

CMSC 132: Object-Oriented Programming II



Linear Data Structures

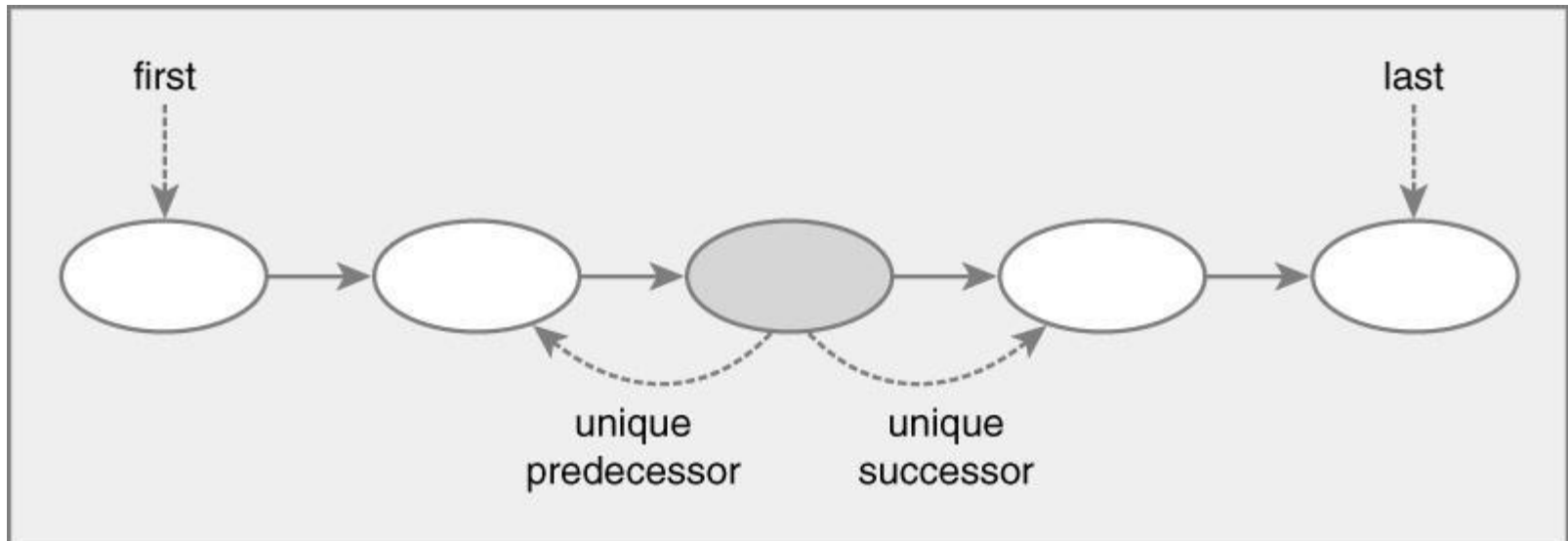
Department of Computer Science
University of Maryland, College Park

Overview

- **Linear data structures**
 - **General properties**
- **Implementations**
 - **Array**
 - **Linked list**
- **Restricted abstractions**
 - **Stack**
 - **Queue**

Linear Data Structures

- **1-to-1 relationship between elements**
 - Each element has unique predecessor & successor
 - Results in total ordering over elements
 - For any two distinct elements x and y , either x comes before y or y comes before x



Linear Data Structures

■ Terminology

- Head (first element in list) $\Rightarrow$ no predecessor
- Tail (last element in list) $\Rightarrow$ no successor

■ Operations

- Add element
- Remove element
- Find element

Add & Remove Elements

■ Add an element

■ Where?

- At head (front) of list
- At tail (end) of list
- After a particular element

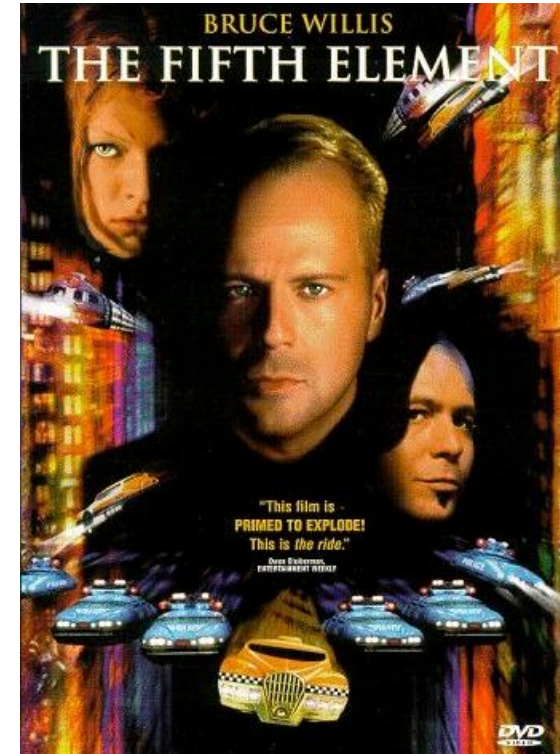
■ Remove an element

- Remove first element
- Remove last element
- Remove a particular element (e.g., String “Happy”)
 - What if “Happy” occurs more than once in list?

Accessing Elements

■ How do you find an element?

- At head (front) of list
- At tail (end) of list
- By position
 - Example: the 5th element →
- By iterating through the list, and using relative position
 - Next element (successor)
 - Previous element (predecessor)



List Implementations

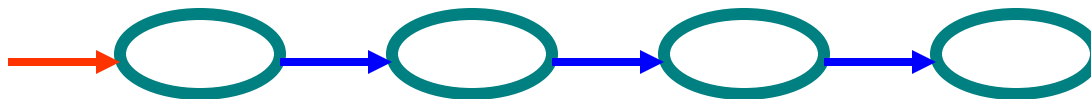
■ Two basic implementation techniques for lists

■ Store elements in an array



■ Store as a linked list

- Place each element in a separate object (node)
- Node contains reference to other node(s)
- Link nodes together



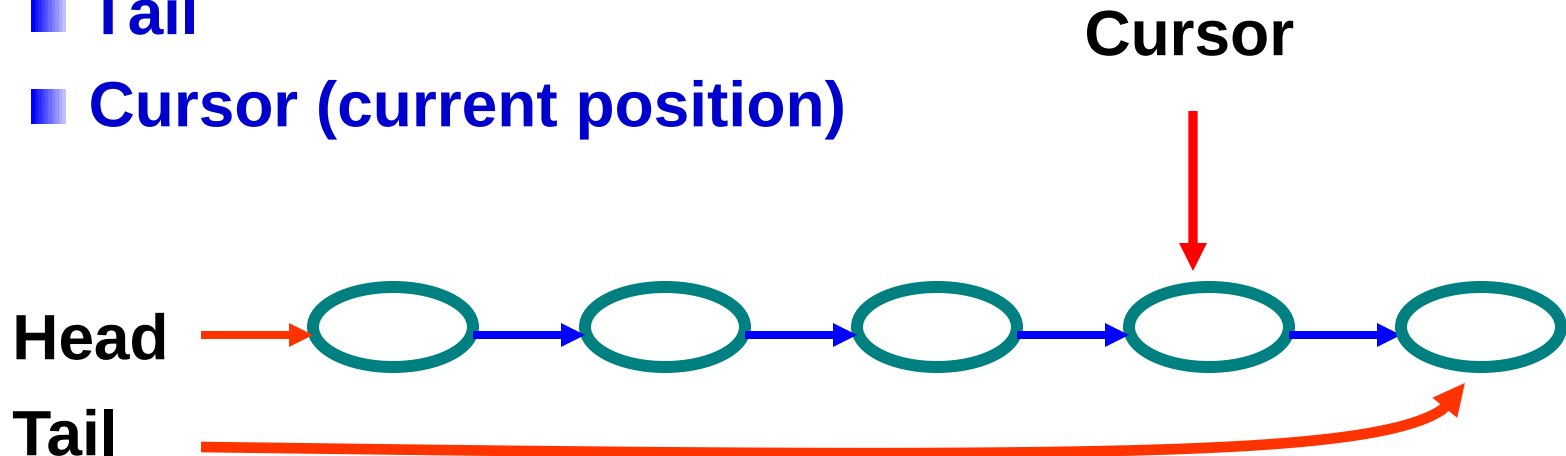
Linked List

■ Properties

- Elements in linked list are **ordered**
- Element has **successor**

■ State of List

- Head
- Tail
- Cursor (current position)



Array Implementations

■ Advantages

- Can efficiently access element at any position
- Efficient use of space
 - Space to hold reference to each element

■ Disadvantages

- Expensive to grow / shrink array
 - Can amortize cost (grow / shrink in spurts)
- Expensive to insert / remove elements in middle
- Tricky to insert / remove elements at both ends

Linked Implementation

■ Advantages

- Can efficiently insert / remove elements anywhere

■ Disadvantages

- Cannot efficiently access element at any position
 - Need to traverse list to find element
- Less efficient use of space
 - 1-2 additional references per element

Efficiency of Operations

■ Array

- Insertion / deletion = $O(n)$

- Indexing = $O(1)$

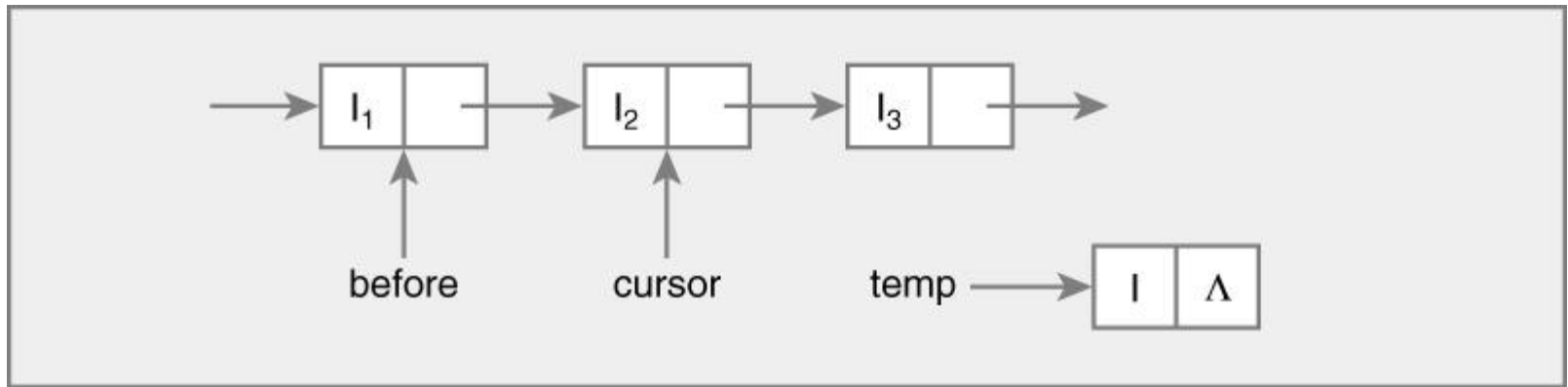
■ Linked list

- Insertion / deletion = $O(1)$

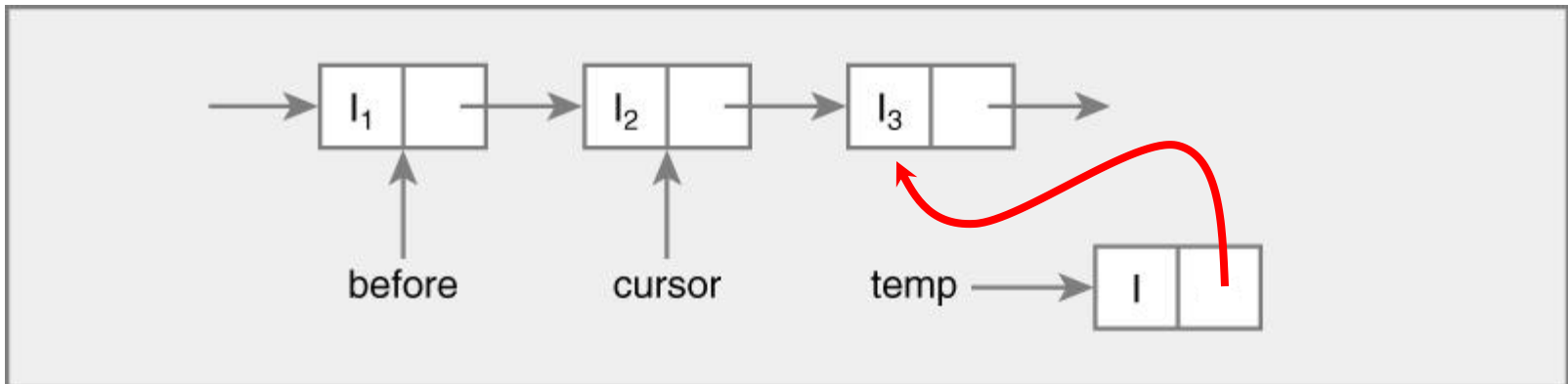
- Indexing = $O(n)$

Linked List – Insert (After Cursor)

1. Original list & new element **temp**

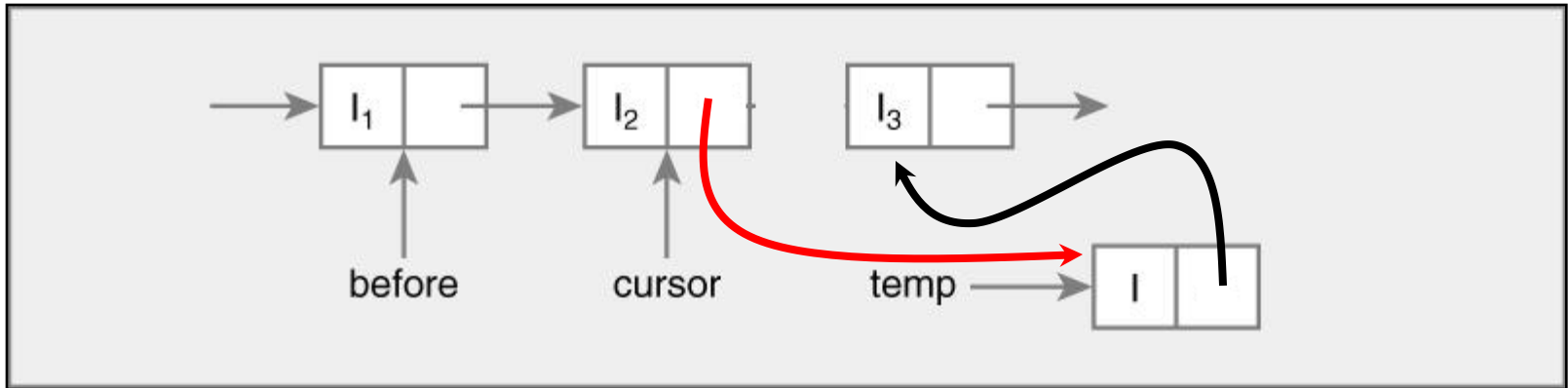


2. Modify **temp.next** $\rightarrow$ **cursor.next**

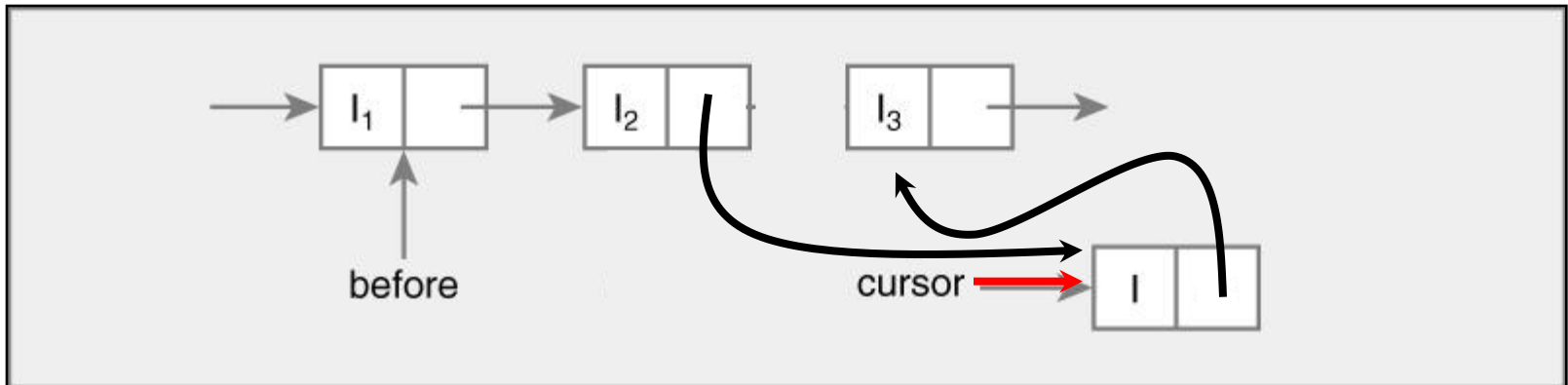


Linked List – Insert (After Cursor)

3. Modify **cursor.next** → **temp**

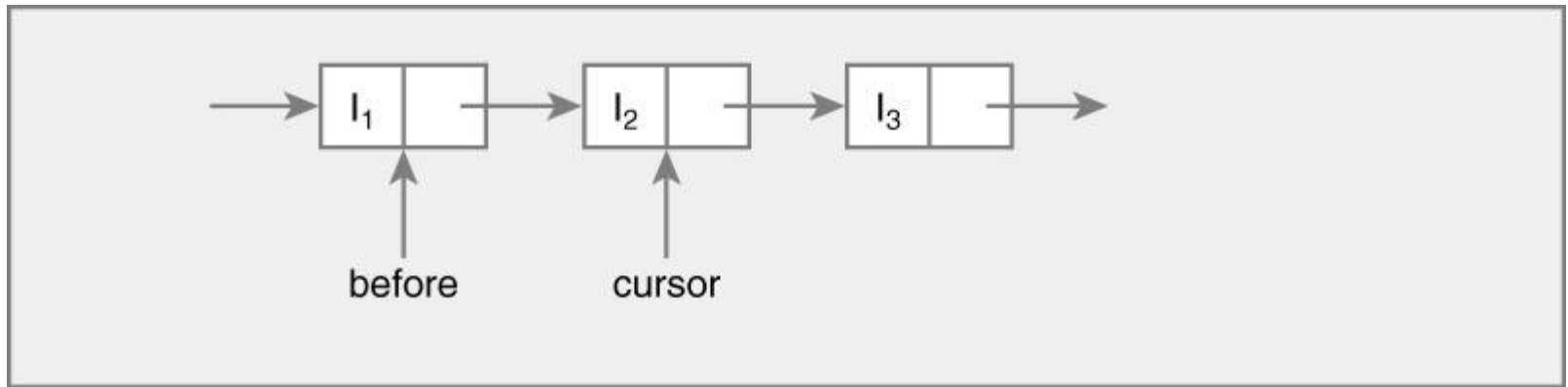


4. Modify **cursor** → **temp**

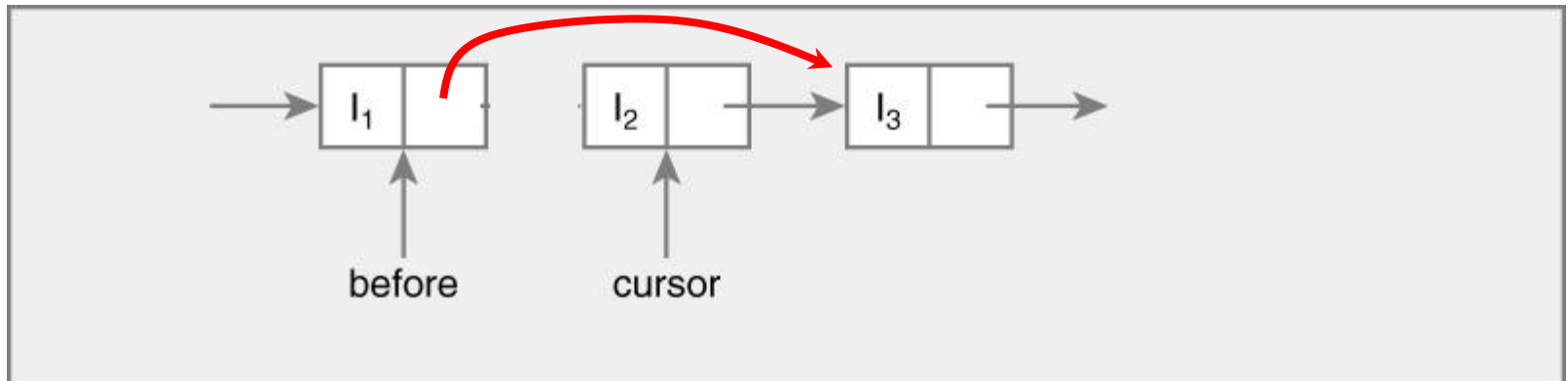


Linked List – Delete (Cursor)

1. Find **before** such that **before.next = cursor**

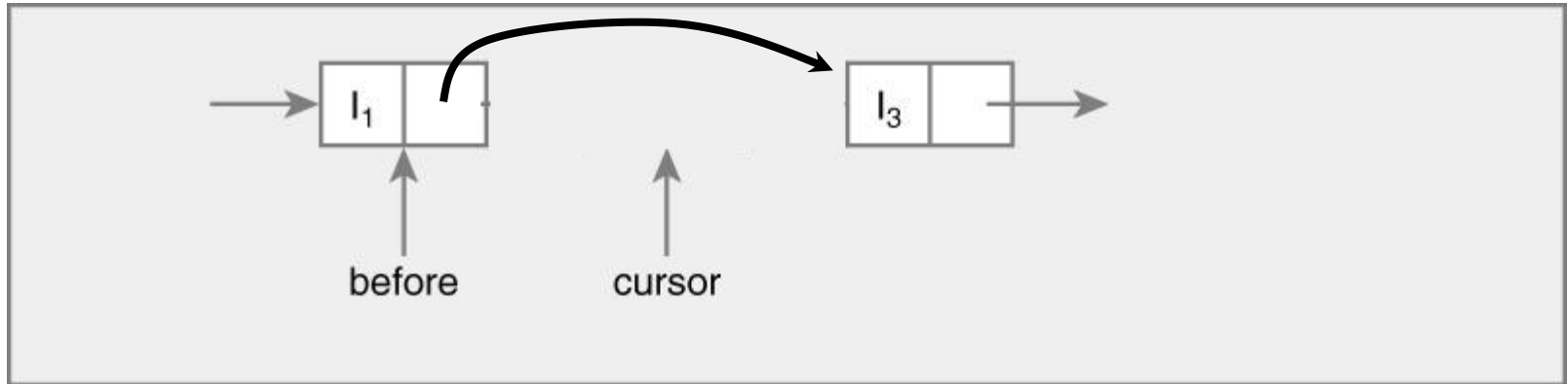


2. Modify **before.next** $\rightarrow$ **cursor.next**

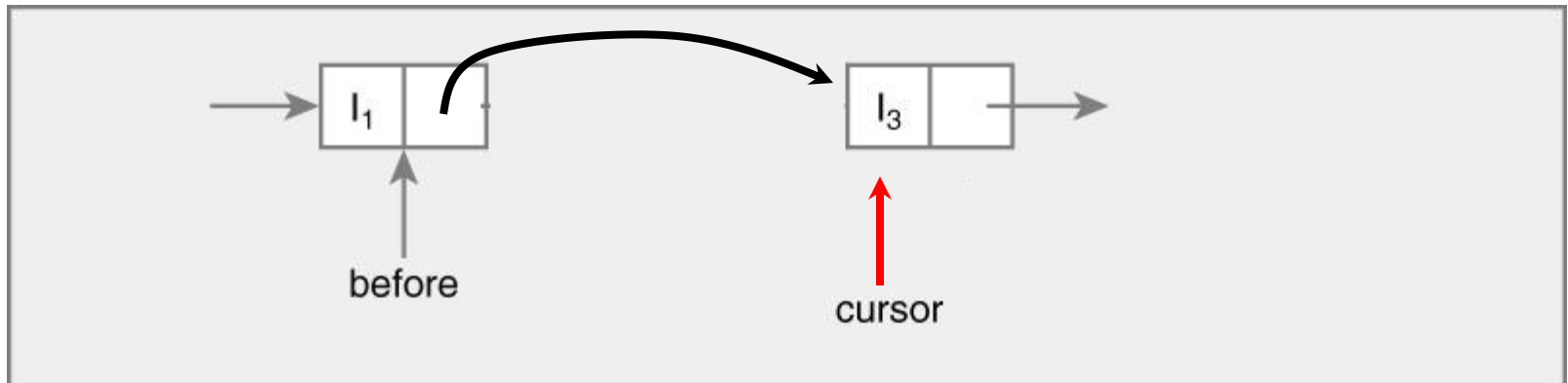


Linked List – Delete (Cursor)

3. Delete **cursor**



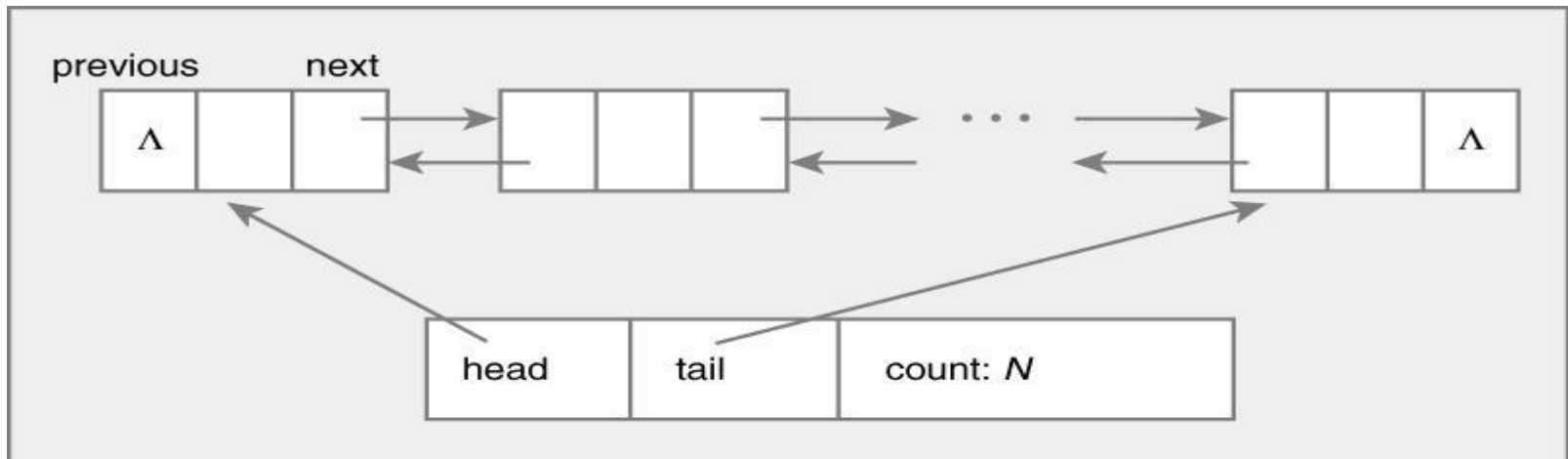
4. Modify **cursor** $\rightarrow$ before.next



Doubly Linked List

■ Linked list where

- Element has predecessor & successor

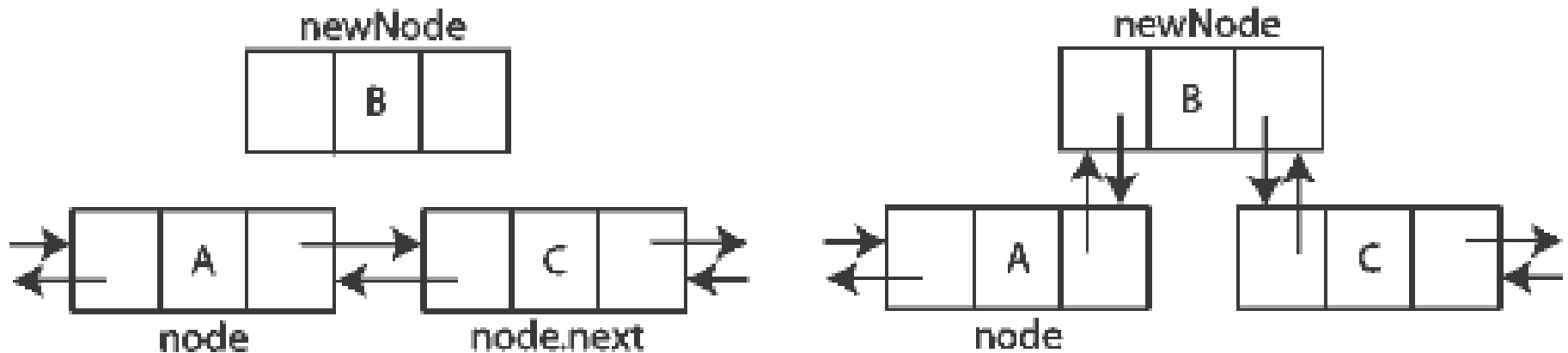


■ Issues

- Easy to find preceding / succeeding elements
- Extra work to maintain links (for insert / delete)
- More storage per node

Doubly Linked List – Insertion

■ Example

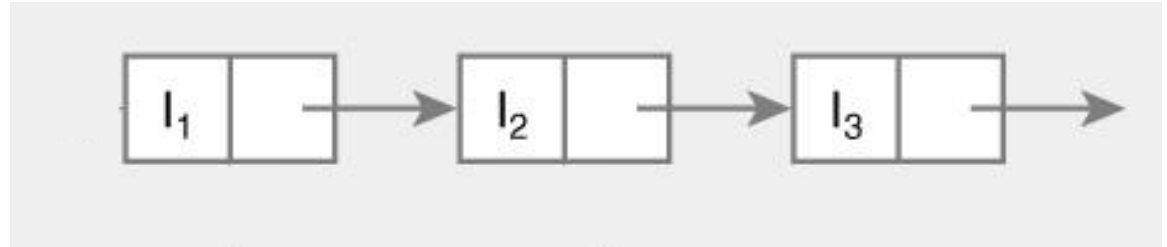


- Must update references in **both** predecessor and successor nodes

Node Structures for Linked Lists

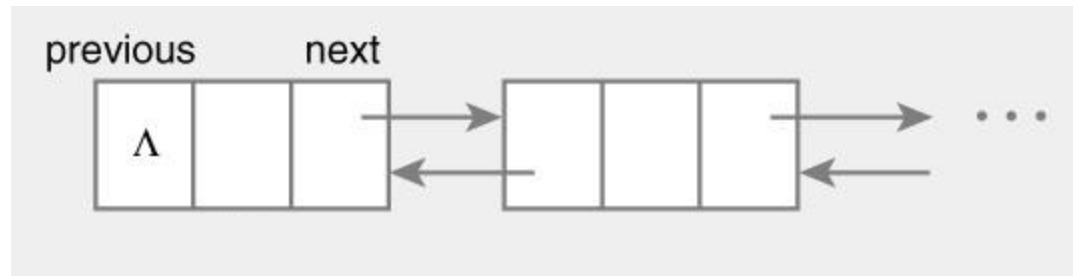
■ Linked list

```
Class Node {  
    Object data;  
    Node next;  
}
```



■ Doubly linked list

```
Class Node {  
    Object data;  
    Node next;  
    Node previous;  
}
```



Restricted Abstractions

- **Restricting the operations an abstraction supports can be a good thing**
 - Efficiently supporting only a few operations efficiently is easier
 - If limited abstraction is sufficient, easier to reason about limited abstraction than a more general one
- **Restricted list abstractions**
 - Stack (aka LIFO queue)
 - Queue (aka FIFO queue)
 - Dequeue (aka double ended queue)

Stack

■ Stack operations

■ **Push** = add element (to top)

■ **Pop** = remove element (from top)

■ Example

top → Z
Y
X

(a) *A three-element stack*

top → Y
X

(b) *After a pop () operation*

top → W
Y
X

(c) *After a push (W) operation*

Stack

■ Properties

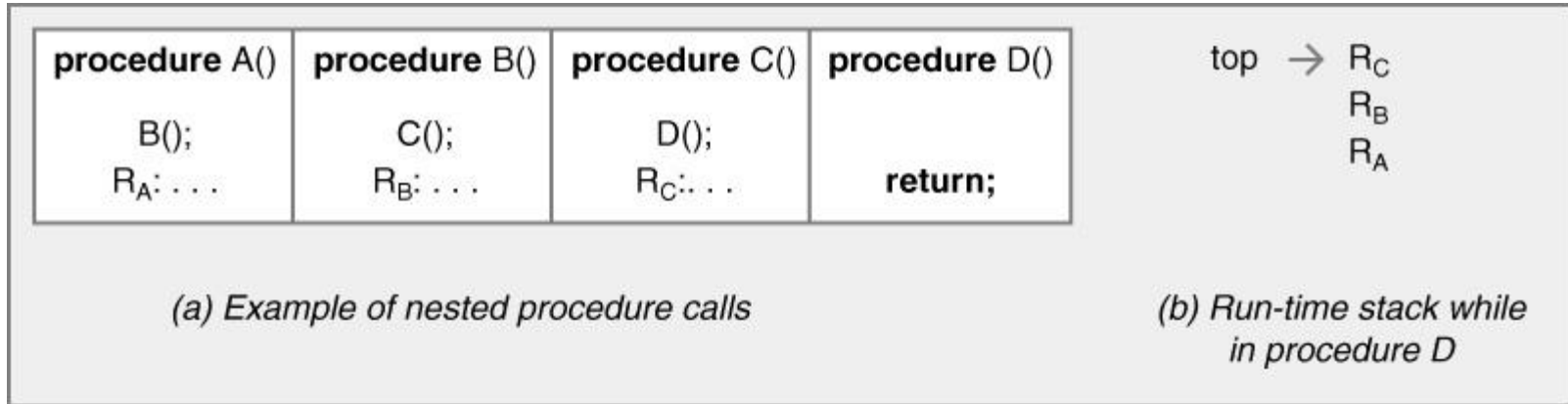
- Elements removed in **opposite** order of insertion
- Last-in, First-out (**LIFO**)

■ A restricted list where

- Access only to elements at one end
- Can add / remove elements only at one end

Stack Applications

■ Run-time procedure information



■ Arithmetic computations

■ Postfix notation

■ Simplified instruction set

■ Java bytecode

Stack Implementations

■ Linked list

■ Add / remove from head of list

top → Z
Y
X

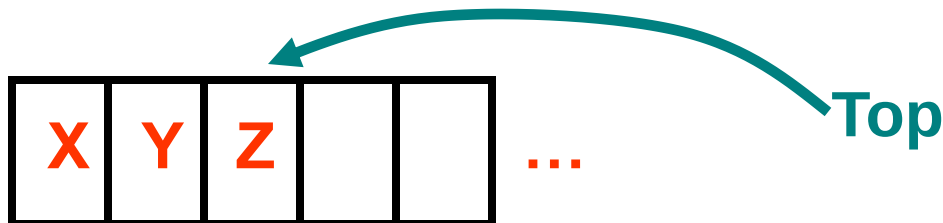
head → Z → Y → X

(a) Logical view of the stack

(b) Its linked list implementation

■ Array

■ Increment / decrement Top pointer after push / pop



Queue

■ Queue operations

- Enqueue = add element (to back)
- Dequeue = remove element (from front)

■ Example

X Y Z
^ ^
front back

(a) *Three-element queue*

Y Z
^ ^
front back

(b) *After deletion of X*

Y Z W
^ ^ ^
front back

(c) *After insertion of W*

Queue

■ Properties

- Elements removed **in order** of insertion
- First-in, First-out (**FIFO**)

■ A restricted list where

- Access only to elements at beginning / end of list
 - Add elements only to end of list
 - Remove elements only from front of list
- Alternatively, can add to front & remove from end

Queue Applications

■ Examples

- Songs to be played
- Jobs to be printed
- Customers to be served
- Citizens to cast votes

South Africa, 2004 →



Queue Implementations

■ Linked list

- Add to **tail** (back) of list
- Remove from **head** (front) of list



■ Array

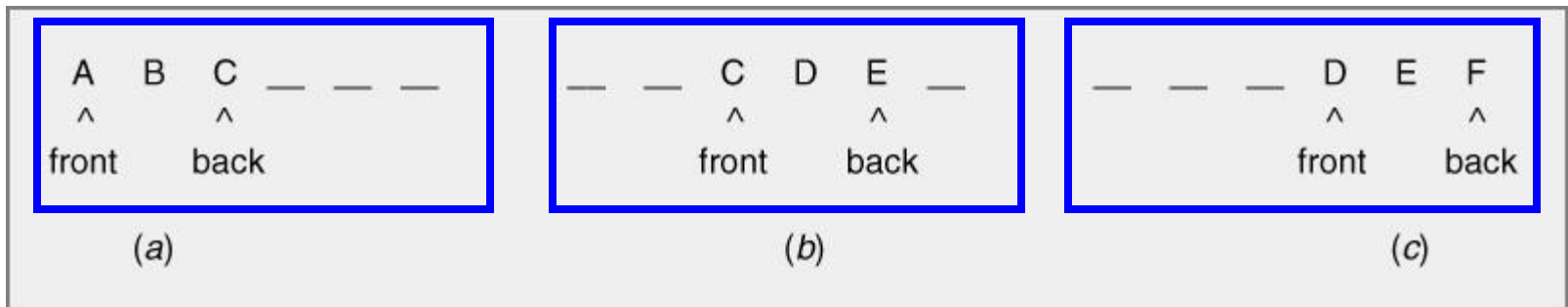
■ Circular array

Queue – Array

- Store queue as elements in array

- Problem

- Queue contents move (“inchworm effect”)



- As result, can not add to back of queue, even though queue is not full

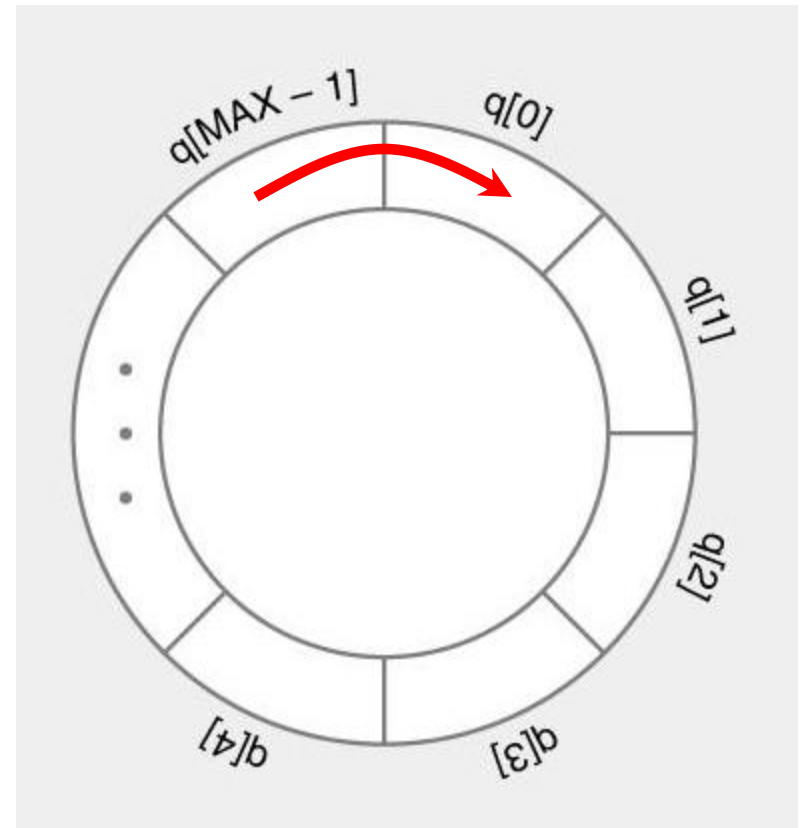
Queue – Circular Array

■ Circular array (ring)

- $q[0]$ follows $q[MAX - 1]$
- Index using $q[i \% MAX]$

■ Problem

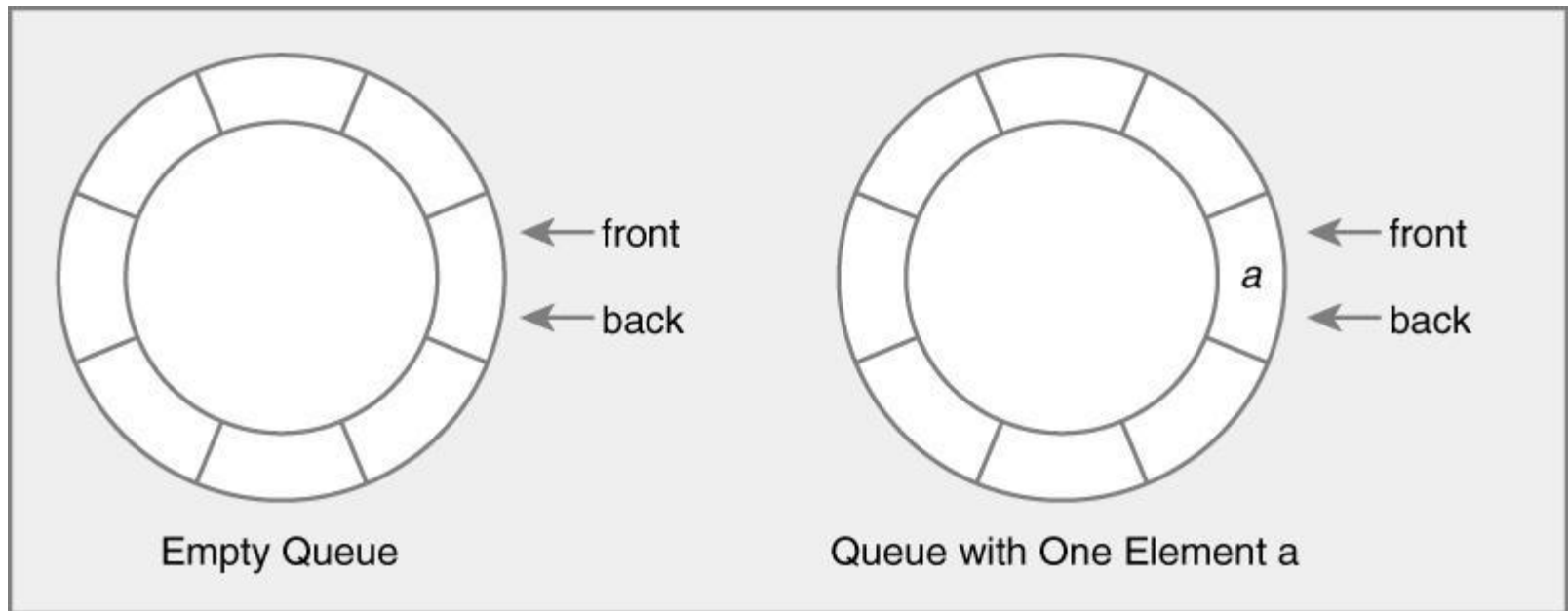
- Detecting difference between **empty** and **nonempty** queue



Queue – Circular Array

■ Approach 1

- Keep **Front** at first in
- Keep **Back** at last in



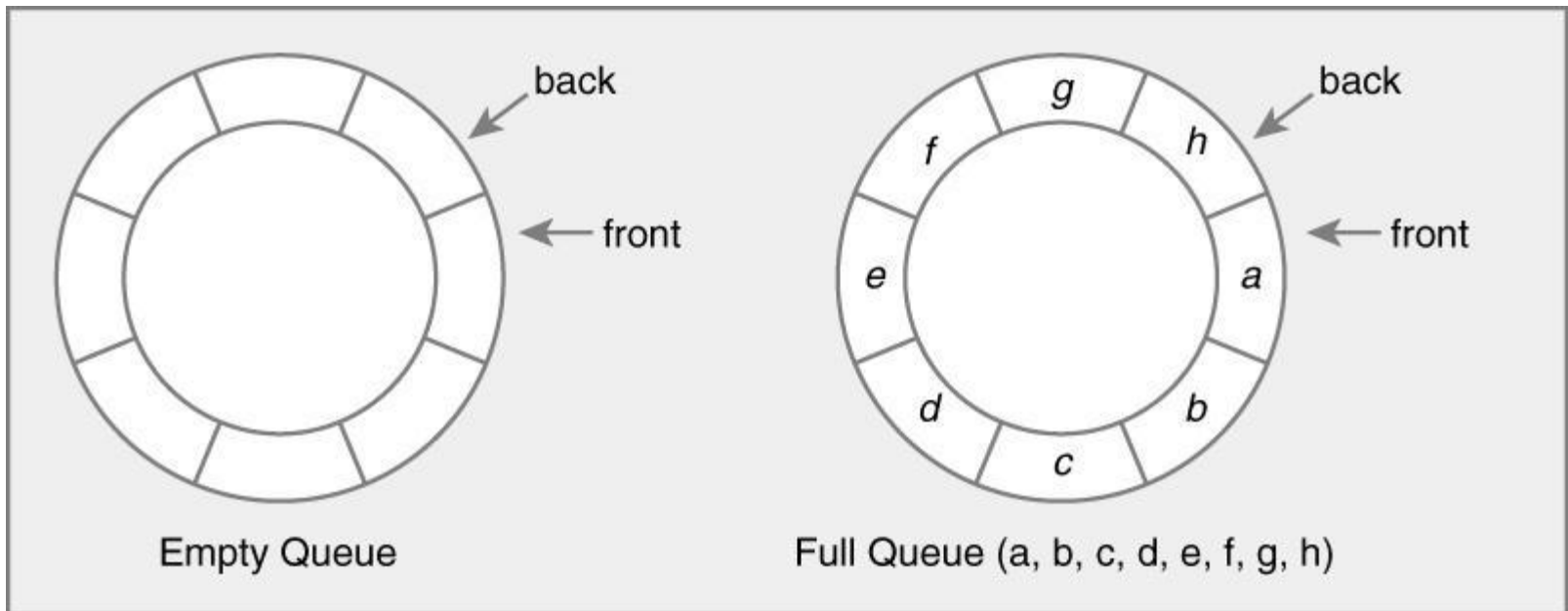
■ Problem

- Empty queue identical to queue with **1** element

Queue – Circular Array

■ Approach 2

- Keep **Front** at first in
- Keep **Back** at last in – 1



■ Problem

- Empty queue identical to **full** queue

Queue – Circular Array

- Inherent problem for queue of size **N**
 - Only **N** possible (Front – Back) pointer locations
 - **N+1** possible queue configurations
 - Queue with 0, 1, ... **N** elements
- Solutions
 - Maintain additional state information
 - Use state to recognize empty / full queue
 - Examples
 - Record **Size**
 - Record **QueueEmpty** flag
 - Leave empty element in queue
 - Store marker in queue