Lecture 10

Stack and Queue ADTs

Implementing
using arrays and linked lists

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

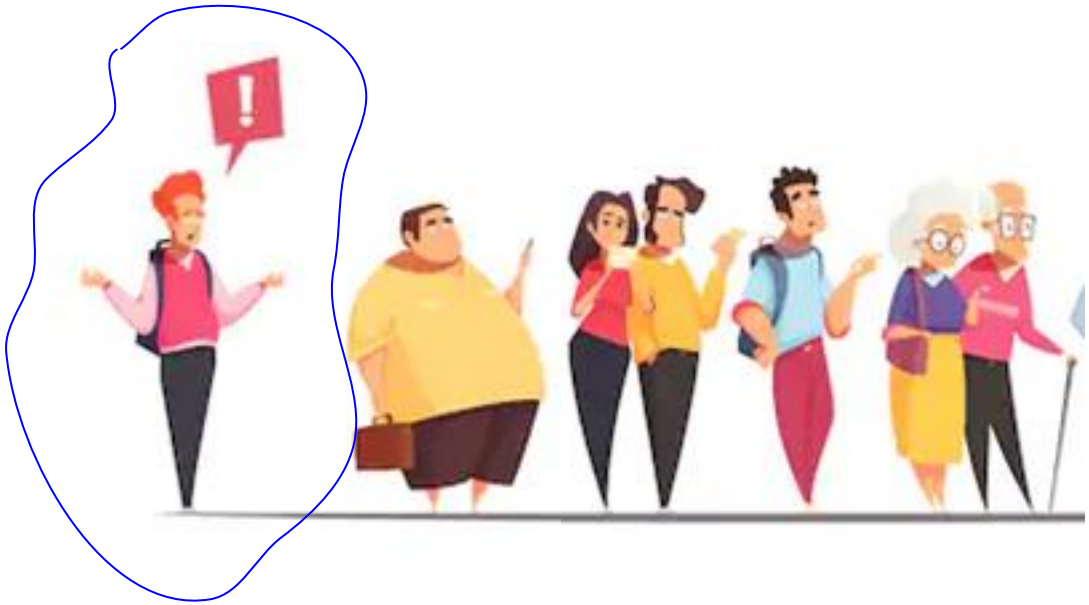
Modeling HR roster

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

Modeling HR roster

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



Modeling 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)

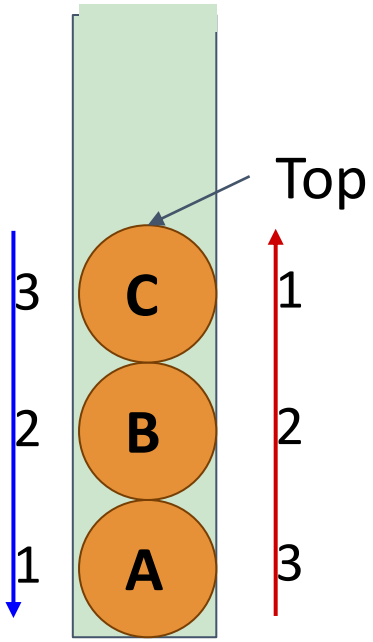
Modeling HR roster

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

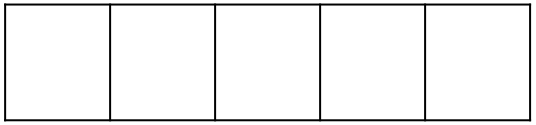
Specification and **implementation** have to be disjoint:

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

Stack Implementation with Array

size: 0

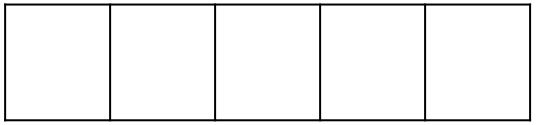
capacity: 5



Stack Implementation with Array

size: 0

capacity: 5

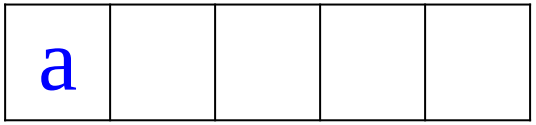


push(a)

Stack Implementation with Array

size: 1

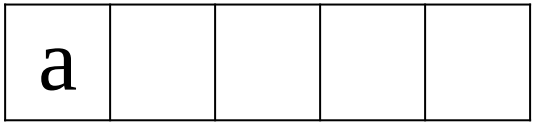
capacity: 5



Stack Implementation with Array

size: 1

capacity: 5

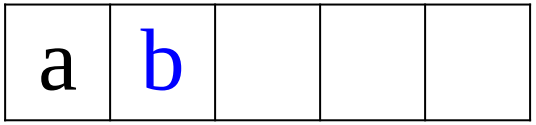


push(b)

Stack Implementation with Array

size: 2

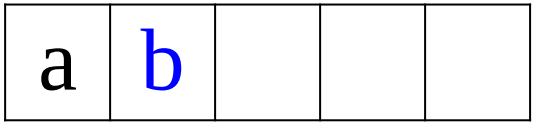
capacity: 5



Stack Implementation with Array

size: 2

capacity: 5

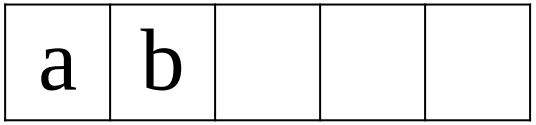


peek() → *b*

Stack Implementation with Array

size: 2

capacity: 5

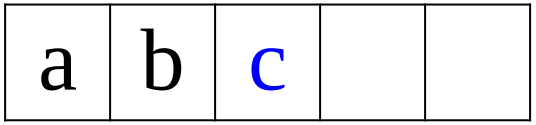


push(c)

Stack Implementation with Array

size: 3

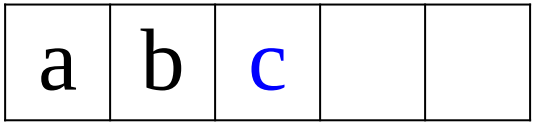
capacity: 5



Stack Implementation with Array

size: 3

capacity: 5

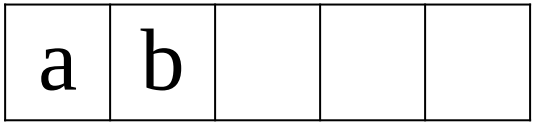


pop()

Stack Implementation with Array

size: 2

capacity: 5

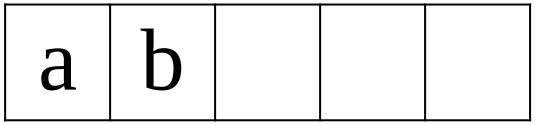


pop() → *c*

Stack Implementation with Array

size: 2

capacity: 5

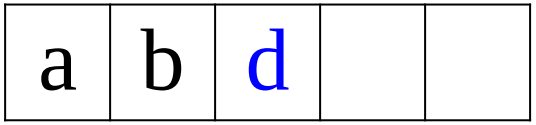


push(d)

Stack Implementation with Array

size: 3

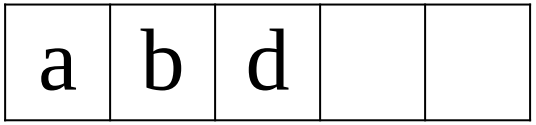
capacity: 5



Stack Implementation with Array

size: 3

capacity: 5

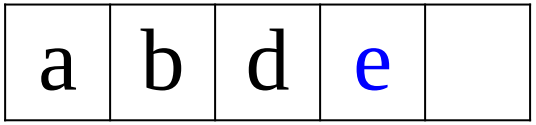


push(e)

Stack Implementation with Array

size: 4

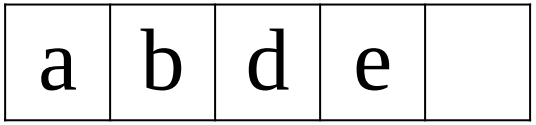
capacity: 5



Stack Implementation with Array

size: 4

capacity: 5

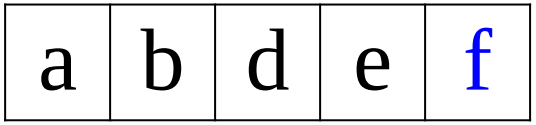


push(f)

Stack Implementation with Array

size: 5

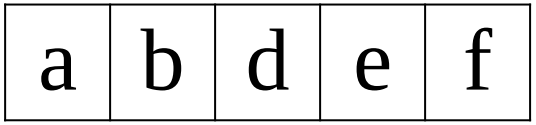
capacity: 5



Stack Implementation with Array

size: 5

capacity: 5

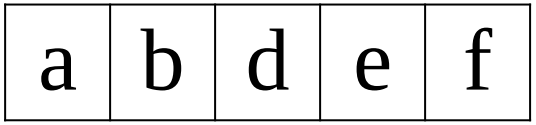


push(g)

Stack Implementation with Array

size: 5

capacity: 5



What can be done to prevent this from happening?

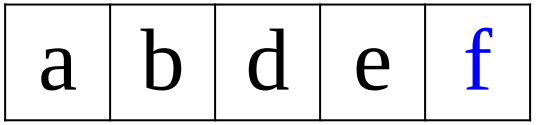
ERROR

isFull() → True

Stack Implementation with Array

size: 5

capacity: 5

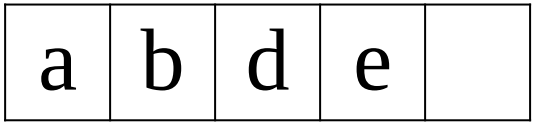


pop()

Stack Implementation with Array

size: 4

capacity: 5

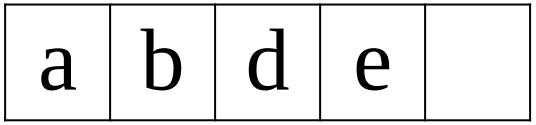


isEmpty → *False*

Stack Implementation with Array

size: 4

capacity: 5

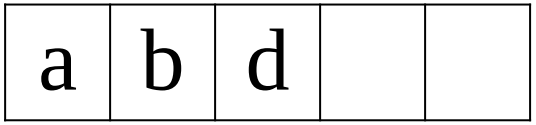


pop()

Stack Implementation with Array

size: 3

capacity: 5

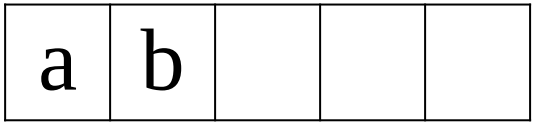


pop()

Stack Implementation with Array

size: 2

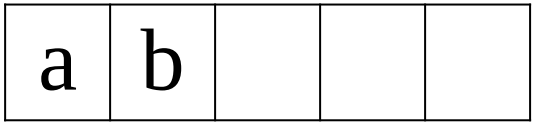
capacity: 5



Stack Implementation with Array

size: 2

capacity: 5

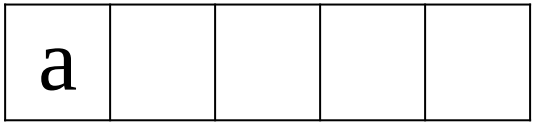


pop()

Stack Implementation with Array

size: 1

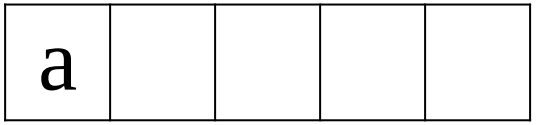
capacity: 5



Stack Implementation with Array

size: 1

capacity: 5

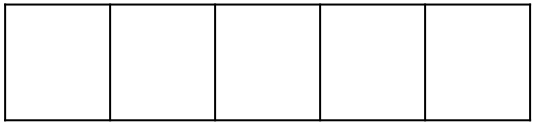


pop()

Stack Implementation with Array

size: 0

capacity: 5

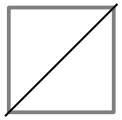


isEmpty() → *True*

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)	

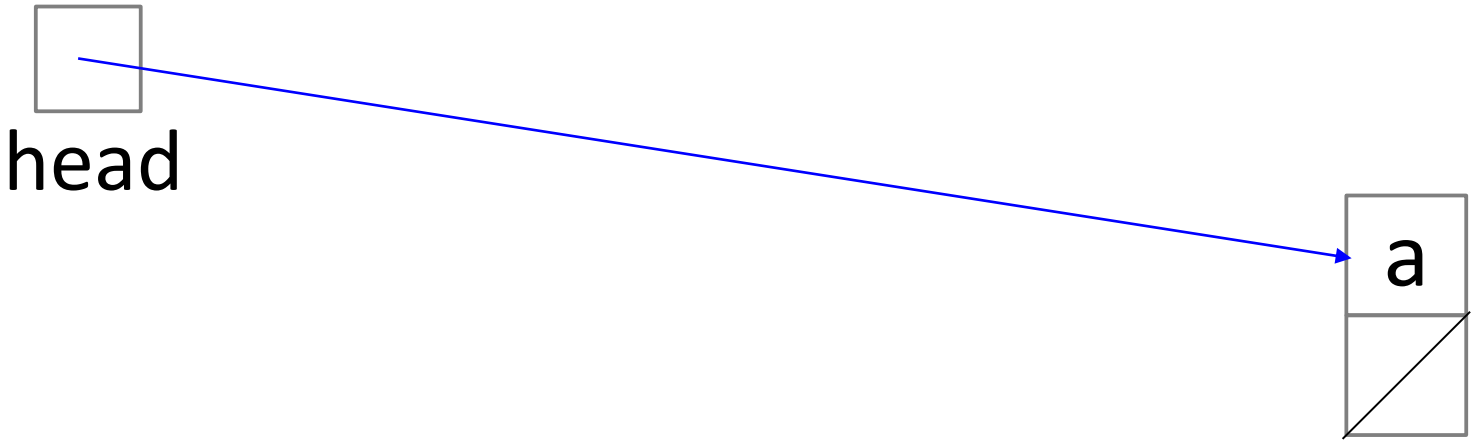
Stack Implementation with Linked List



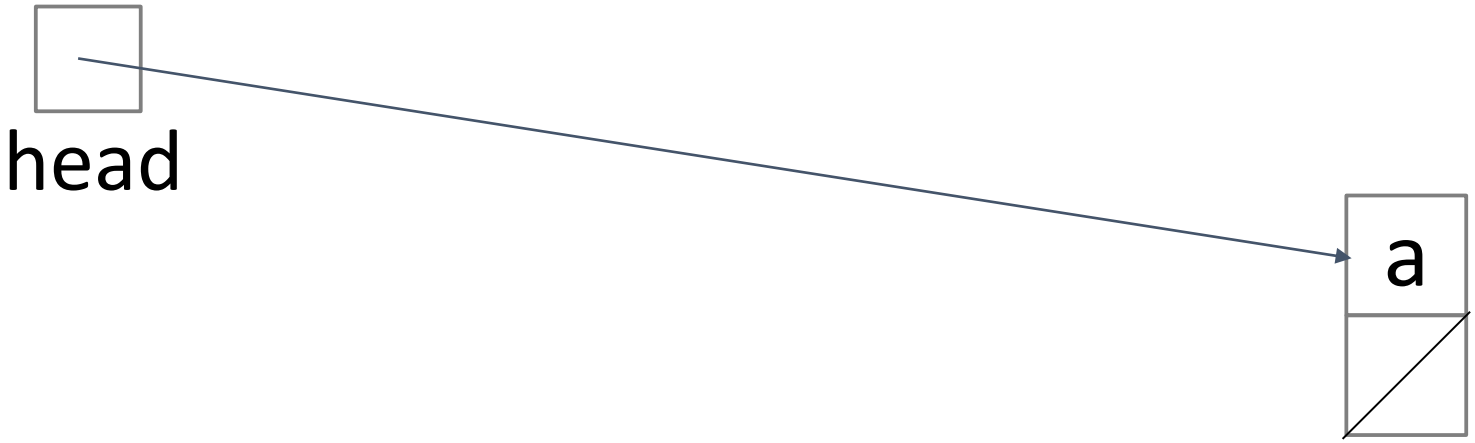
head

push(a)

Stack Implementation with Linked List

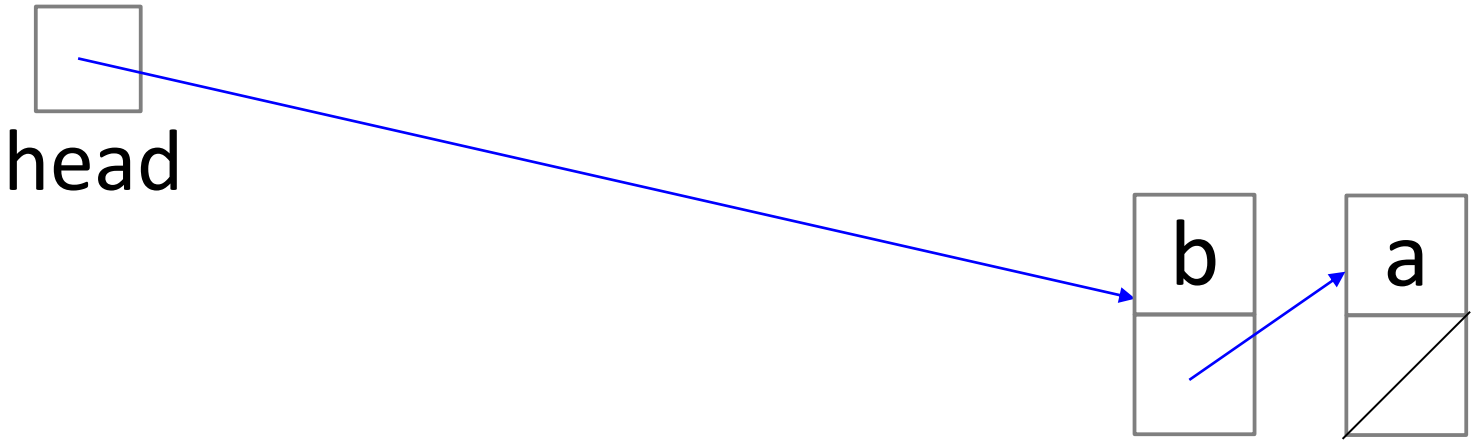


Stack Implementation with Linked List

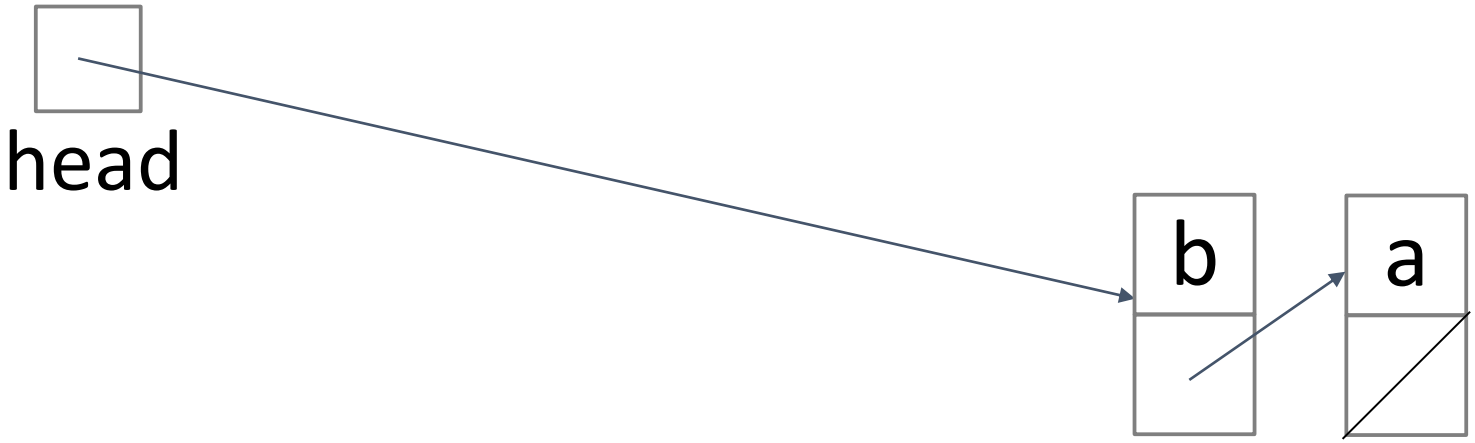


push(b)

Stack Implementation with Linked List

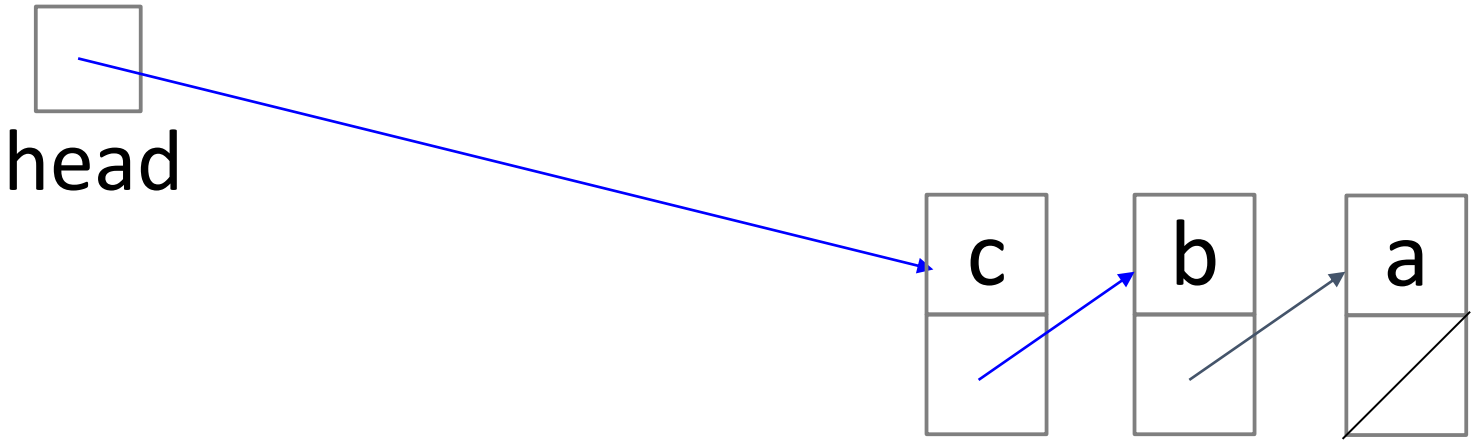


Stack Implementation with Linked List

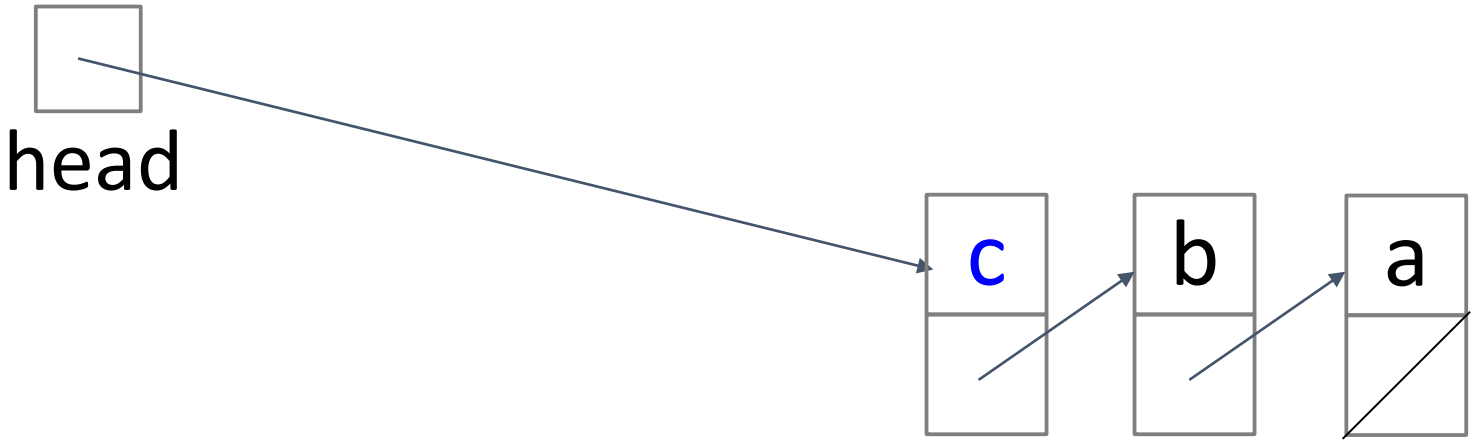


push(c)

Stack Implementation with Linked List

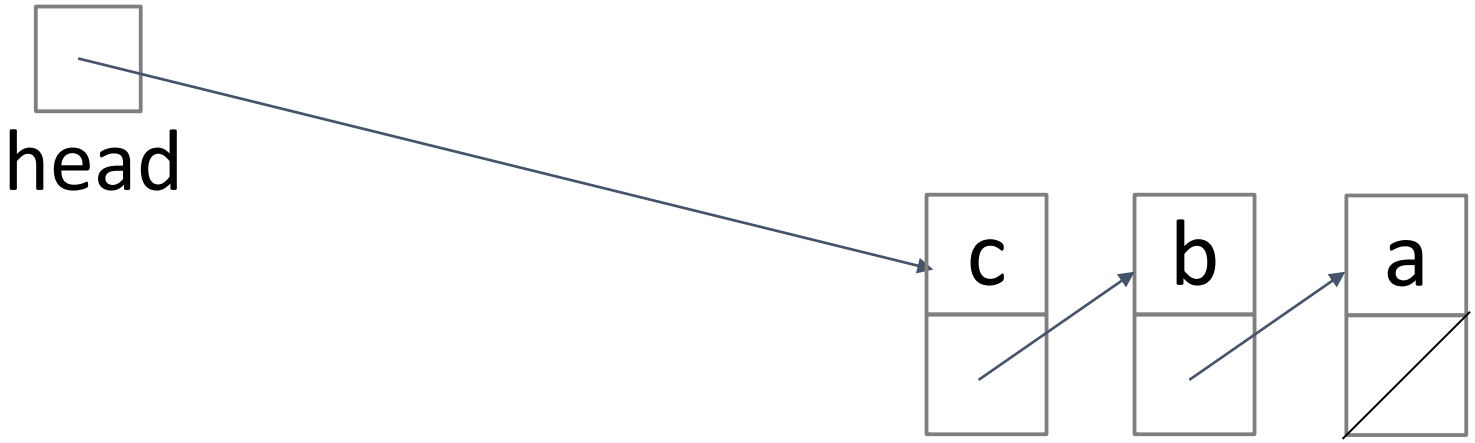


Stack Implementation with Linked List



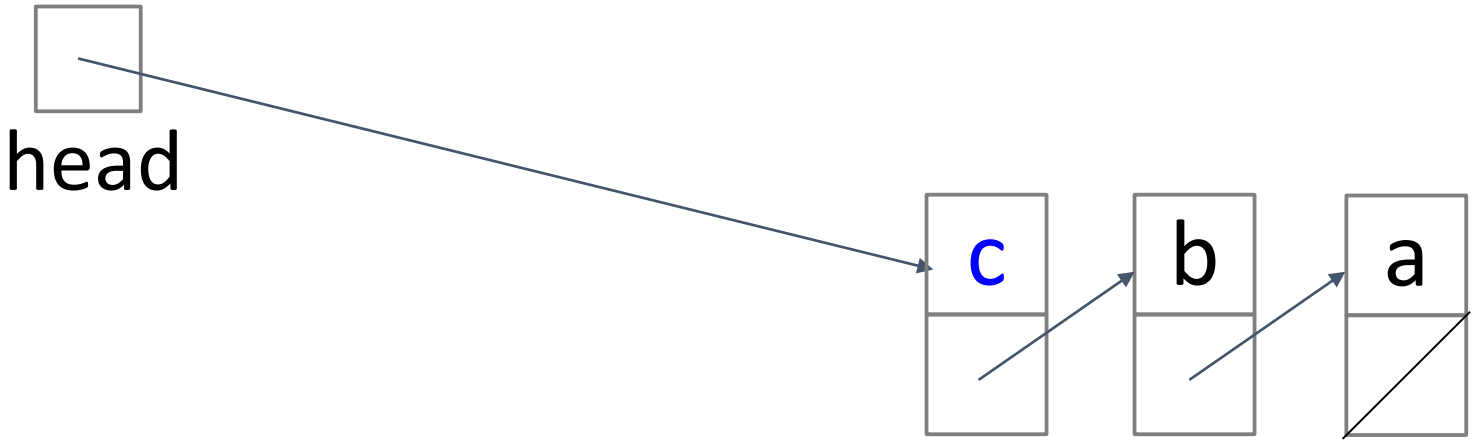
peek()

Stack Implementation with Linked List



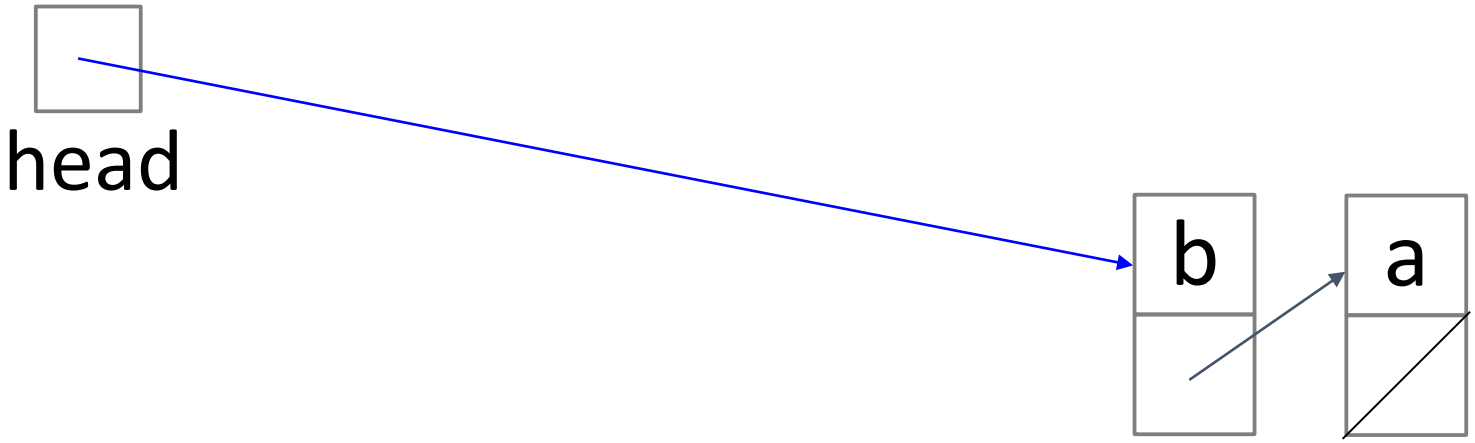
peek() → *c*

Stack Implementation with Linked List



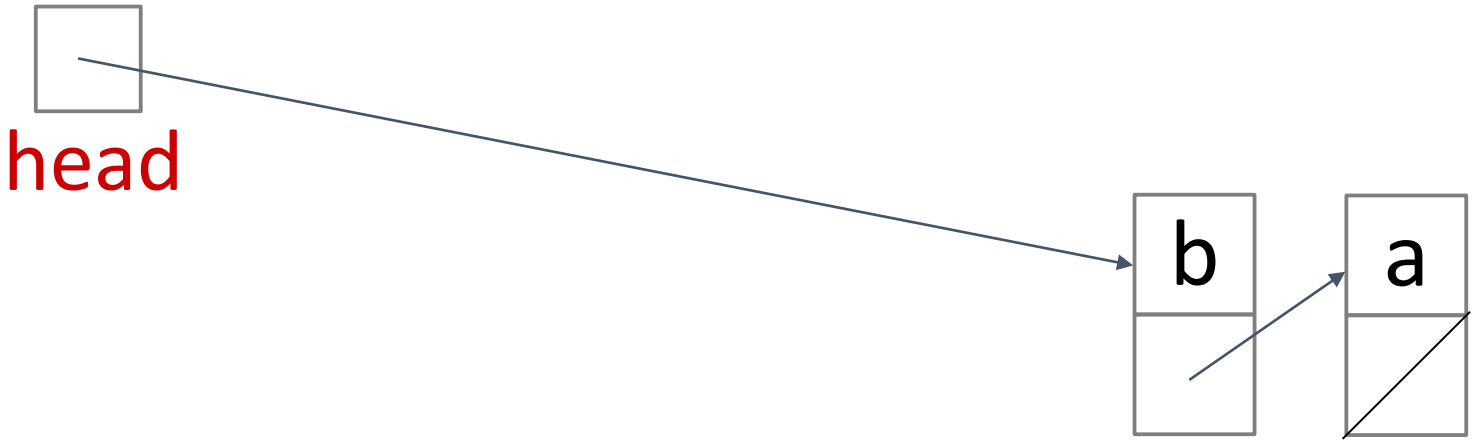
pop()

Stack Implementation with Linked List



pop( ) → *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

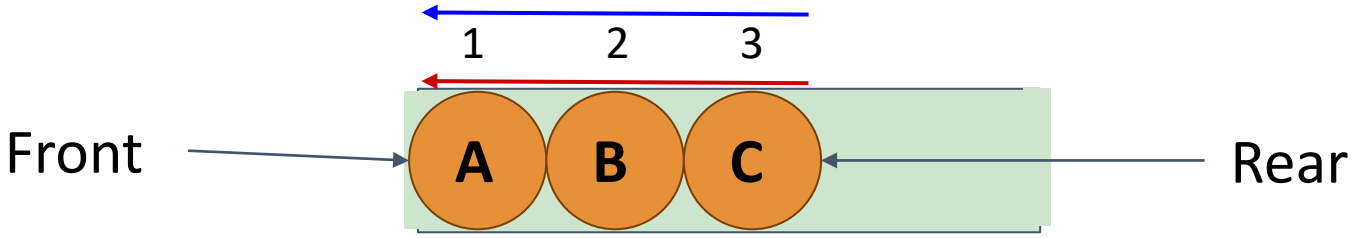
Recap: Abstraction of the Doctor Queue

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 end of 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)



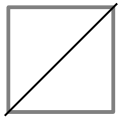
Specification

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

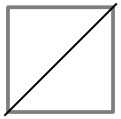
- ***enqueue(e)***: adds element *e* to collection
- ***getFront()***: returns least recently-added (the oldest) key
- ***dequeue()***: removes and returns least recently-added key
- Boolean ***isEmpty()***: are there any elements?
- Boolean ***isFull()***: is there any space left?

Queue Implementation with Linked List

head

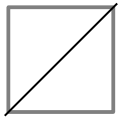


tail

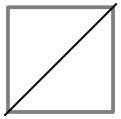


Queue Implementation with Linked List

head

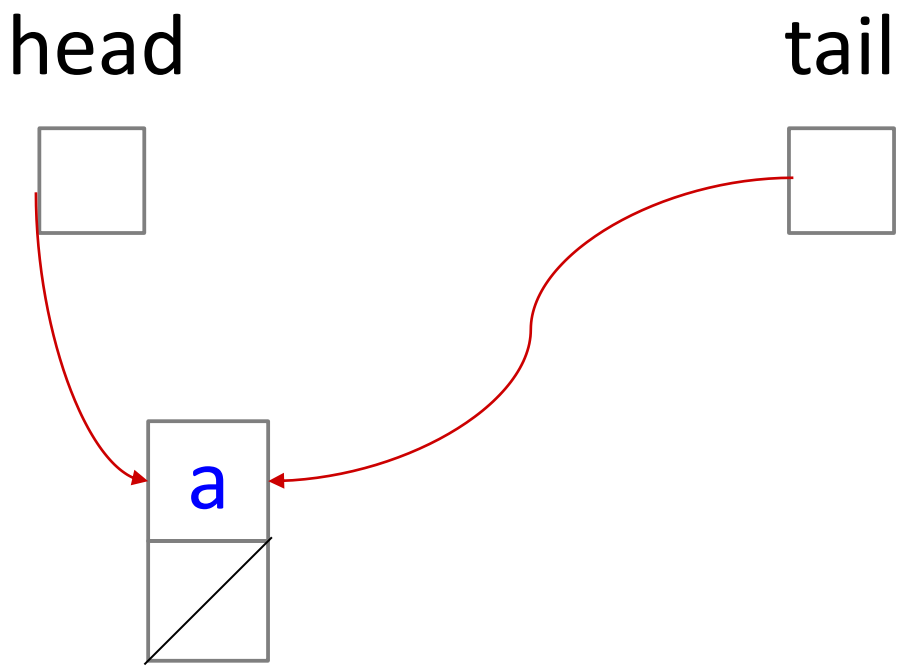


tail

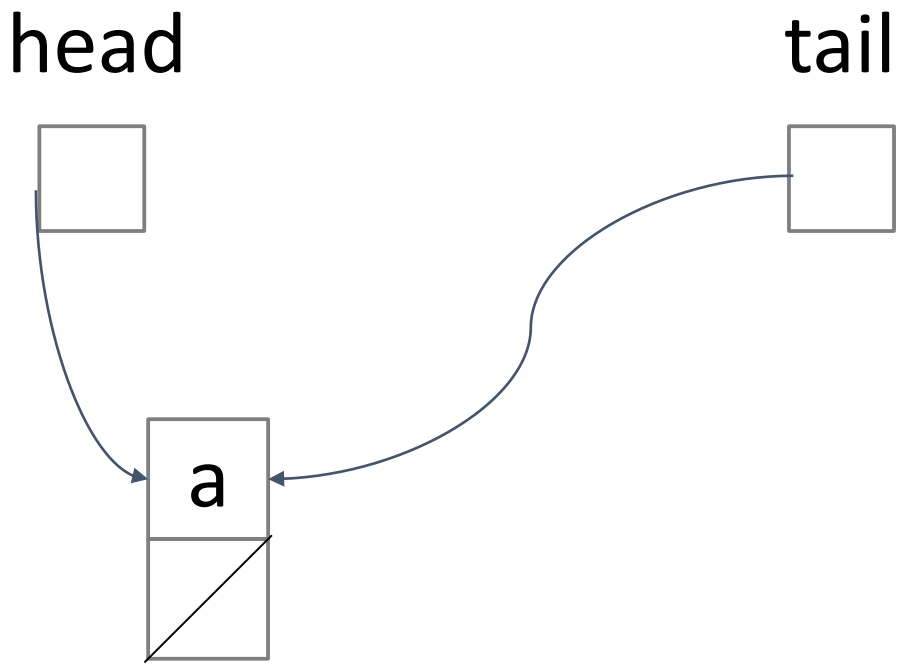


enqueue(a)

Queue Implementation with Linked List

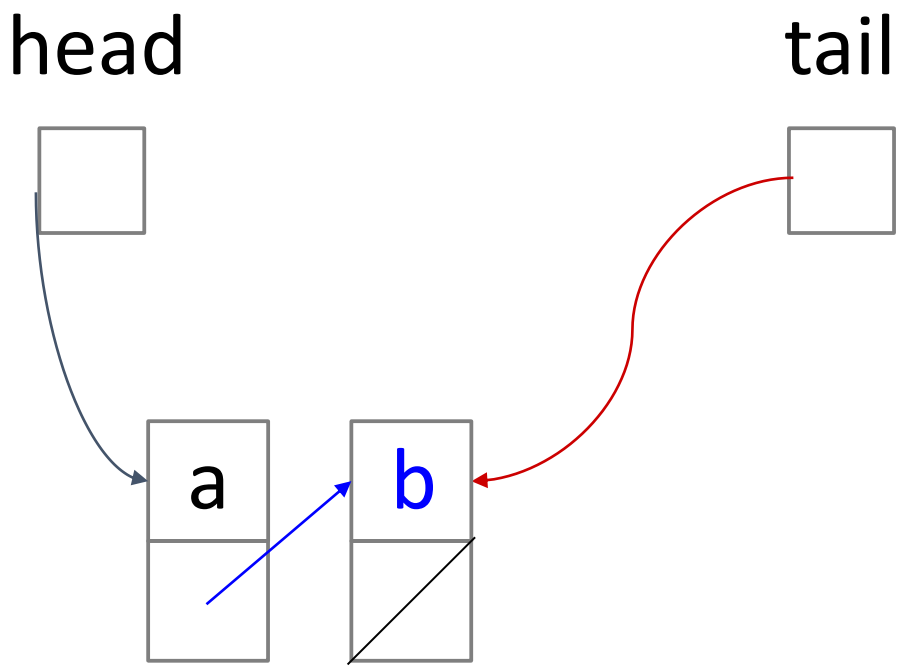


Queue Implementation with Linked List

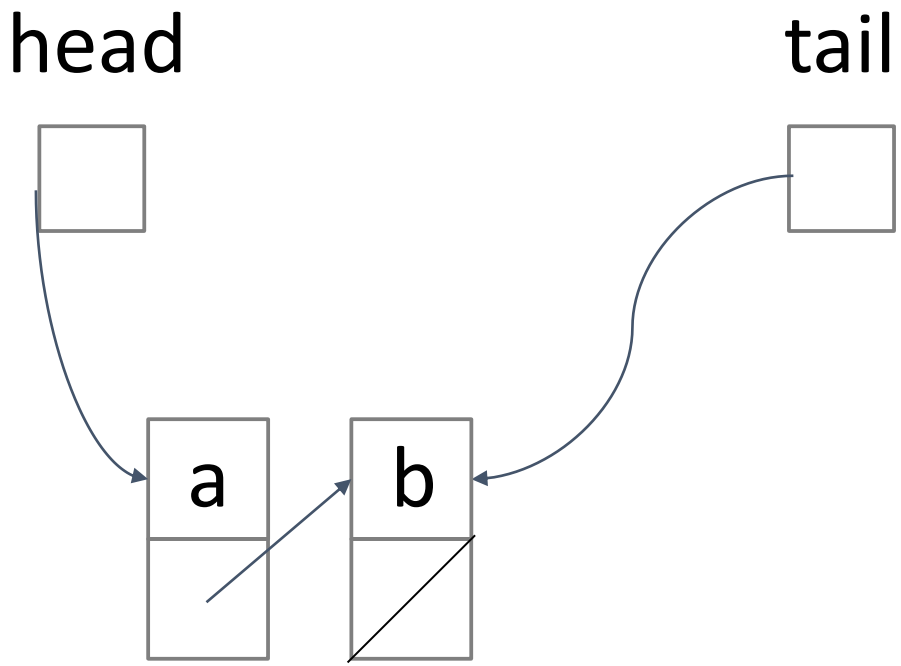


enqueue(b)

Queue Implementation with Linked List

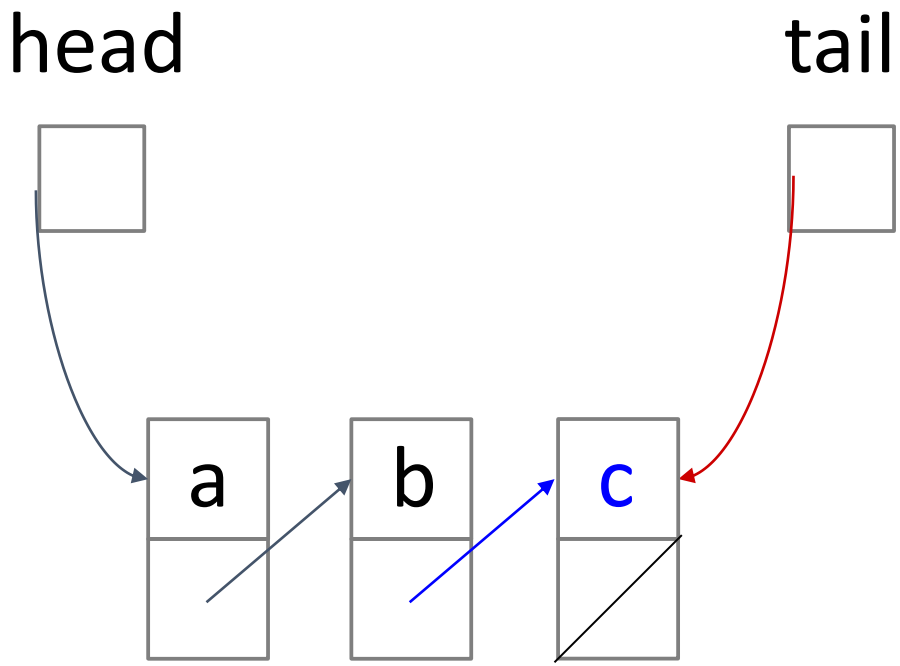


Queue Implementation with Linked List

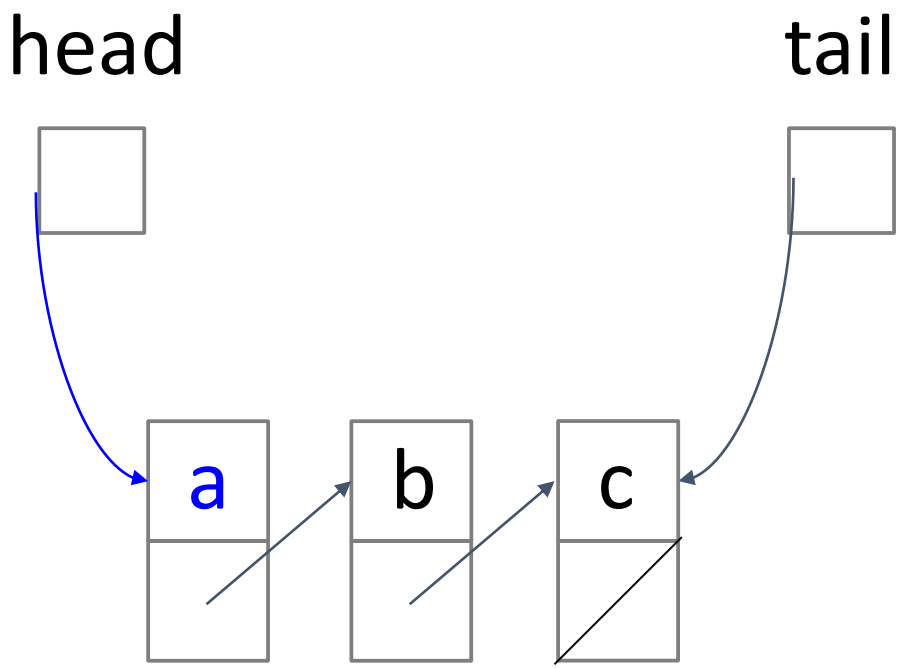


enqueue(c)

Queue Implementation with Linked List

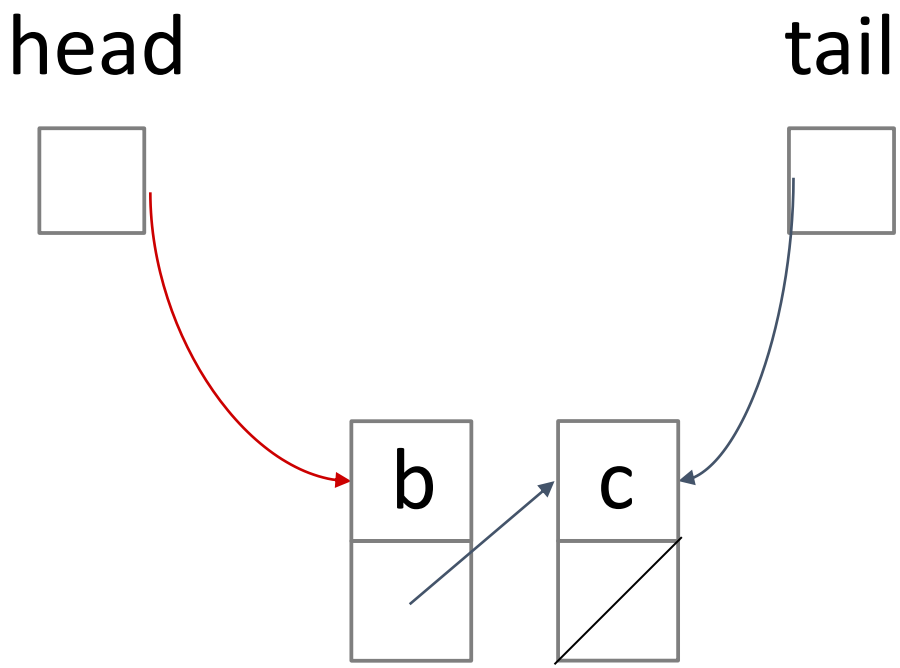


Queue Implementation with Linked List



dequeue()

Queue Implementation with Linked List



dequeue() → a

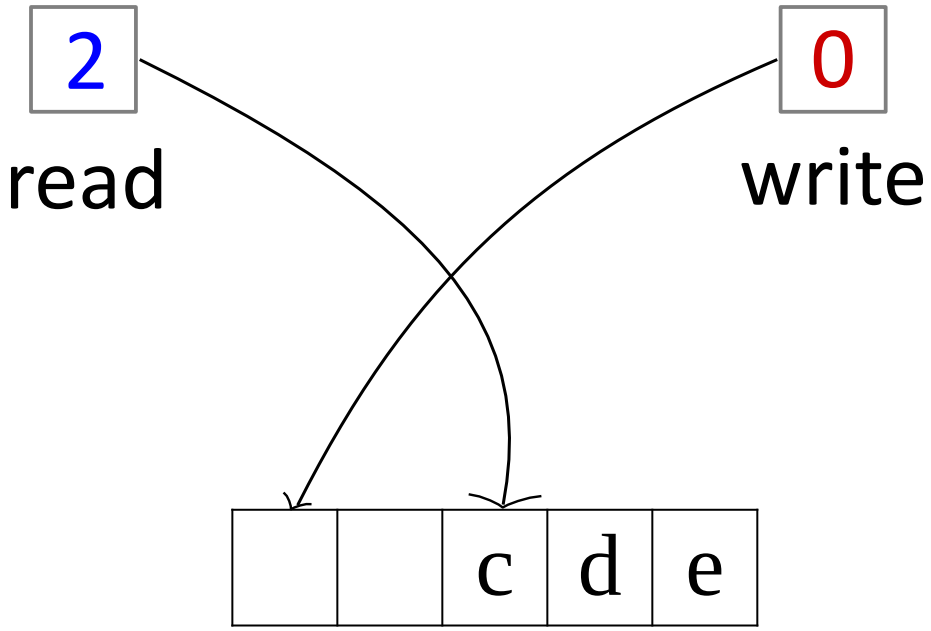
Queue Implementation with Linked List

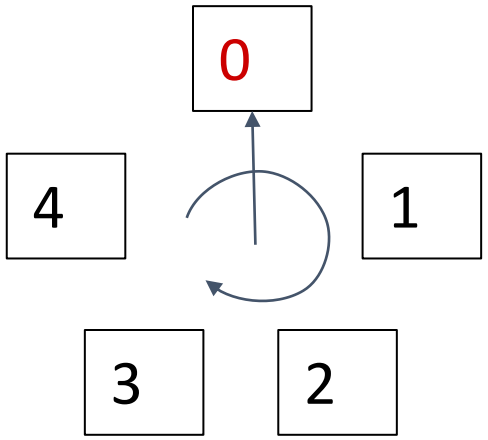
- ➔ Use Linked List **augmented** with the *tail* pointer
- ➔ For *enqueue(e)* add an element to the end
- ➔ For *dequeue()* remove from the front
- ➔ For *isEmpty()* use (*head* == NULL?)

Queue ADT: cost of operations

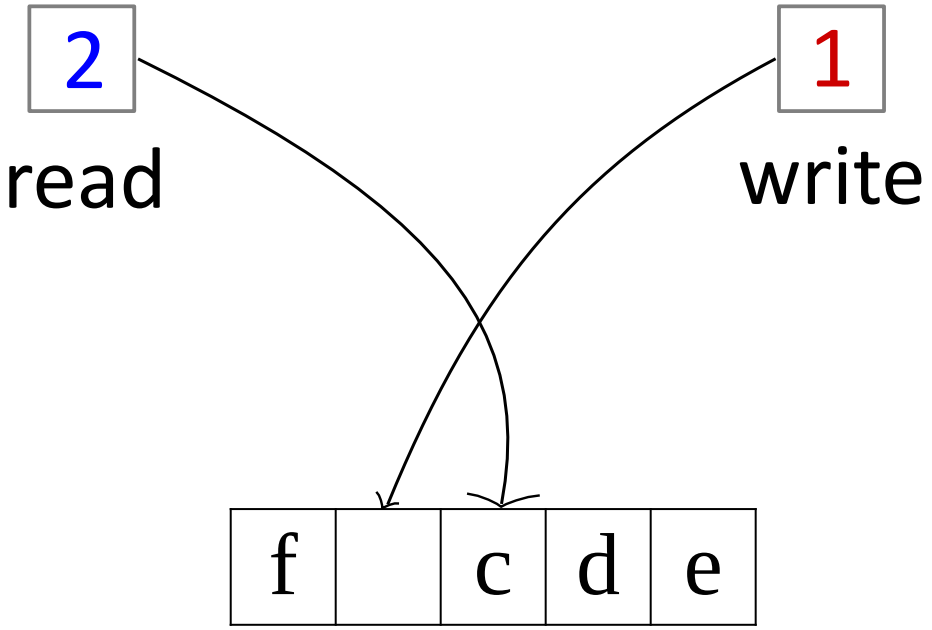
	Link. List Impl. with tail	Array Impl.
enqueue (e)	O(1)	
dequeue ()	O(1)	
getFront ()	O(1)	
IsEmpty()	O(1)	

Recap:
Queue Implementation with a circular Array





Circular array with one empty slot



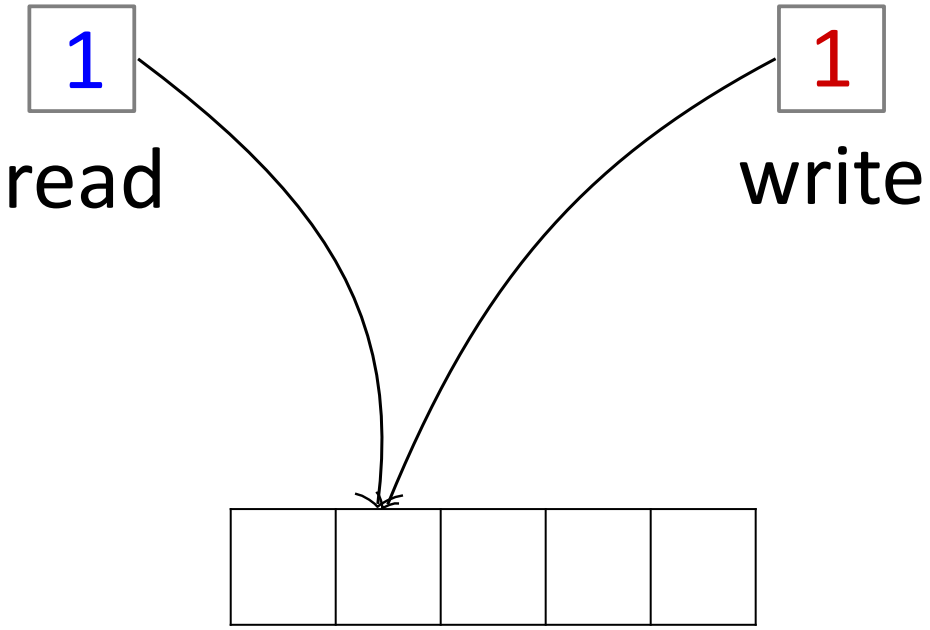
Of course we can resize the array at this point

Enqueue(g) → ERROR

Cannot set read = write

isFull() → True

When read = write the queue is empty



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)
getFront()	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 for the Queue ADT

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:

“(]()”

“][”

“([)]”

“([]”

Solution

- Stacks can be used to check whether the given expression has balanced symbols. This algorithm is used by compilers.
- Each time the parser reads one character at a time.
- If the character is an opening delimiter such as (, {, or [- then it is written to the stack.
- When a closing delimiter is encountered like), }, or]-the stack is popped.
- The opening and closing delimiters are then compared. If they match, the parsing of the string continues.
- If they do not match, the parser indicates that there is an error on the line. A linear-time
- O(n) algorithm based on stack can be given as:

Solution pseudocode

Create a stack

while (end of input is not reached) :

    If the character read is not a symbol to be balanced, ignore it

    If the character is an opening symbol like (, [, {:

        Push it onto the stack

    If it is a closing symbol like),],}:

        Pop the stack

        If the symbol popped is not the corresponding opening symbol:

            Return false

Return true

Algorithm *isBalanced*

Create a stack

while (end of input is not reached) :

    If the character read is not a symbol to be balanced, ignore it

    If the character is an opening symbol like (, [, {:

        Push it onto the stack

    If it is a closing symbol like),],}:

        Pop the stack

        If the symbol popped is not the corresponding opening symbol:

            Return false

Return true

There are two errors in this solution

Sample input 1: () (() [()])

Next symbol	Stack	
(	(	push
)		pop (. match
(	(	push
(	((	push
)	(	pop (. match
[	([	push
(	([(	push
)	([	pop (. match
]	(	pop [. match
)		pop (. match

Tracing the algorithm.
For this input the algorithm correctly return True (is balanced)

Sample input 2: ()]

Next symbol	Stack	
(	(	push
)		pop (. match
]		Stack is empty – nothing to pop

For this input the algorithm will blow off:
it will try to pop but the Stack is empty

Sample input 3: ([]

Next symbol	Stack	
(	(	push
[	([	push
]	(	pop [. match

For this input the algorithm will finish the loop over the input and will return true (is balanced).
However there is an unbalanced square bracket in the Stack.
By the end: we must check that the Stack is empty to return true

Algorithm *isBalanced*

Create a stack

while (end of input is not reached) :

    If the character read is not a symbol to be balanced, ignore it

    If the character is an opening symbol like (, [, {:

        Push it onto the stack

    If it is a closing symbol like),],}:

        If the stack is empty return false

        Pop the stack

        If the symbol popped is not the corresponding opening symbol:

            Return false

If the stack is not empty return false

Return true

CORRECTED