Implementation with Array

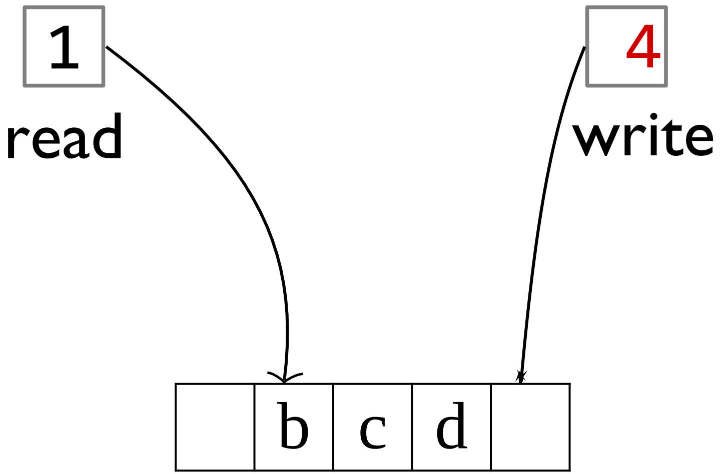


Queue Implementation with Array

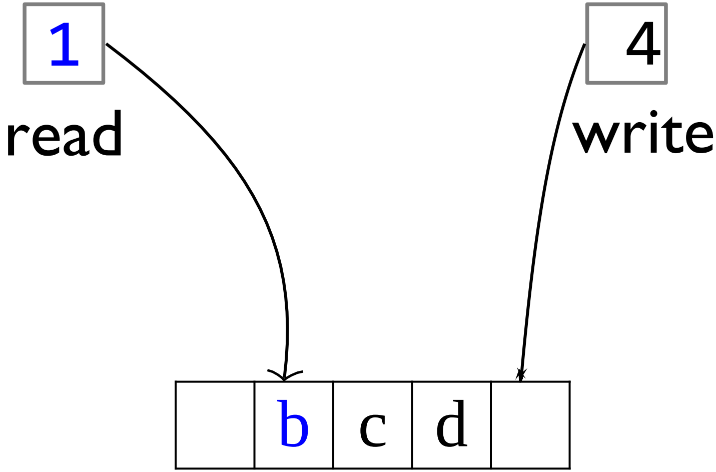


Enqueue(d)

Queue Implementation with Array

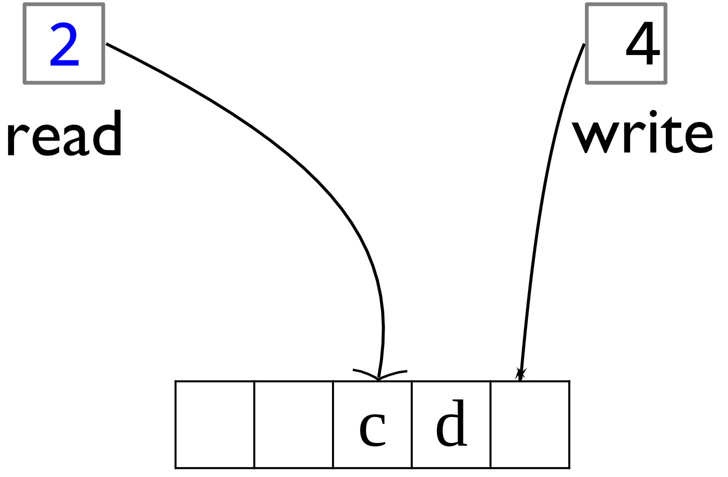


Queue Implementation with Array



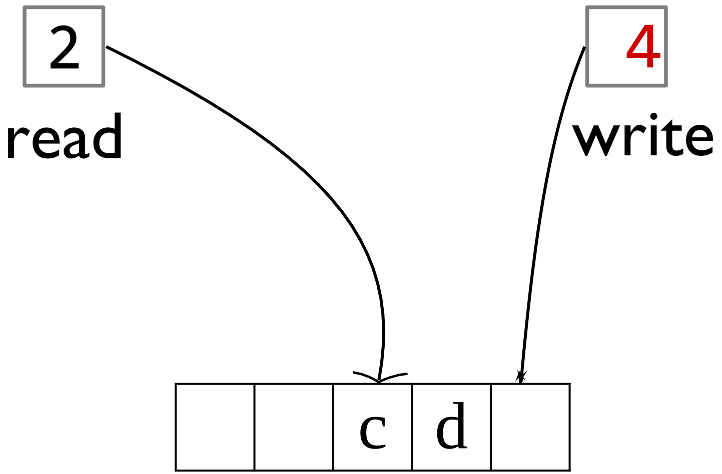
Dequeue()

Queue Implementation with Array



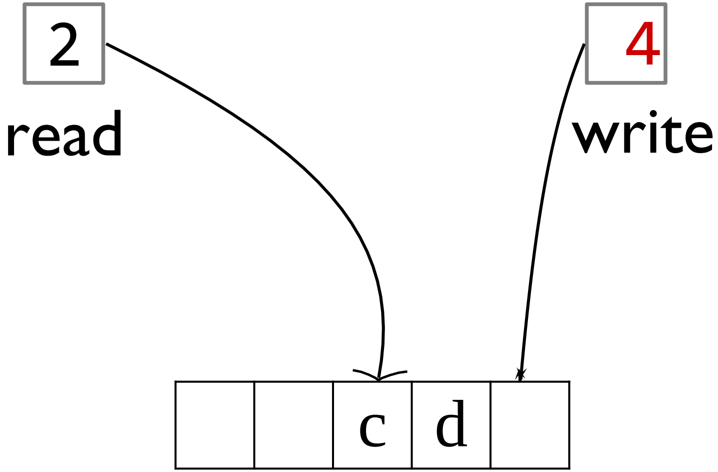
Dequeue() → *b*

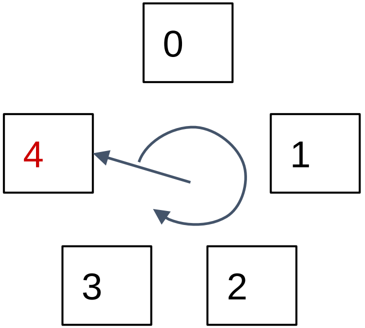
Queue Implementation with Array



Enqueue(e)

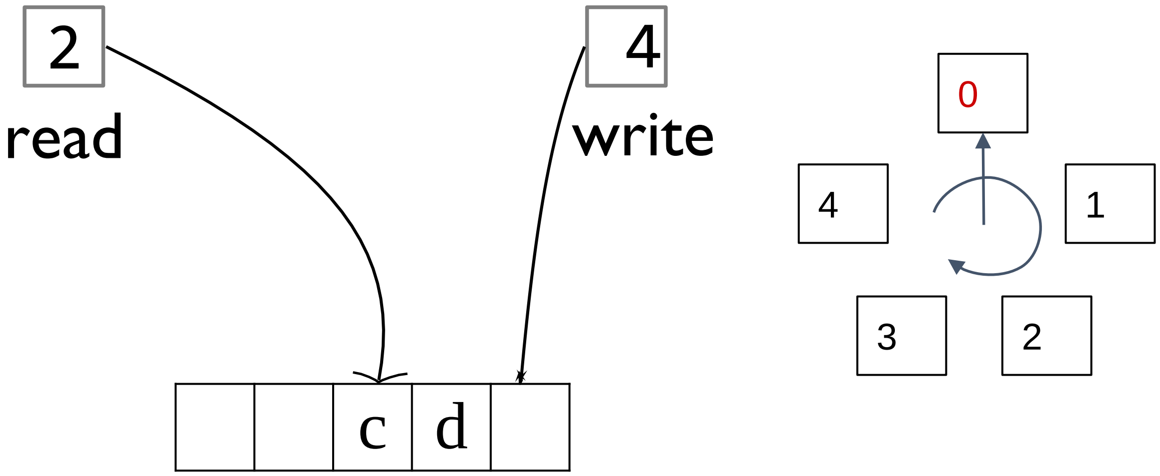
Concept of a Circular Array





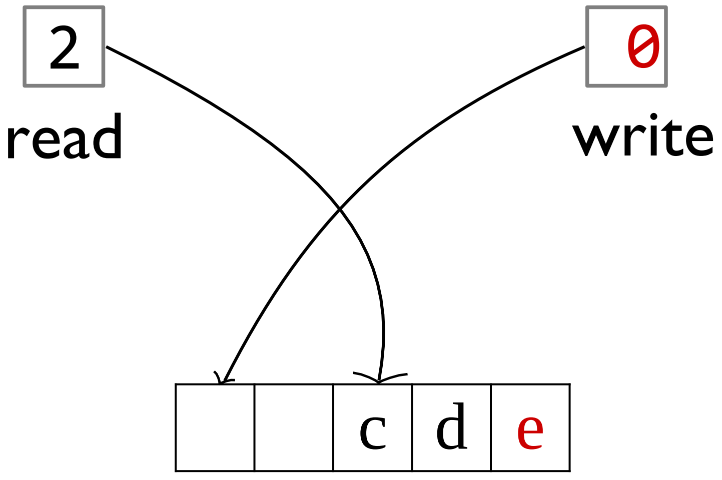
Enqueue(e)

Concept of a Circular Array

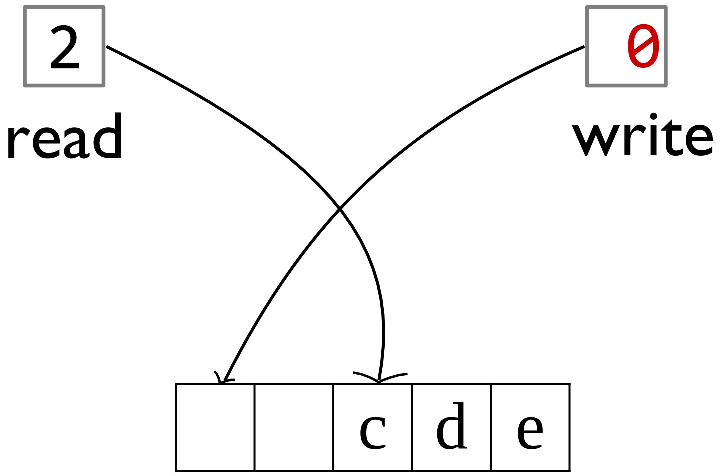


Enqueue(e)

Queue Implementation with Array

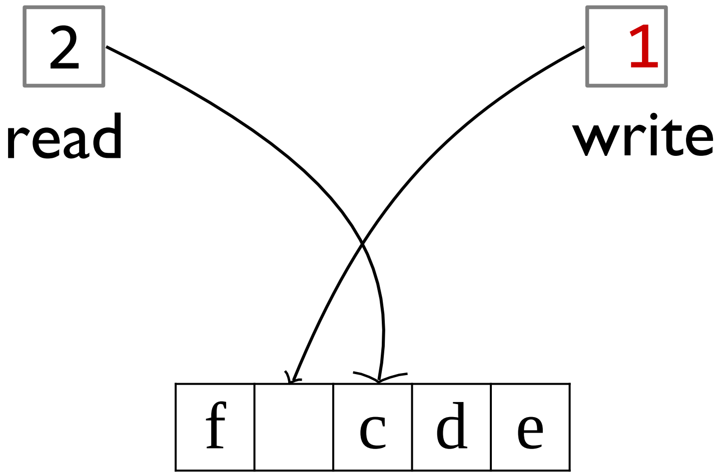


Queue Implementation with Array

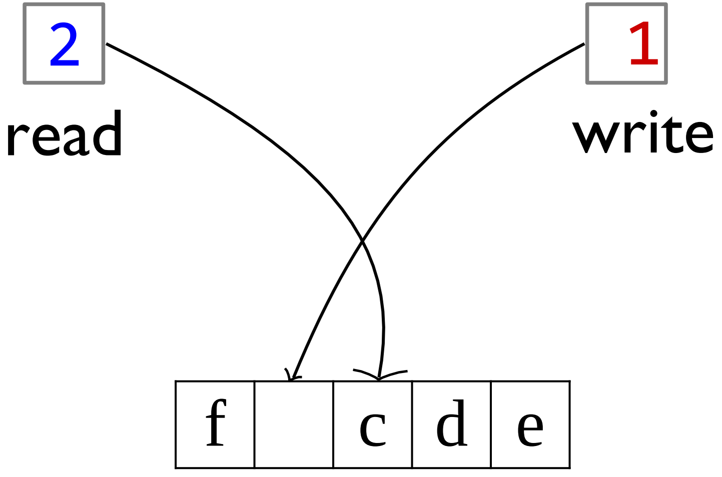


Enqueue(f)

Queue Implementation with Array

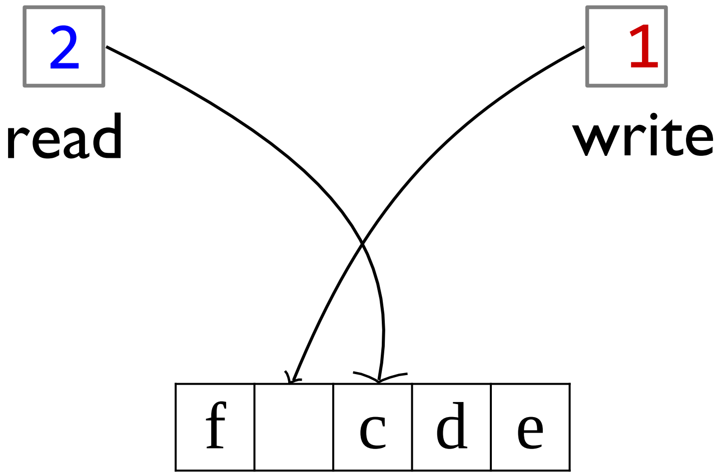


Queue Implementation with Array



Enqueue(g)

Queue Implementation with Array

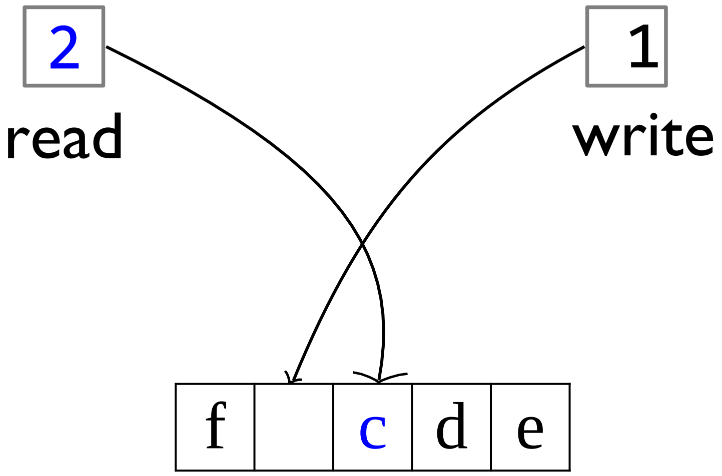


Enqueue(g) → ERROR

Cannot set read = write

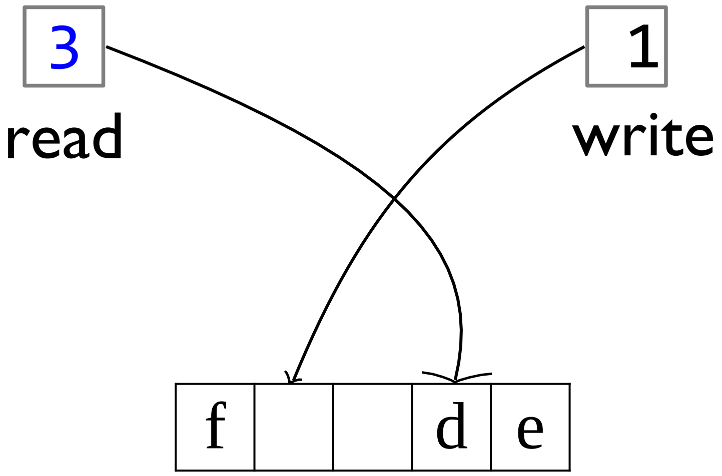
isFull() → True

Queue Implementation with Array



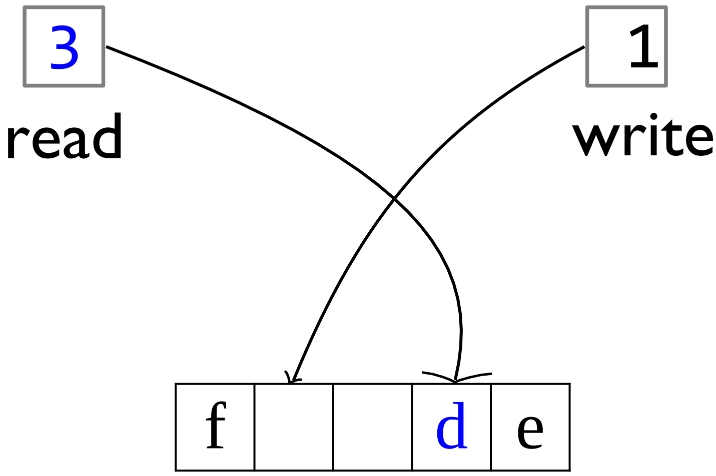
Dequeue()

Queue Implementation with Array



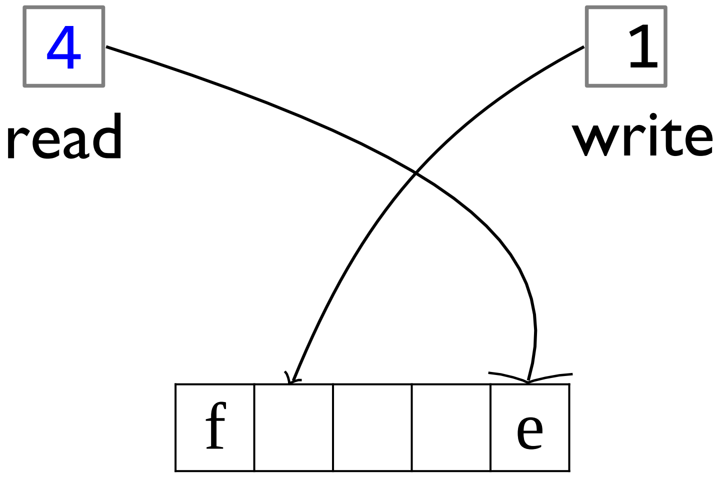
Dequeue() → *c*

Queue Implementation with Array



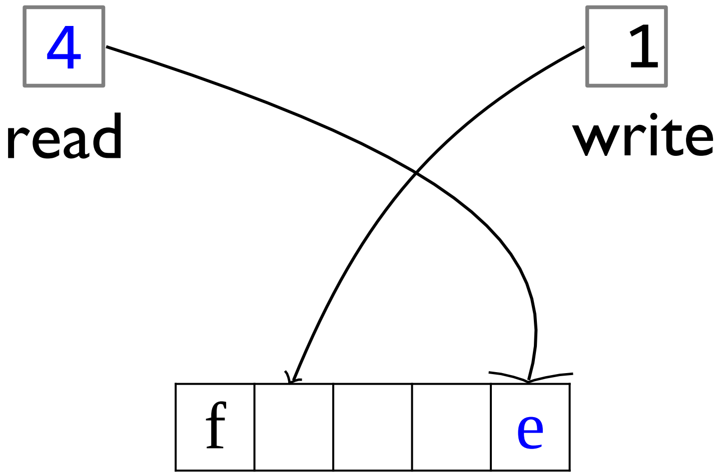
Dequeue()

Queue Implementation with Array



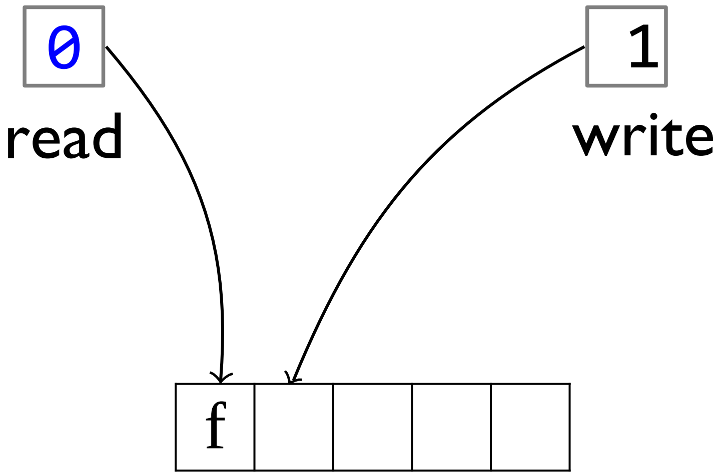
Dequeue() → d

Queue Implementation with Array



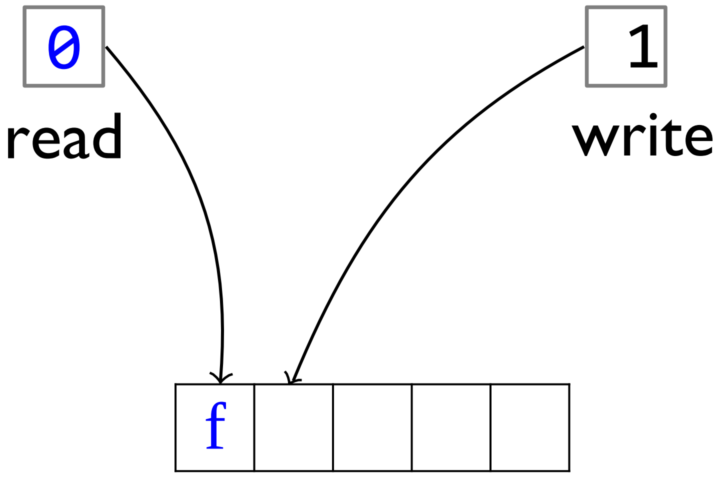
Dequeue()

Queue Implementation with Array



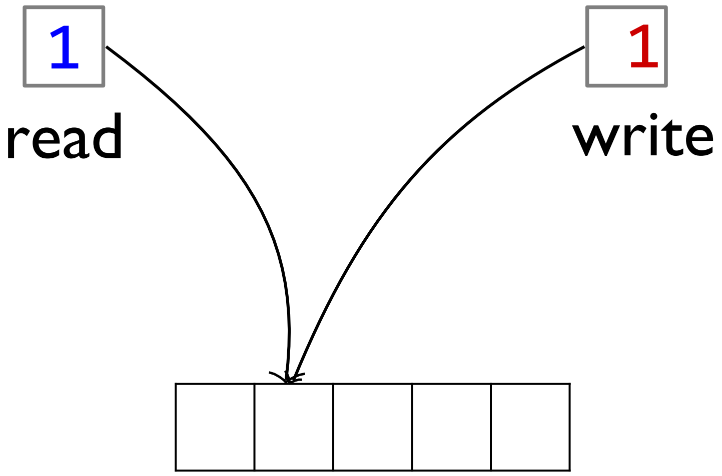
Dequeue() → *e*

Queue Implementation with Array



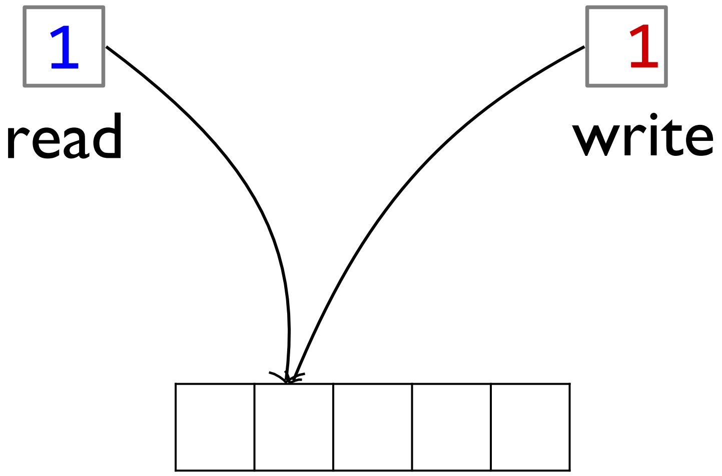
Dequeue()

Queue Implementation with Array



Dequeue() → *f*

Queue Implementation with Array



IsEmpty() → True

Queue Implementation with 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

Queue ADT: cost of operations

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

Queue: Summary

- ➔ **ADT Queue** 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

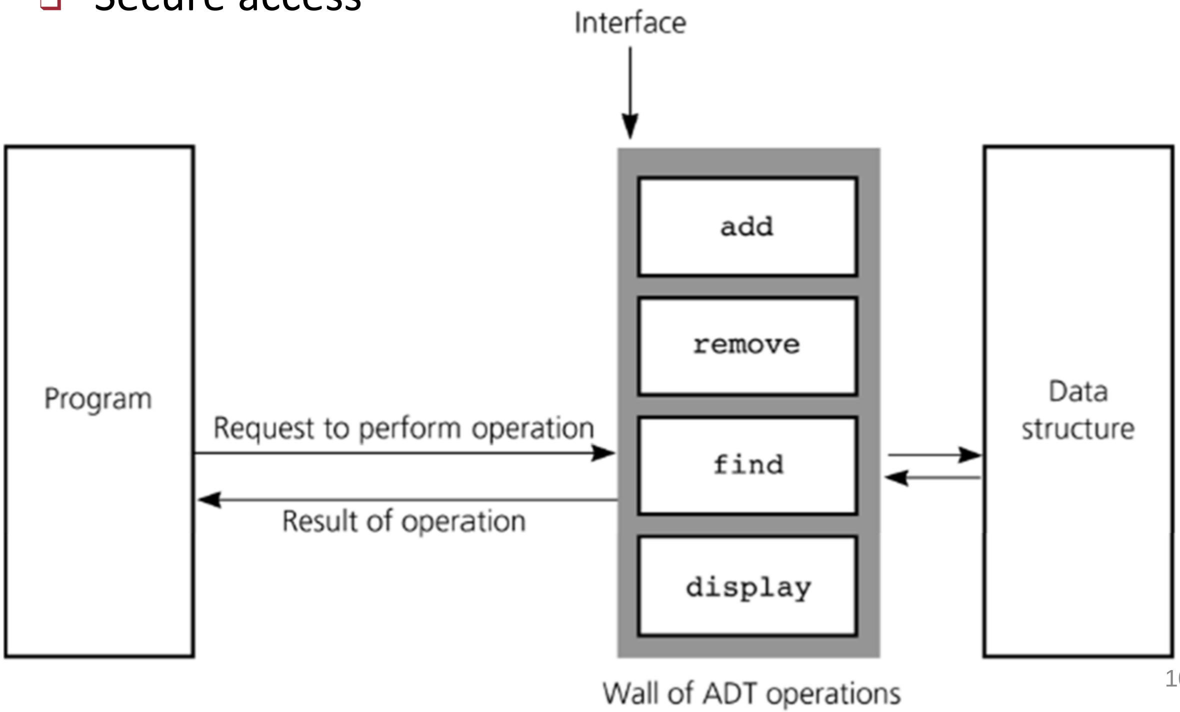
Hide implementation details from users of ADT

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 do **not** affect the users of ADT

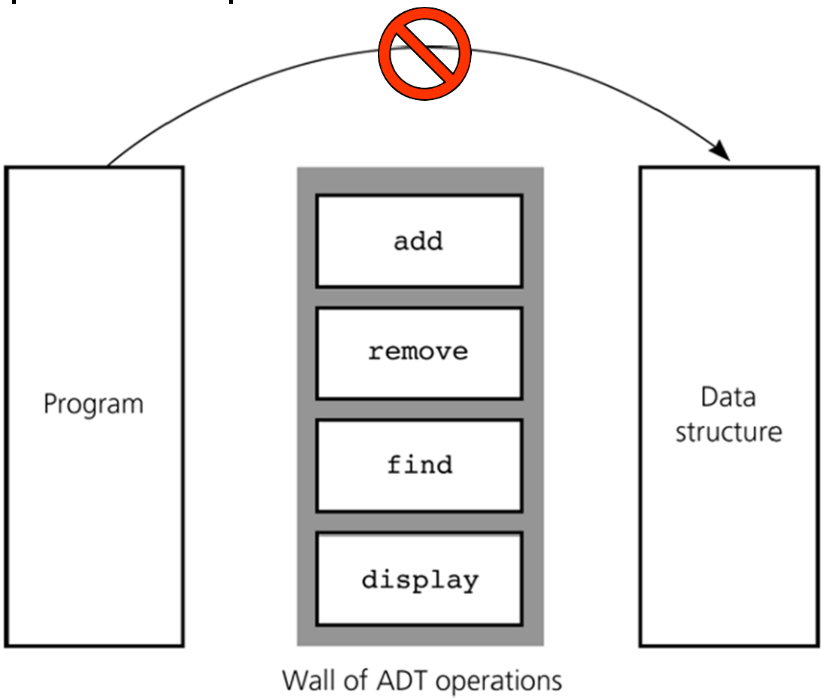
A Wall of ADT

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

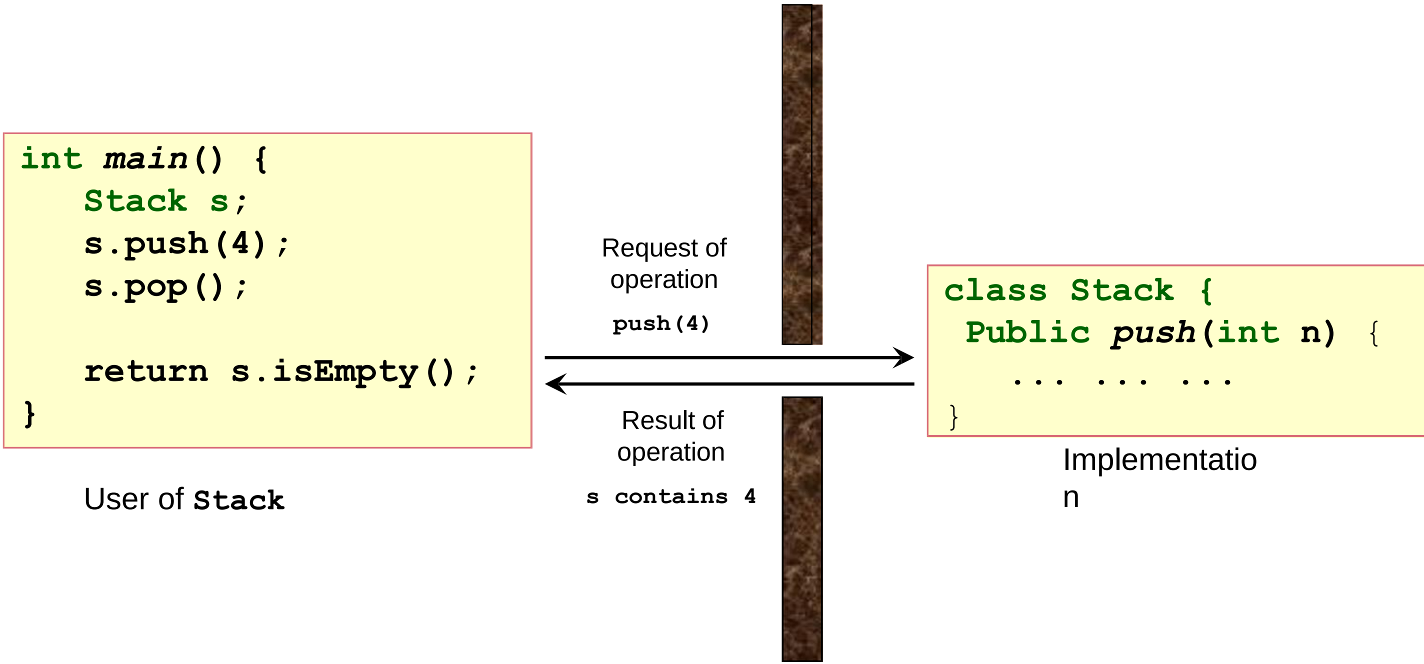


Violating the abstraction

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



Specification as slit in the wall



- User only depends on specifications:
  - Function name, parameter types, and return type

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
- Localise rather than globalise changes
- Help manage software complexity