CSE 12:

Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Nine
15 Aug 2011

Stacks.

Review of stacks

- Stacks are a last-in-first-out (LIFO) data structure designed primarily to store data temporarily.
- Data are always added to/removed from the top of the stack.
- Stack ADT interface:

```
interface Stack<T> {
  // Adds the specified object to the top of the stack.
  void push (T o);

  // Removes the top of the stack and returns it.
  T pop () throws NoSuchElementException;

  // Returns the top of the stack without removing it.
  T peek () throws NoSuchElementException;
}
```

Stack implementations

- A stack can be implemented straightforwardly using two kinds of backing stores/underlying storage.
  - Array
    - More efficient for stacks of a fixed maximum capacity.
  - Linked list
    - More flexible for stacks with a growable capacity.

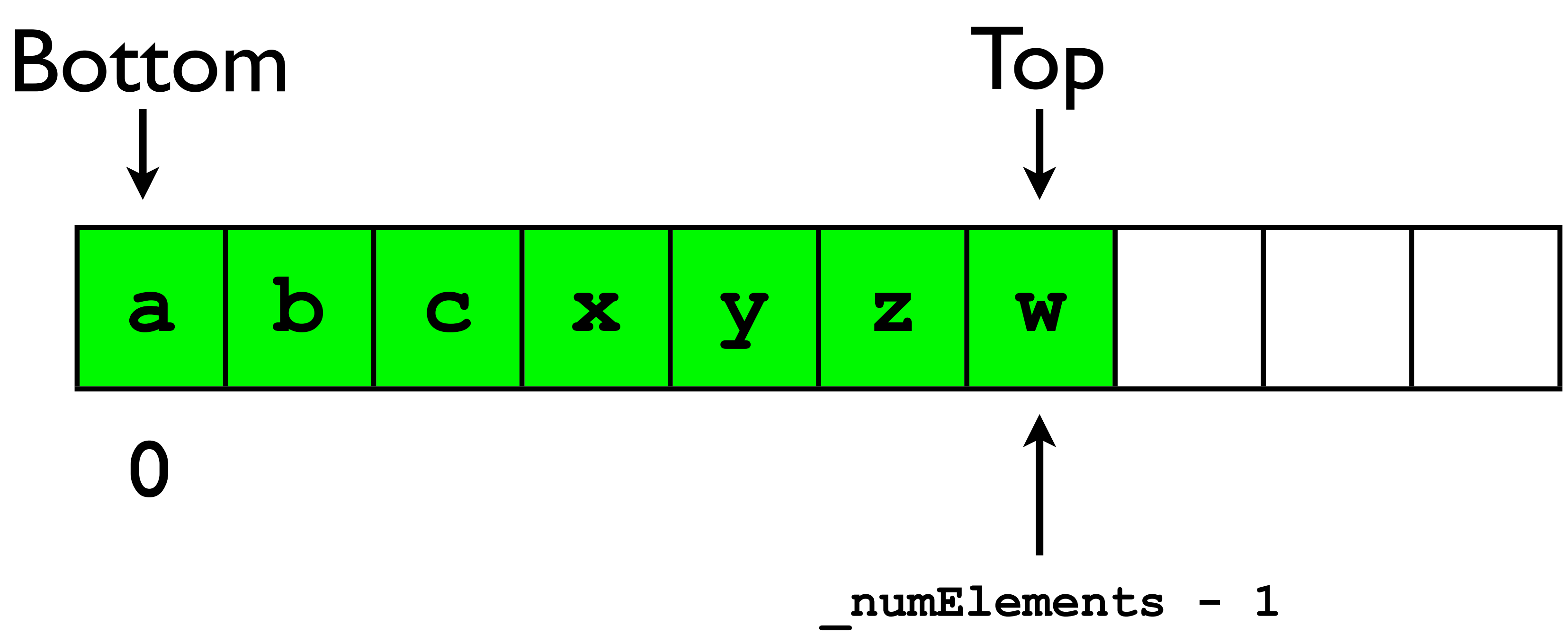
Array-based stacks

- Arrays offer a natural implementation of stacks:
  - Use `T[] _underlyingStorage` to hold elements added to stack.
    - Maximum capacity is `_underlyingStorage.length`
  - Keep track of "height" of stack using `_numElements` instance variable.

```
...
_stack.push(y);
_stack.push(z);
_stack.push(w);
```

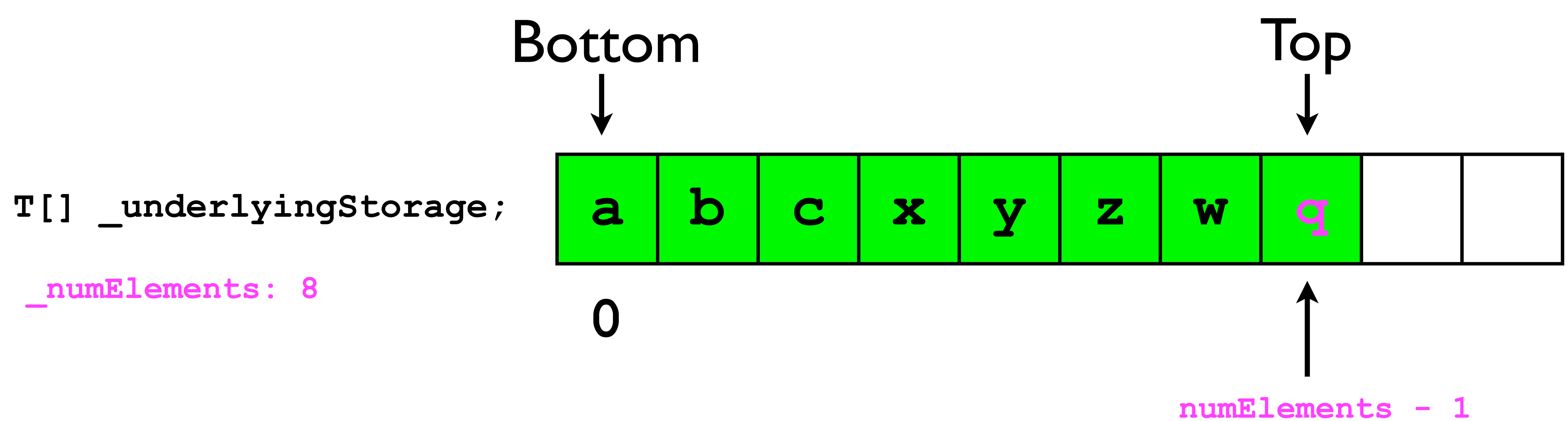
`T[] _underlyingStorage;`

`_numElements: 7`



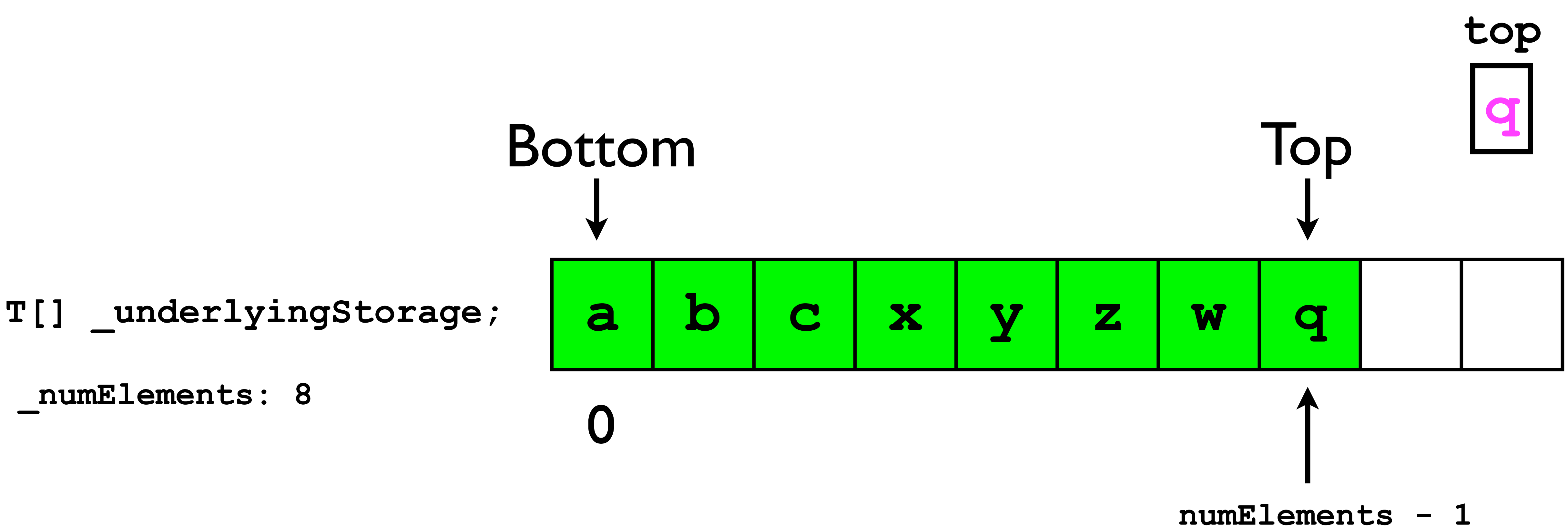
Array-based stacks

- In every call to **`push(o)`**, e.g., **`_stack.push(q);`**
  - **`_numElements`** is incremented.
  - **`o`** is stored at index **`_numElements - 1`**.



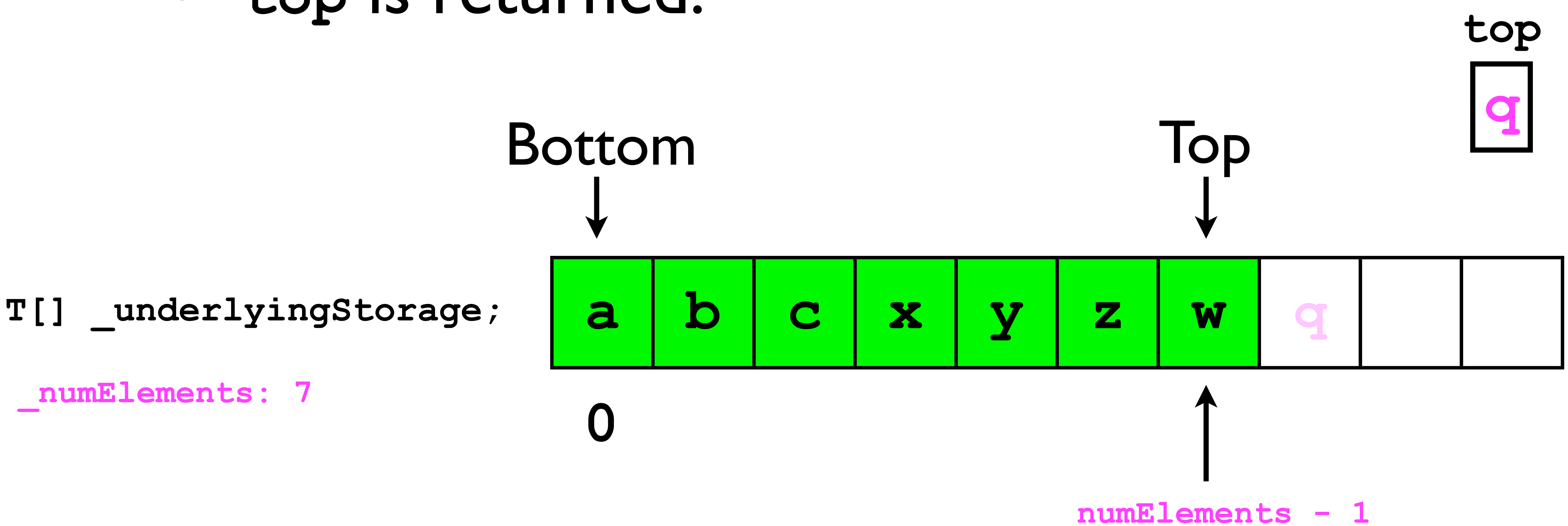
Array-based stacks

- In every call to `peek()`:
  - The element stored at index `_numElements - 1` is saved to a local variable `top`.
  - `top` is returned.



Array-based stacks

- In every call to `pop()`:
  - The element stored at index `_numElements - 1` is saved to a local variable `top`.
  - `_numElements` is decremented.
  - `top` is returned.

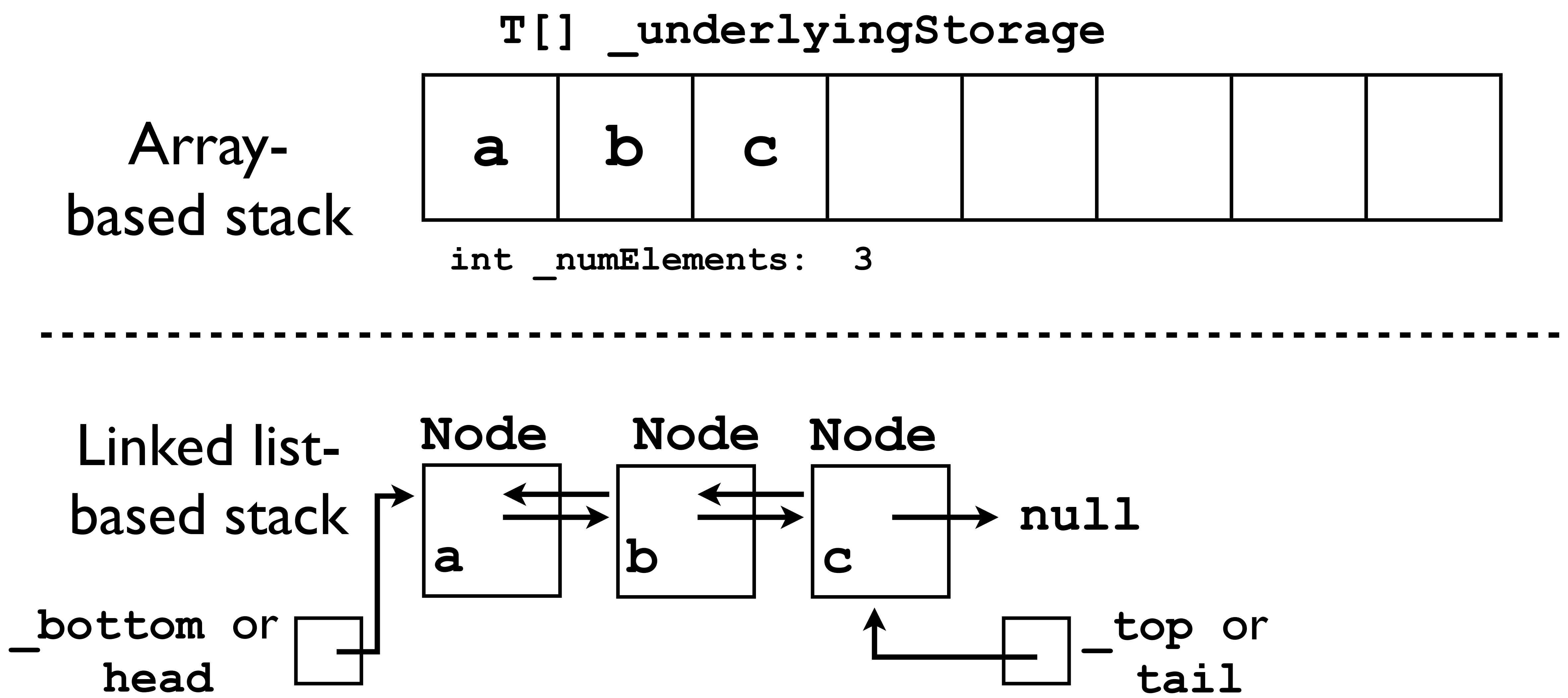


Exceptions

- If a stack has reached its maximum capacity (i.e., `_numElements == _underlyingStorage.length`) and the user calls `push(o)`, then the stack will **overflow**.
- If a stack is empty (`_numElements == 0`) and the user calls `pop()`, then the stack will **underflow**.

Linked list-based stacks

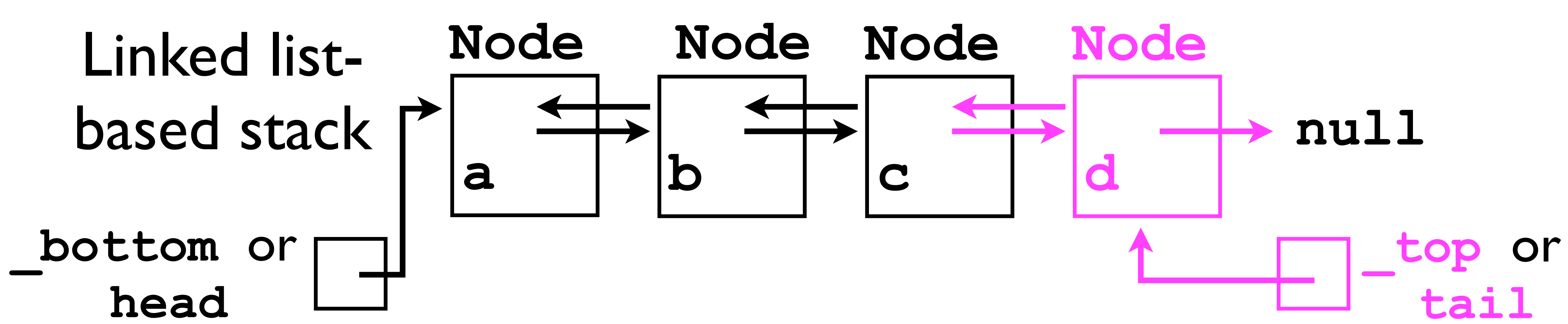
- A stack can also be implemented using a linked-list of nodes:



Linked list-based stacks

- Each call to **`push(o)`** adds a new **`Node`** to the **`_top`** of the stack (or **`_tail`** of the list), e.g.:

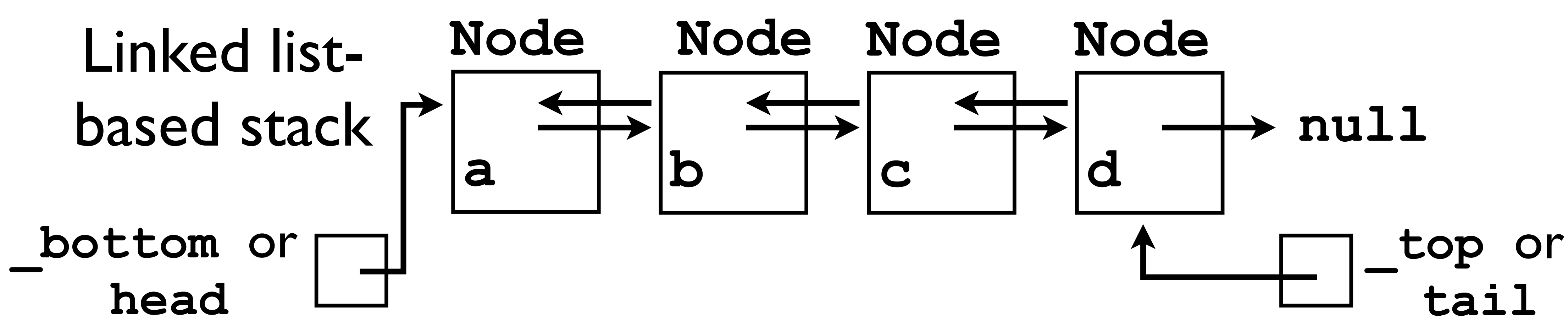
```
_stack.push(d);
```



Linked list-based stacks

- Each call to `peek()` simply returns the data referenced by `_top` (or `_tail`):

```
final T top = _stack.peek(); // d
```

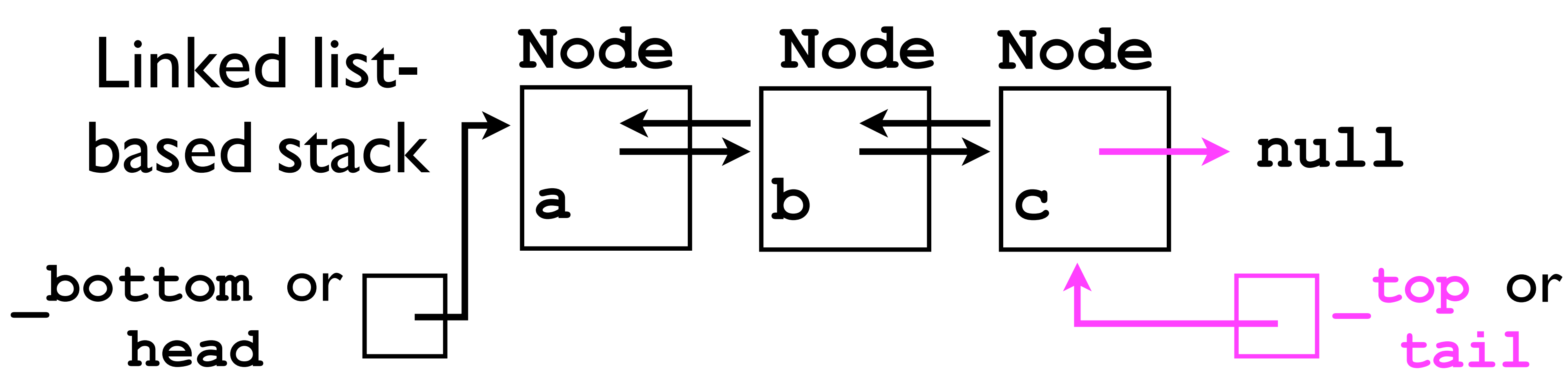


Linked list-based stacks

- Each call to **`pop()`** removes the **`Node`** at the **`_top`** of the stack (or **`_tail`** of the list) and returns the data it referenced, e.g.:

```
final T top = _stack.pop(); // d
```

Linked list-based stack



Linked list-based stacks

- A linked list-based stack ADT could be implemented by defining a static inner-class **`Node`** and essentially "re-implementing" the **`DoublyLinkedList12`** functionality.
  - But this would be wasteful -- we already have a functioning **`DoublyLinkedList12`** ADT.
  - We can save time and the possibility of human error by "adapting" the **`DoublyLinkedList12`** ADT to a **`Stack`** ADT.

“Adapter” design pattern

- In software engineering, one of the classic “design patterns” is the *adapter*.
  - An *adapter* is a class that “maps” from the interface of one ADT -- the one we’re trying to implement -- into the interface of another ADT *that already exists*.
  - If we adapt an ADT B to implement another ADT A, then every method of A must be “converted” into a related call of B.
  - In particular, we can adapt the `List12` ADT (implemented by `DoublyLinkedList12`) to satisfy the `Stack` ADT interface specification...

Stack as adaptation of linked list

- How to "map" from `Stack` ADT to `List12` ADT:
  - Stack constructor instantiates
    `_dll = new DoublyLinkedList12<T>();`
  - `push(o)` calls `_dll.addToBack(o)`
  - `pop()` calls `_dll.removeBack()`
  - `peek()` calls `_dll.get(_dll.size() - 1)`

Queues.

Queues

- Queues are a first-in-first-out (FIFO) data structure used typically for temporary data storage.
  - Similarly to a train entering a tunnel, the first car to enter the tunnel is the first car to exit the tunnel.

- As with stacks, queues find many uses in *systems programming* (programming of the operating system).

Queues for Interprocess Communication

- One of the classical use-cases for queues is for **inter-process communication** (IPC).
- Programs sometimes need to send messages to other programs in order to get work done.
  - E.g., to write anything to the terminal or to a file, a program must send a message to the operating system requesting that the specified data be written.

Queues for Interprocess Communication

- IPC can take place between a computer program and the operating system, or between two computer programs:
- Examples:
  - **`cat`** program to operating system: "please take this message [contents of a particular file] and print it to the screen."
  - **`ls`** program to **`more`** program: "please take this message [contents of a directory] and split it into convenient page-size chunks."

Queues for Interprocess Communication

- In IPC, it is crucial that the messages be received in the same order that they are sent.
  - E.g., if we send the following messages...
    `sendMsg("h");`
    `sendMsg("e");`
    `sendMsg("l");`
    `sendMsg("l");`
    `sendMsg("o");`
  - ...then we expect "**`hello`**" to be received, and not "**`lehol`**"!
    - We need messaging to be a *FIFO* process.

Queues for Interprocess Communication

- Suppose Program A wishes to send a message to Program B (perhaps running concurrently).
- One way in which we might conceive of implementing IPC is for A to call a *method* of B (“procedure call”).

```
class A {  // Program A
  Program _b;
  void someMethod () {
    _b.pleasePrint(“testing”);
  }
}
```

Send message

```
class B {  // Program B
  void pleasePrint (String msg) {
    ...  // Process the request
  }
}
```

Queues for Interprocess Communication

- Unfortunately, this "procedure call" from A to B is problematic:
  - What if B is currently doing something else? (Remember that it's a separate program.) B might need a long time before it can process A's message.
    - As a consequence, A's procedure call will "hang" execution of A.

Implementing IPC using a procedure call effectively "couples" programs A and B.

```
class A {  // Program A
  Program _b;
  void someMethod () {
    _b.pleasePrint("testing");
  }
}
```

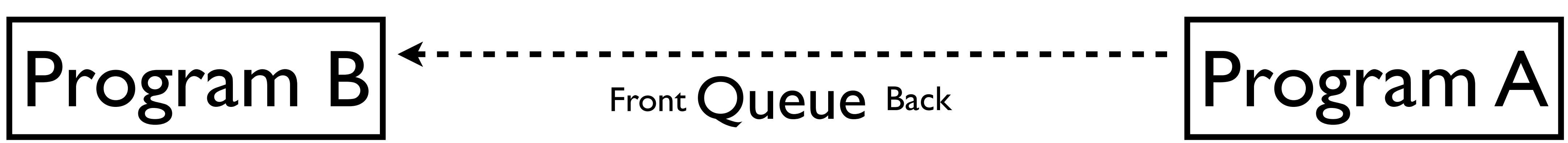
Send message

```
class B {  // Program B
  void pleasePrint (String msg) {
    ...  // Process the request
  }
}
```

Queues for Interprocess Communication

- Message queues offer a way of "decoupling" the sending of a message (from A) and the receiving/processing of a message (in B).
- With message queues, two programs A and B that wish to communicate can instantiate a message queue between them.

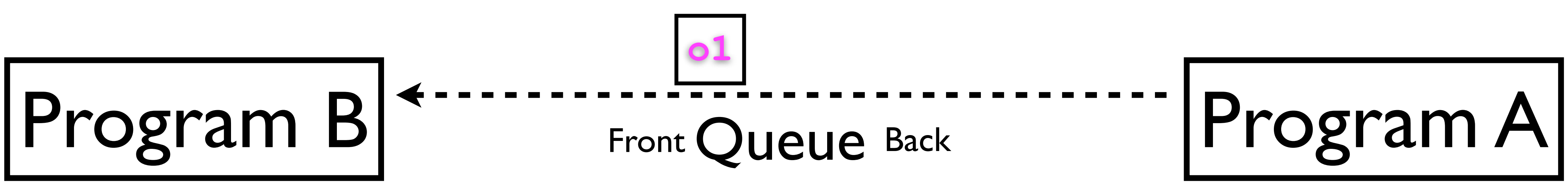
```
queueAB = new Queue();
```




```
_b._queue = queueAB;                    _a._queue = queueAB;
```

Queues for Interprocess Communication

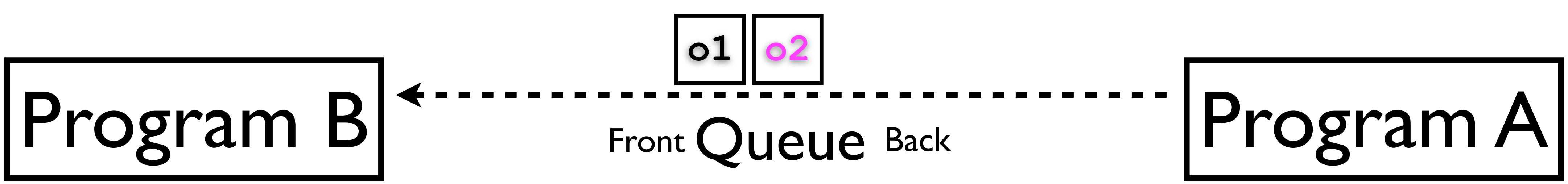
- Whenever A wishes to send a message M to B, it **enqueues** the message onto the queue.




```
_queue.enqueue(o1);
```

Queues for Interprocess Communication

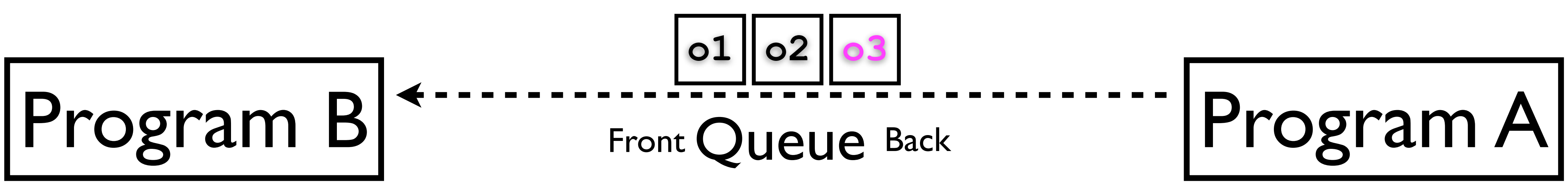
- Whenever A wishes to send a message M to B, it **enqueues** the message onto the queue.




```
_queue.enqueue(o1);
_queue.enqueue(o2);
```

Queues for Interprocess Communication

- Whenever A wishes to send a message M to B, it **enqueues** the message onto the queue.

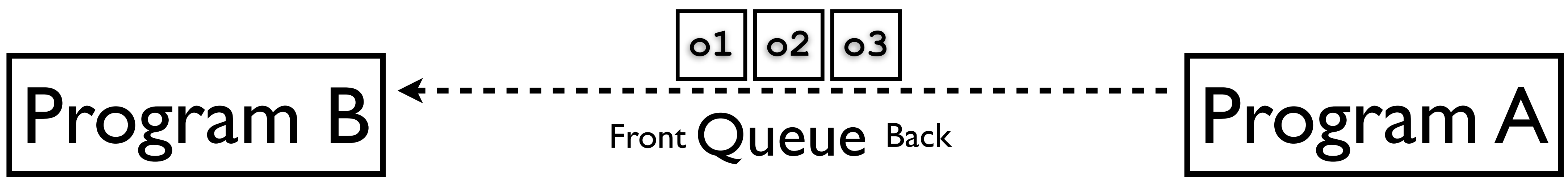



```
_queue.enqueue(o1);
_queue.enqueue(o2);
_queue.enqueue(o3);
```

Queues for Interprocess Communication

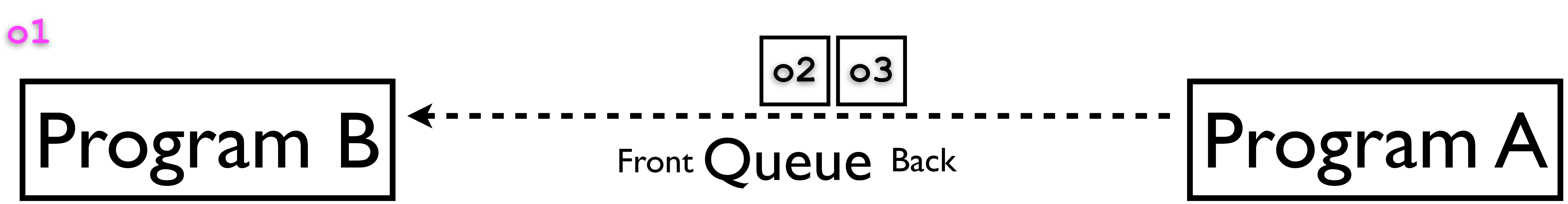
- Whenever B wishes to receive/process a message M from A, it **dequeues** a message from the queue.
  - In accordance with the FIFO principle, the *first* message B dequeues is the *first* message A had enqueued.

A queue is sometimes referred to simply as a *FIFO*.



Queues for Interprocess Communication

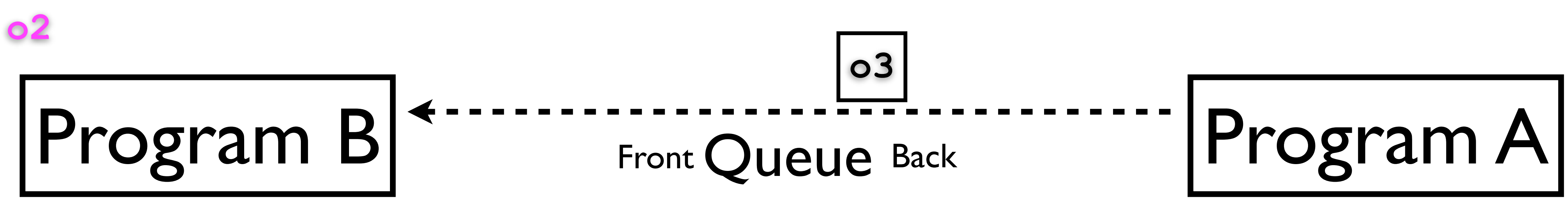
- Whenever B wishes to receive/process a message M from A, it **dequeues** a message from the queue.
  - In accordance with the FIFO principle, the *first* message B dequeues is the *first* message A had enqueued.




```
_queue.dequeue();  // o1
```

Queues for Interprocess Communication

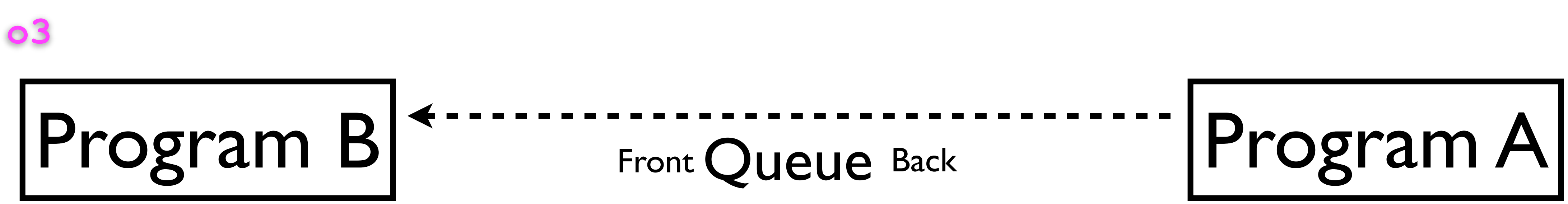
- Whenever B wishes to receive/process a message M from A, it **dequeues** a message from the queue.
  - In accordance with the FIFO principle, the *first* message B dequeues is the *first* message A had enqueued.




```
_queue.dequeue();  // o2
```

Queues for Interprocess Communication

- Whenever B wishes to receive/process a message M from A, it **dequeues** a message from the queue.
  - In accordance with the FIFO principle, the *first* message B dequeues is the *first* message A had enqueued.




```
_queue.dequeue();  // o3
```

Queues for Interprocess Communication

- The queue as an "intermediary communication medium" between A and B allows both programs to operate independently.
  - A can send a message to B without waiting for B to finish processing it.
  - B can process messages from A when it is convenient to receive them.

```
class B {  // Program B
  void processQueue () {
    ...
    final String msg =
      _queue.dequeue();
    print(msg);
  }
}
```

```
class A {  // Program A
  Queue _queue;
  void someMethod () {
    _queue.enqueue(
      "print: testing"
    );
  }
}
```

Queue ADT

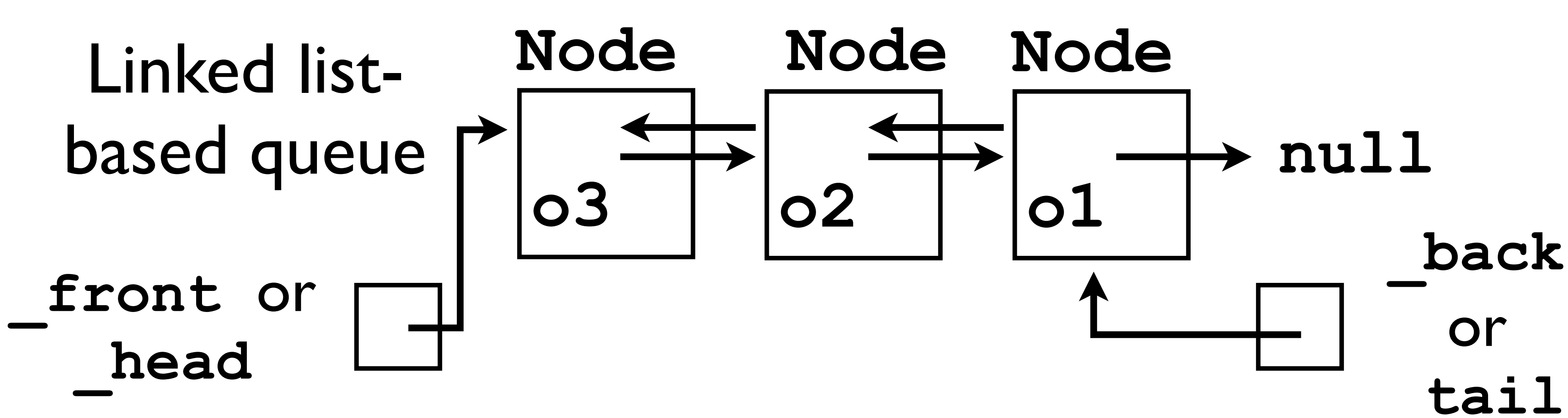
- The interface for a Queue ADT looks as follows:

```
interface Queue<T> {
  // Adds o to the back of the queue.
  void enqueue (T o);

  // Removes the object at the front of the
  // queue.
  T dequeue () throws NoSuchElementException;
}
```

Implementing a queue

- A queue can probably be most easily conceptualized and implemented as a linked list.
- The head of the list is the *front* of the queue.
- The tail is the *back* of the queue.
- Calls to **`enqueue(o)`** add a new **`Node`** to the *back*.
- Calls to **`dequeue()`** remove a **`Node`** (and return its data) from the *front*.



Adapting a DoublyLinkedList12

- As with the `Stack` ADT, the `Queue` ADT also lends itself to *adapting* the existing `DoublyLinkedList12` ADT to suit its needs:
  - Instantiate `_dll = new DoublyLinkedList12<T>();`
  - Calls to `enqueue(o)`: `_dll.addToBack(o);`
  - Calls to `dequeue()`: `return _dll.removeFront();`

Array-based queue

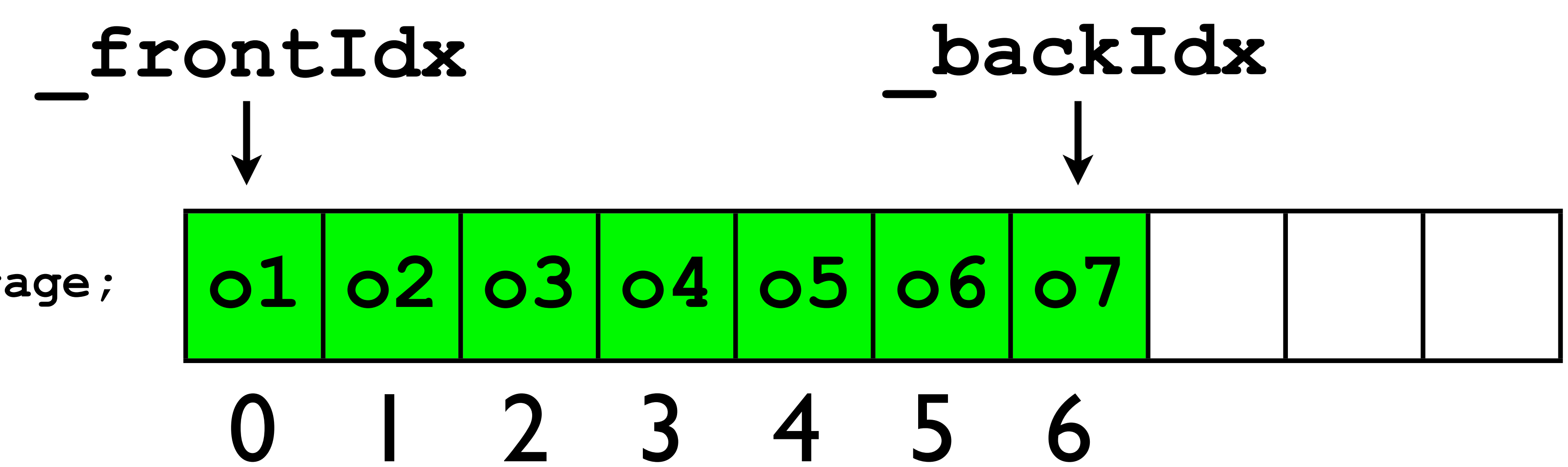
- Like stacks, queues too can be implemented using an array as the underlying storage.
- However, arriving at at an efficient solution is non-trivial.
- Assume following instance variables:
  - **`T[] _underlyingStorage`**
  - **`int _frontIdx, _backIdx`** -- indices into **`_underlyingStorage`** of where the front and back of the queue are located.

Array-based queue

- `enqueue(o)`: Append to the *back* of the array:
  - This is easy:

```
_backIdx++;
_underlyingStorage[_backIdx] = o;
```

`T[] _underlyingStorage;`



Array-based queue

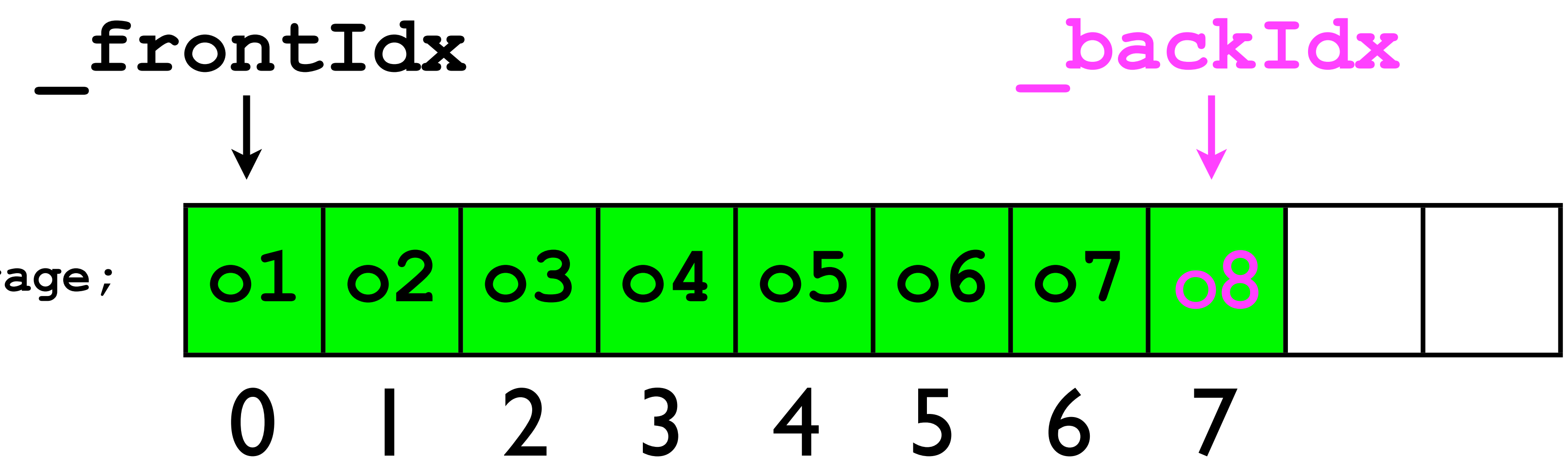
- `enqueue(o)`: Append to the *back* of the array:
  - This is easy:

    O(1) time cost

```
_backIdx++;
_underlyingStorage[_backIdx] = o;
```

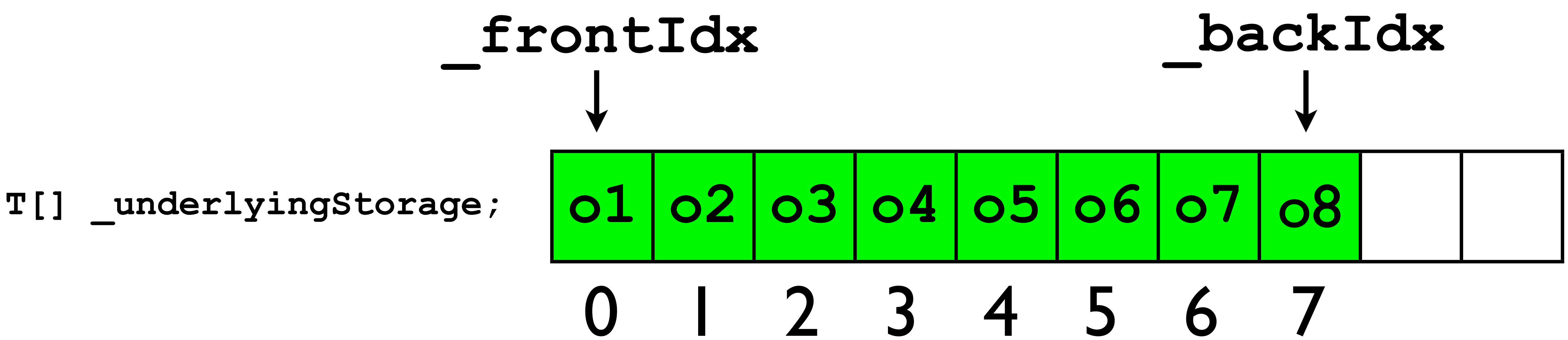
  - Example: `_queue.enqueue(o8);`

`T[] _underlyingStorage;`



Array-based queue

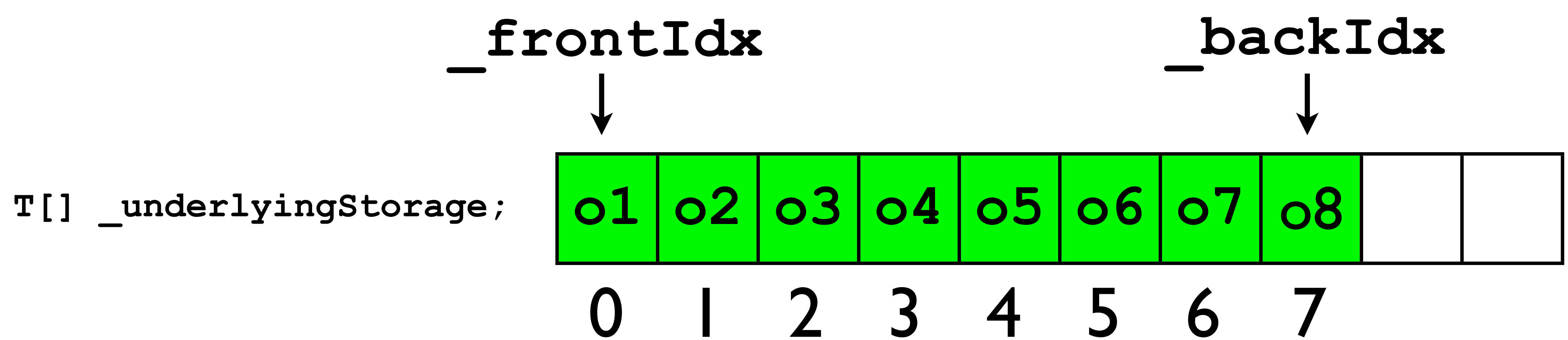
- `dequeue()`: Remove from the *front* of the array:
  - This is harder -- what happens when we remove `o1`?
  - There are several ways one can attempt to implement this method...



dequeue() -- Attempt #1

- One possibility is to “shift down” by 1 the entire queue after the front has been removed:

```
final T front = _underlyingStorage[0];
for (int i = _frontIdx+1; i <= _backIdx; i++) {
  _underlyingStorage[i-1] = _underlyingStorage[i];
}
_backIdx--;  // The back has “moved up” by 1
return front;
```

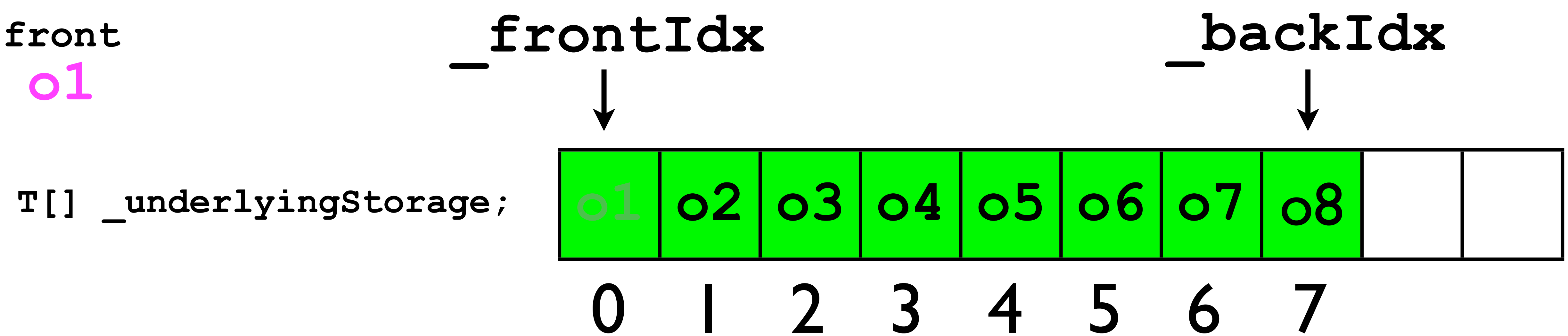


dequeue() -- Attempt #1

- One possibility is to "shift down" by 1 the entire queue after the front has been removed:

```
final T front = _underlyingStorage[0];
for (int i = _frontIdx+1; i <= _backIdx; i++) {
  _underlyingStorage[i-1] = _underlyingStorage[i];
}
_backIdx--;  // The back has "moved up" by 1
return front;
```

- Example: `_queue.dequeue();`



dequeue() -- Attempt #1

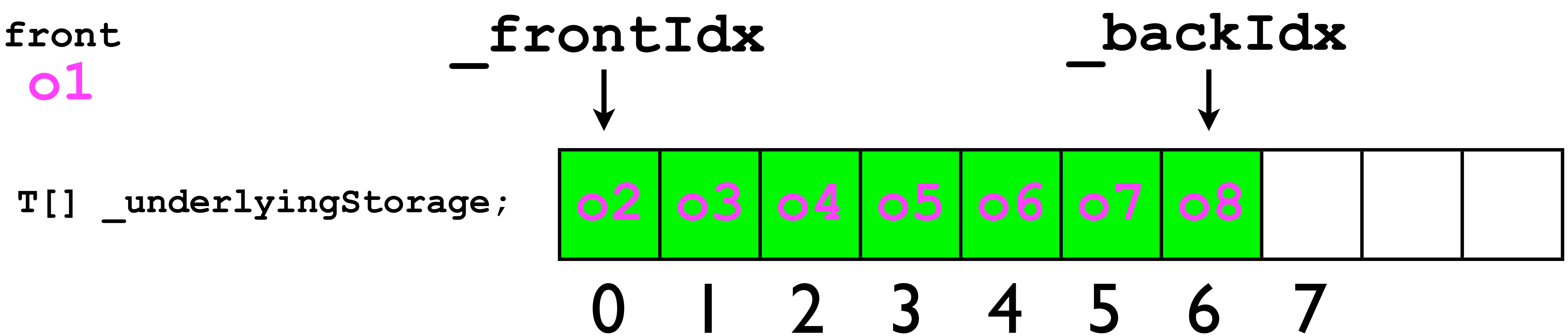
- One possibility is to "shift down" by 1 the entire queue after the front has been removed:

```
final T front = _underlyingStorage[0];
for (int i = 1; i < _backIdx; i++) {
  _underlyingStorage[i-1] = _underlyingStorage[i];
}
_backIdx--;  // The back has "moved up" by 1
return front;
```

O(n) time cost!

`_frontIdx` never changes -- always 1!

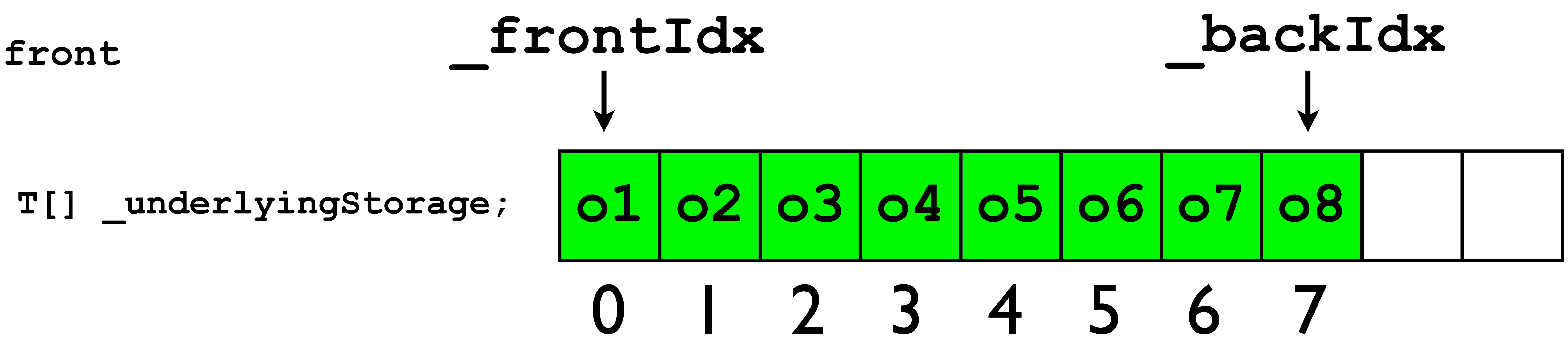
- Example: `_queue.dequeue();`



dequeue() -- Attempt #2

- Another possibility is to allocate a huge array for the `_underlyingStorage`, and then just keep advancing `_frontIdx` by 1 whenever `dequeue()` is called.

```
final T front = _underlyingStorage[_frontIdx];
_frontIdx++;
return front;
```

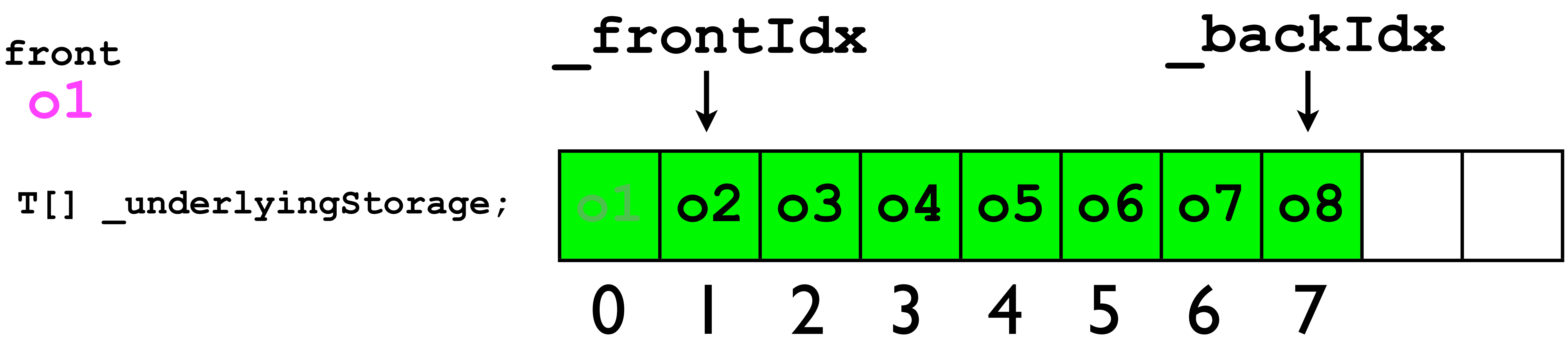


dequeue() -- Attempt #2

- Another possibility is to allocate a huge array for the `_underlyingStorage`, and then just keep advancing `_frontIdx` by 1 whenever `dequeue()` is called.

```
final T front = _underlyingStorage[_frontIdx];
_frontIdx++;
return front;
```

- Example: `_queue.dequeue();`



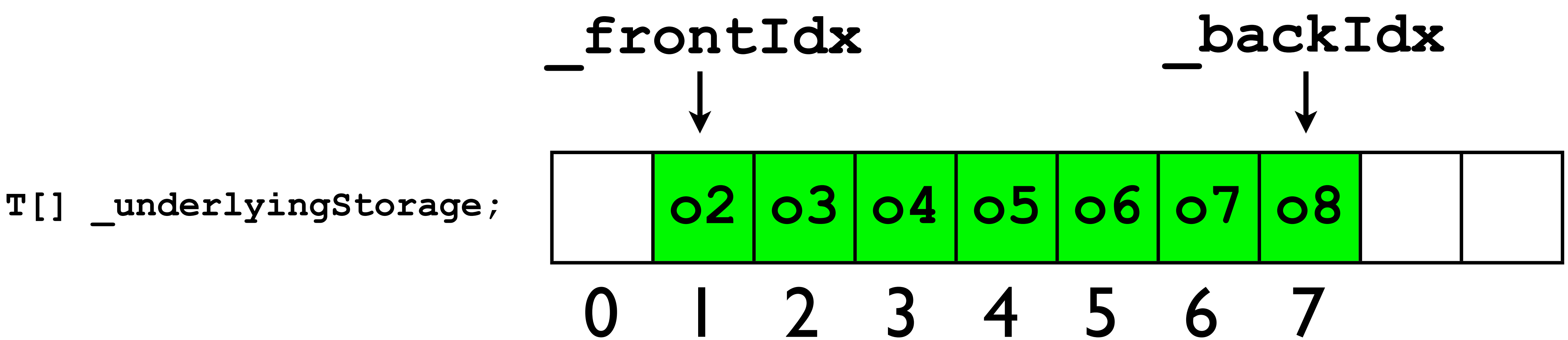
dequeue() -- Attempt #2

- Another possibility is to allocate a *huge* array for the `_underlyingStorage`, and then just keep advancing `_frontIdx` by 1 whenever `dequeue()` is called.

```
final T front = _underlyingStorage[_frontIdx];
_frontIdx++;
return front;
```

O(1) time cost

- Example: `_queue.dequeue();`



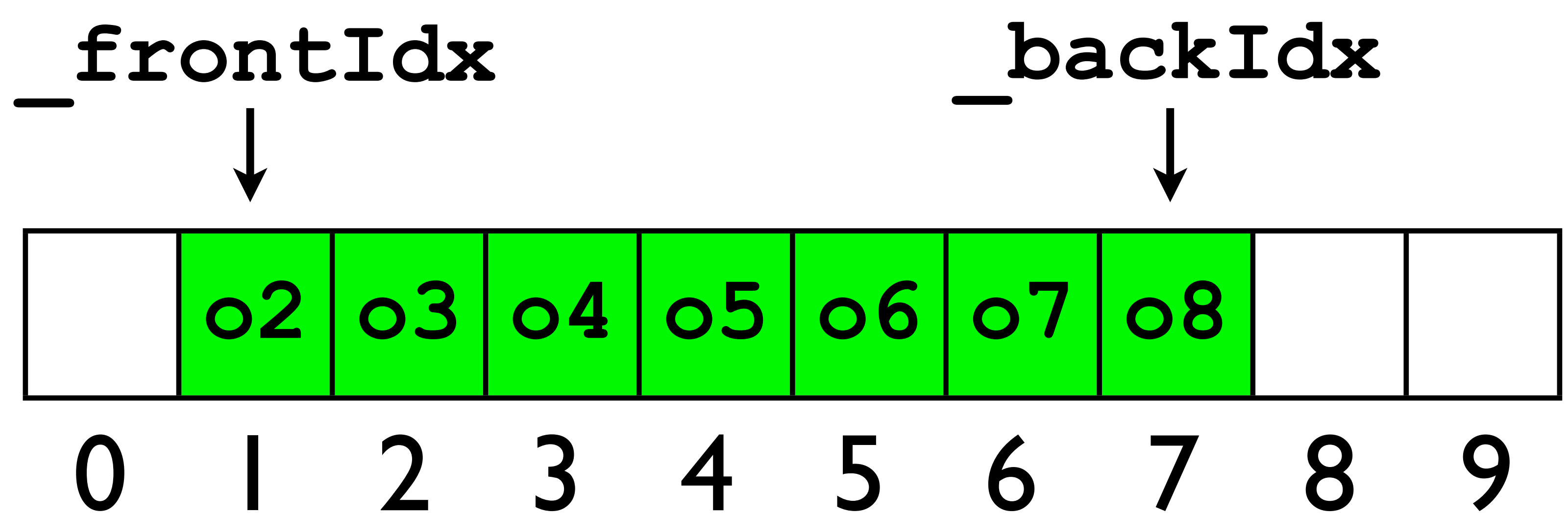
dequeue() -- Attempt #2

- Let's consider this implementation strategy when **`enqueue(o)`** and **`dequeue()`** are called many times...

```
_queue.enqueue(o9);
_queue.dequeue();
_queue.enqueue(o10);
_queue.dequeue();

...
```

`T[] _underlyingStorage;`



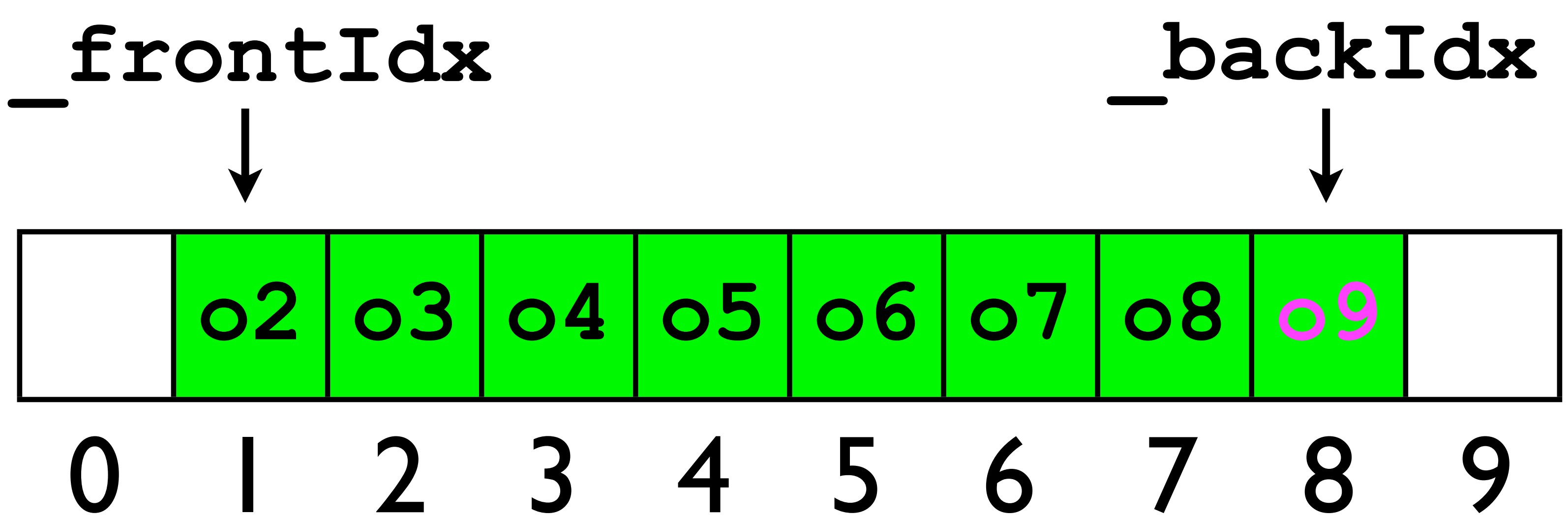
dequeue() -- Attempt #2

- Let's consider this implementation strategy when `enqueue(o)` and `dequeue()` are called many times...

```
_queue.enqueue(o9);
_queue.dequeue();
_queue.enqueue(o10);
_queue.dequeue();

...
```

`T[] _underlyingStorage;`

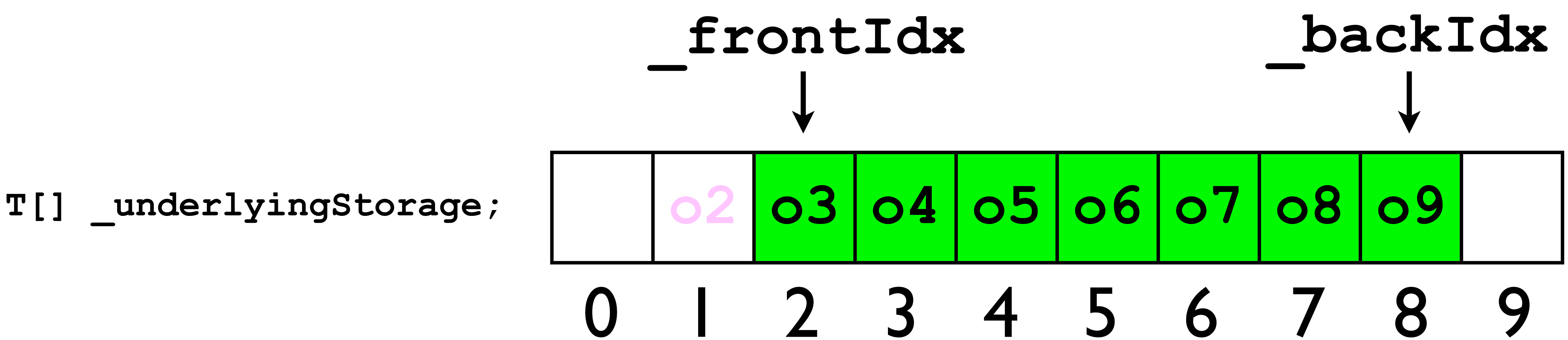


dequeue() -- Attempt #2

- Let's consider this implementation strategy when **`enqueue(o)`** and **`dequeue()`** are called many times...

```
_queue.enqueue(o9);
_queue.dequeue();
_queue.enqueue(o10);
_queue.dequeue();

...
```



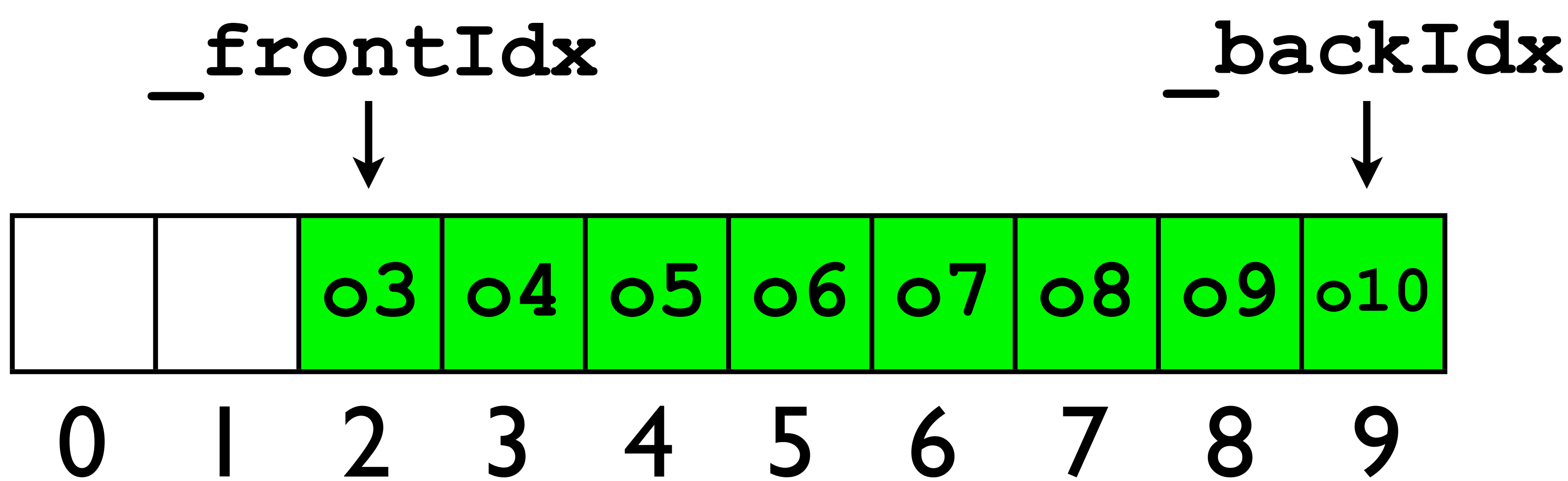
dequeue() -- Attempt #2

- Let's consider this implementation strategy when **`enqueue(o)`** and **`dequeue()`** are called many times...

```
_queue.enqueue(o9);
_queue.dequeue();
_queue.enqueue(o10);
_queue.dequeue();

...
```

`T[] _underlyingStorage;`



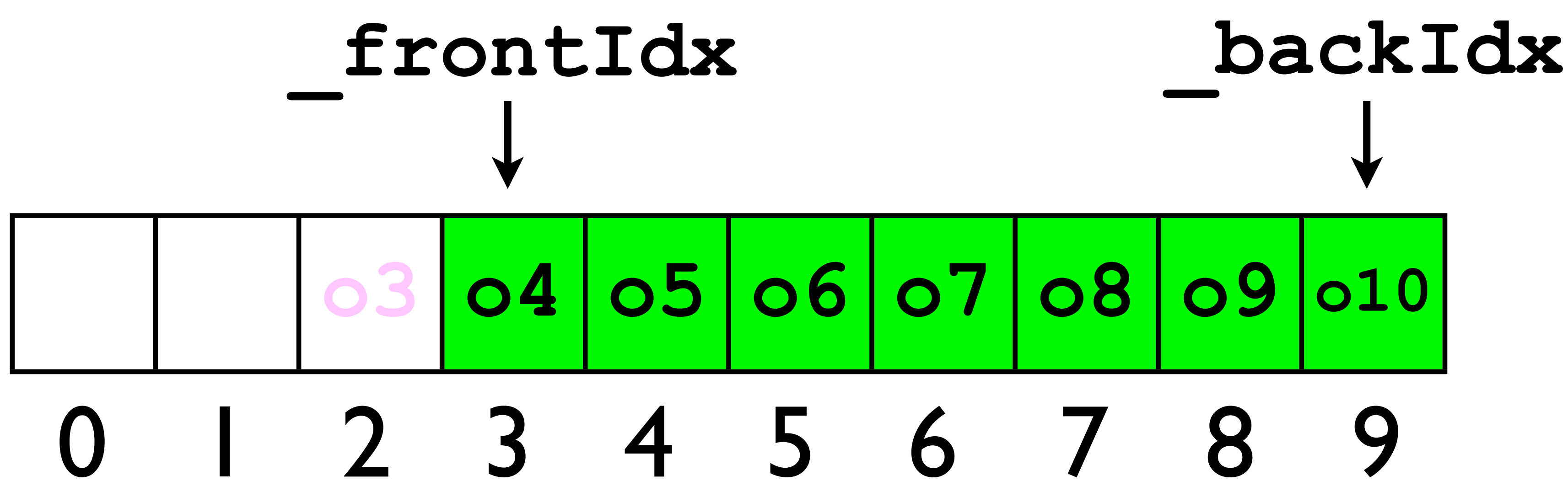
dequeue() -- Attempt #2

- Let’s consider this implementation strategy when **`enqueue(o)`** and **`dequeue()`** are called many times...

```
_queue.enqueue(o9);
_queue.dequeue();
_queue.enqueue(o10);
_queue.dequeue();

...
```

`T[] _underlyingStorage;`

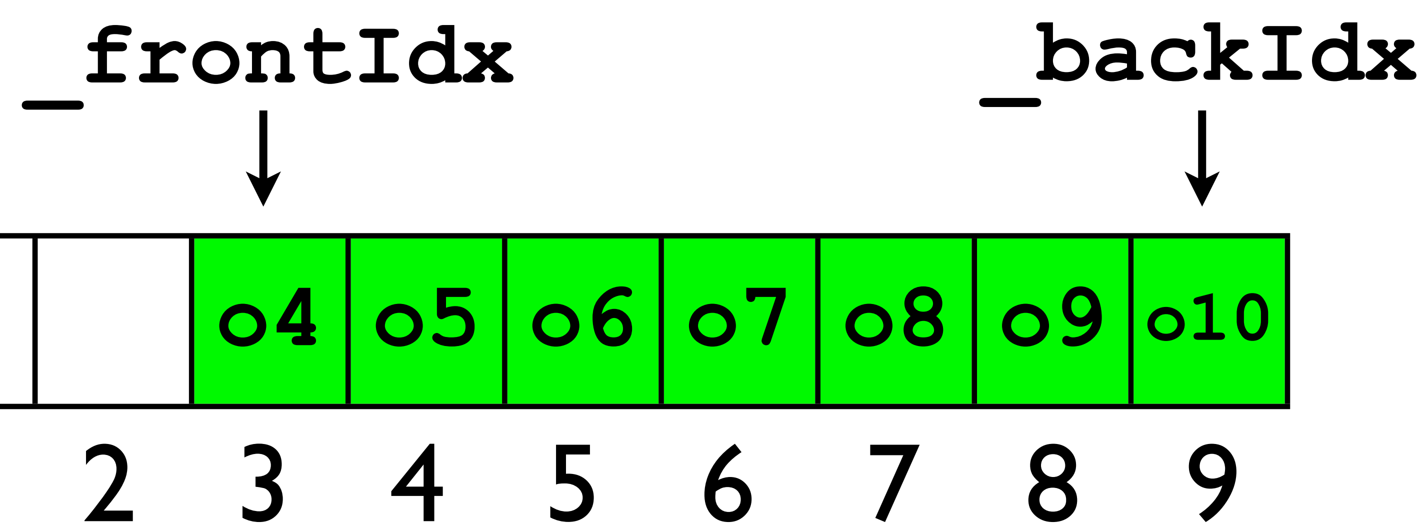


dequeue() -- Attempt #2

- Let's consider this implementation strategy when `enqueue(o)` and `dequeue()` are called many times...

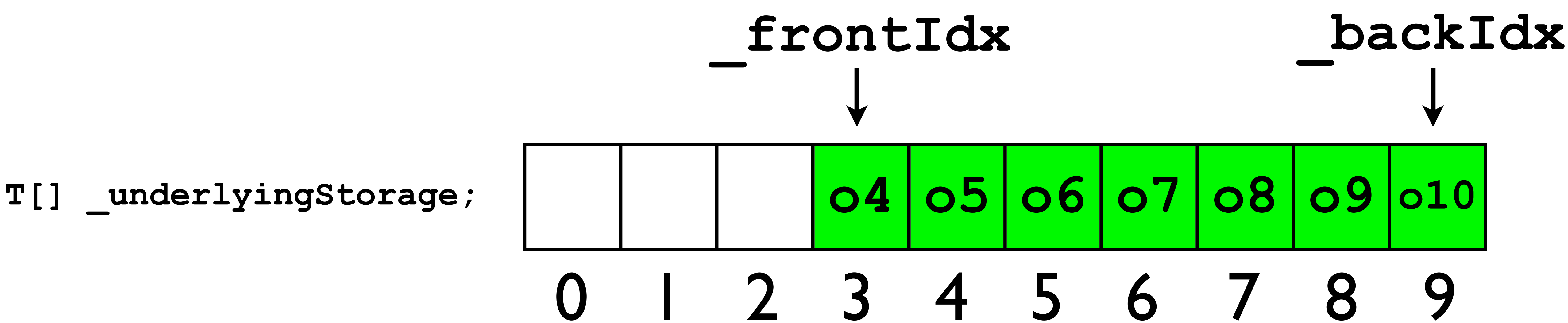
```
_queue.enqueue(o9);
_queue.dequeue();
_queue.enqueue(o10);
_queue.dequeue();
```

`T[] _underlyingStorage;`



dequeue() -- Attempt #2

- This implementation of `dequeue()` is elegant and efficient.
  - The queue keeps “moving” to the right.
  - Even though the length of the queue may be small, the array would have to be of *infinite length* to accommodate the eternal “sliding down”.

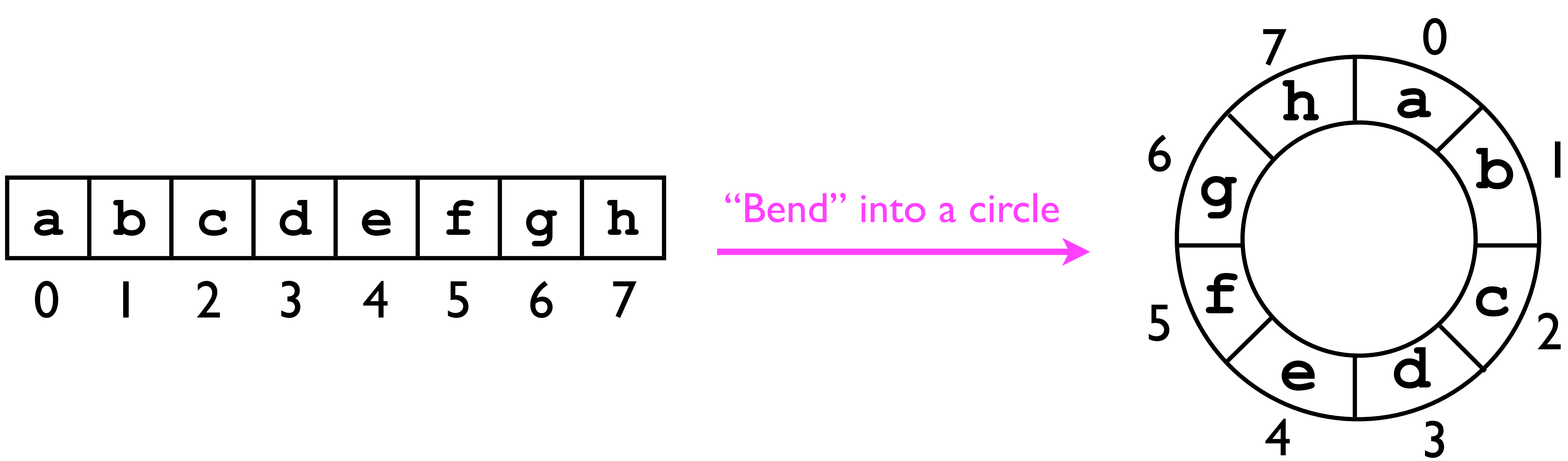


dequeue() -- Attempt #3

- Let's try one more time...
- Let's assume that the maximum length of the queue is *bounded*, i.e., it will never exceed some **`MAX_LENGTH`**.
  - Note -- in general, **`MAX_LENGTH`** and **`_underlyingStorage`** could be different.
- We can simulate an "infinite array" by implementing a *ring buffer*.
  - In a ring buffer, the back of the array is connected to the front of the array by "bending the array into a circle".

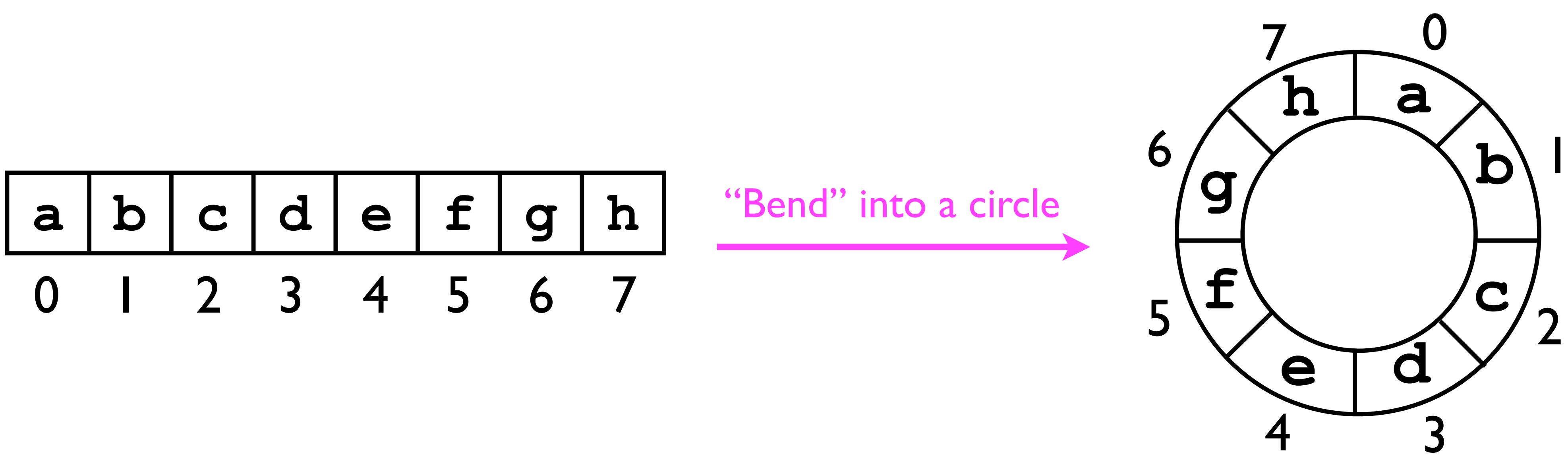
dequeue() -- Attempt #3

- Example: `T[] _ringBuffer = (T[]) new Object[8];`
- In a ring buffer, the array indices 7 and 0 are adjacent.
  - The index "before" 0 is 7.
  - The index "after" 7 is 0.



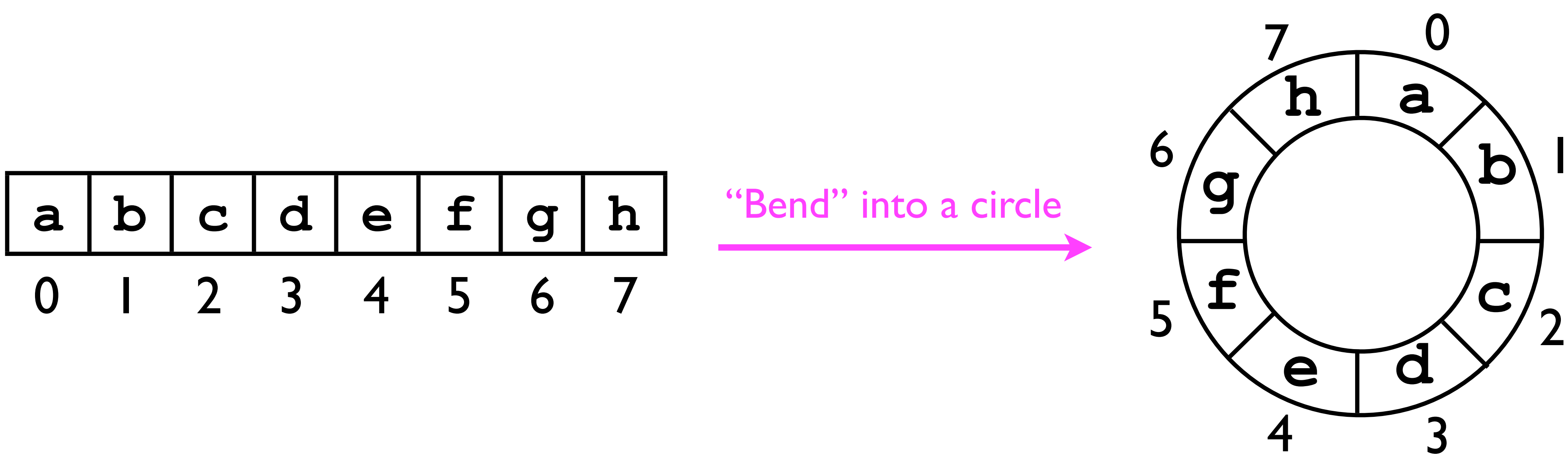
dequeue() -- Attempt #3

- A ring buffer is a convenient programming *abstraction*.
- With ring buffers, when we wish to "iterate around" the array, we can use an index variable **`currentIdx`**.
- Each time we wish to retrieve the "next" element, we return **`_ringBuffer[currentIdx];`**
- We then must "increment" **`currentIdx`**.
  - If **`currentIdx < 7`**, then: **`currentIdx++`**;
  - If **`currentIdx == 7`**, then: **`currentIdx = 0;`**



dequeue() -- Attempt #3

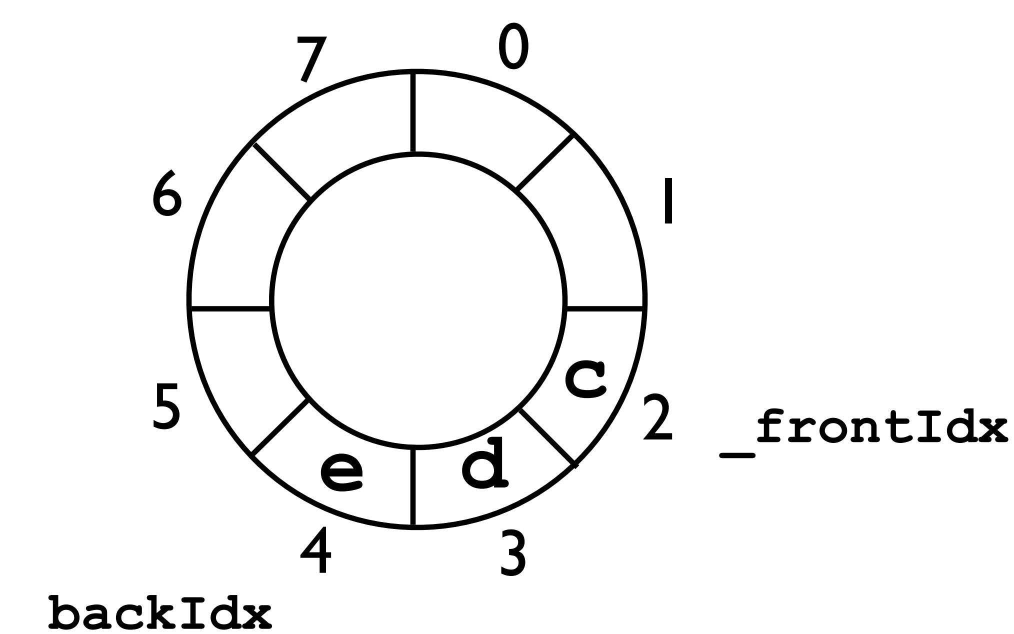
- Similar logic applies to iterating "backwards":
- Each time we wish to retrieve the "previous" element, we return `_ringBuffer[currentIdx];`
- We then must "decrement" `currentIdx`.
  - If `currentIdx > 0`, then: `currentIdx--`;
  - If `currentIdx == 0`, then: `currentIdx = 7;`



dequeue() -- Attempt #3

- Ring buffers are useful when implementing queues because they allow us to keep "moving the queue to the right" *without actually requiring infinite storage*.
- Consider the queue below (initially `_frontIdx = 2` and `_backIdx = 4`).
- We can call enqueue and dequeue repeatedly -- the queue will appear to "slide around" the ring buffer.
- As long as `dequeue()` is called frequently enough (compared to `enqueue(o)`), the ring buffer will never get full.

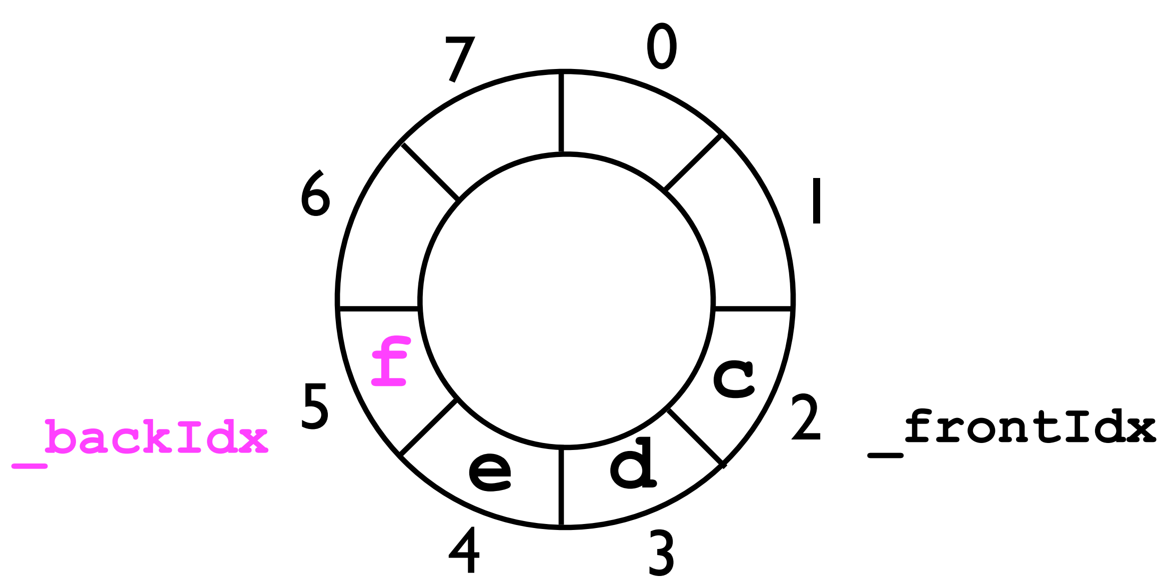
```
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```



dequeue() -- Attempt #3

- Ring buffers are useful when implementing queues because they allow us to keep "moving the queue to the right" *without actually requiring infinite storage*.
- Consider the queue below (initially `_frontIdx = 2` and `_backIdx = 4`).
- We can call enqueue and dequeue repeatedly -- the queue will appear to "slide around" the ring buffer.
- As long as **`dequeue()`** is called frequently enough (compared to **`enqueue(o)`**), the ring buffer will never get full.

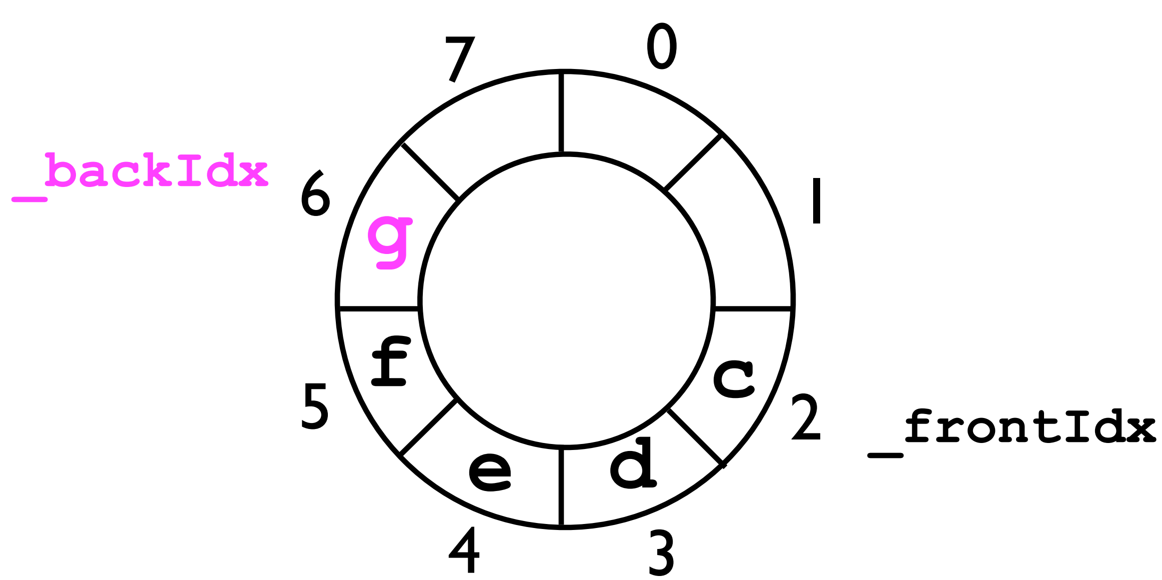
```
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```



dequeue() -- Attempt #3

- Ring buffers are useful when implementing queues because they allow us to keep "moving the queue to the right" *without actually requiring infinite storage*.
- Consider the queue below (initially `_frontIdx = 2` and `_backIdx = 4`).
- We can call enqueue and dequeue repeatedly -- the queue will appear to "slide around" the ring buffer.
- As long as `dequeue()` is called frequently enough (compared to `enqueue(o)`), the ring buffer will never get full.

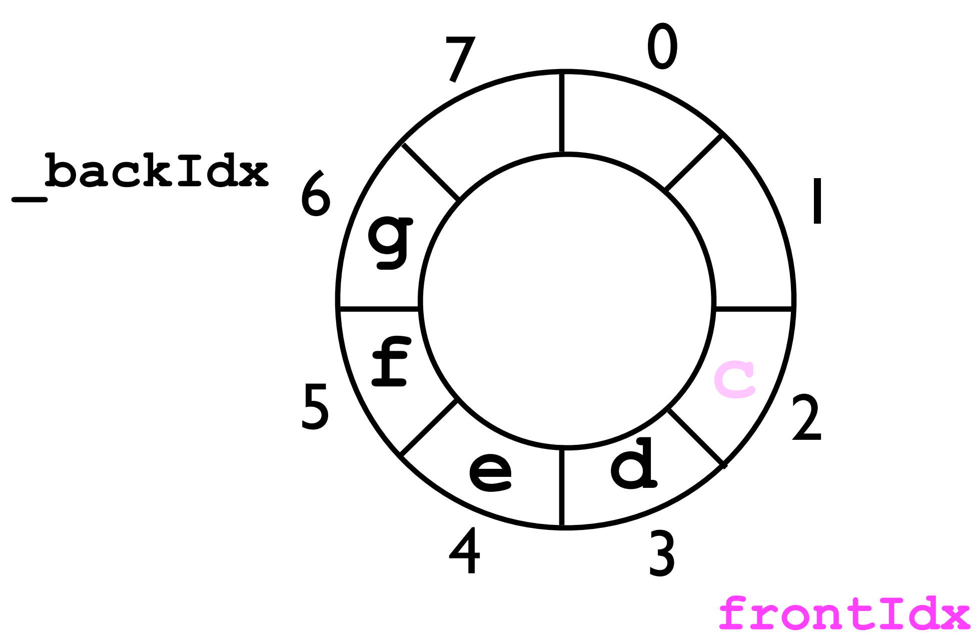
```
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```



dequeue() -- Attempt #3

- Ring buffers are useful when implementing queues because they allow us to keep "moving the queue to the right" *without actually requiring infinite storage*.
- Consider the queue below (initially `_frontIdx = 2` and `_backIdx = 4`).
- We can call enqueue and dequeue repeatedly -- the queue will appear to "slide around" the ring buffer.
- As long as `dequeue()` is called frequently enough (compared to `enqueue(o)`), the ring buffer will never get full.

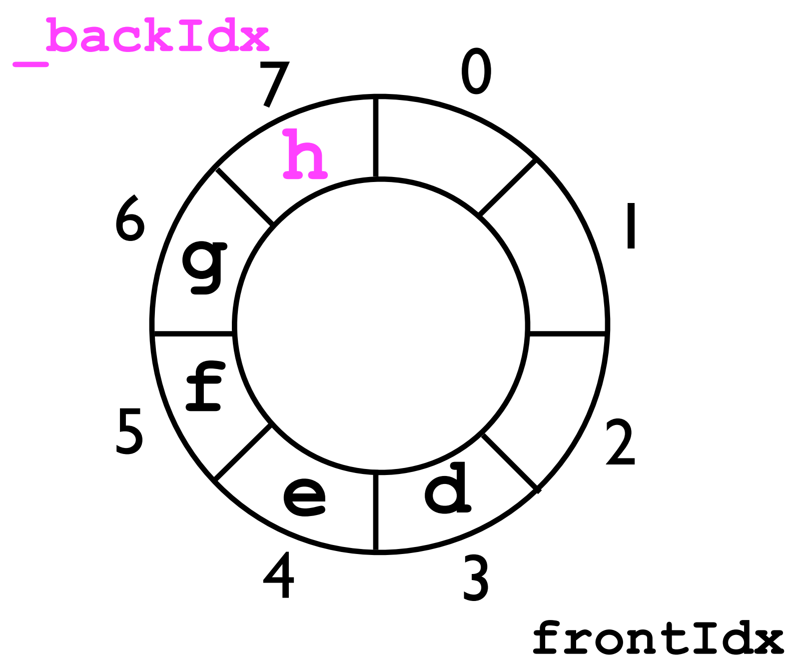
```
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```



dequeue() -- Attempt #3

- Ring buffers are useful when implementing queues because they allow us to keep "moving the queue to the right" *without actually requiring infinite storage*.
- Consider the queue below (initially `_frontIdx = 2` and `_backIdx = 4`).
- We can call enqueue and dequeue repeatedly -- the queue will appear to "slide around" the ring buffer.
- As long as `dequeue()` is called frequently enough (compared to `enqueue(o)`), the ring buffer will never get full.

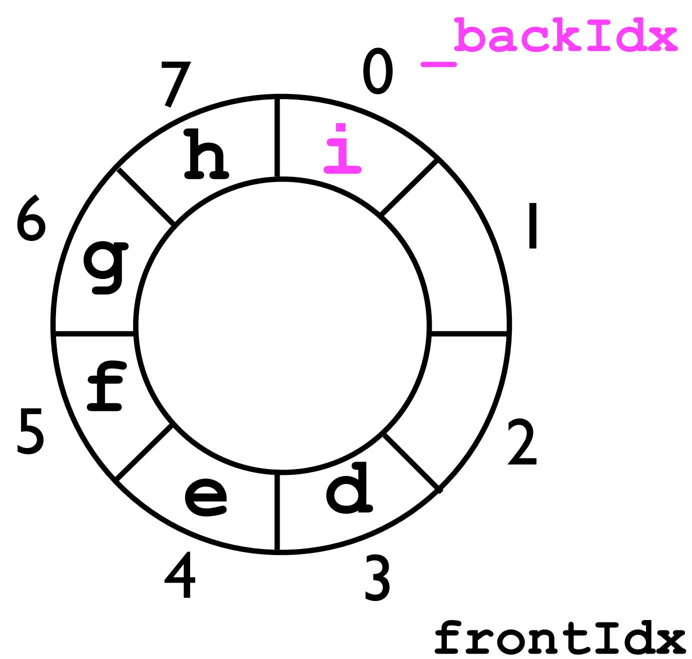
```
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```



dequeue() -- Attempt #3

- Ring buffers are useful when implementing queues because they allow us to keep "moving the queue to the right" *without actually requiring infinite storage*.
- Consider the queue below (initially `_frontIdx = 2` and `_backIdx = 4`).
- We can call enqueue and dequeue repeatedly -- the queue will appear to "slide around" the ring buffer.
- As long as `dequeue()` is called frequently enough (compared to `enqueue(o)`), the ring buffer will never get full.

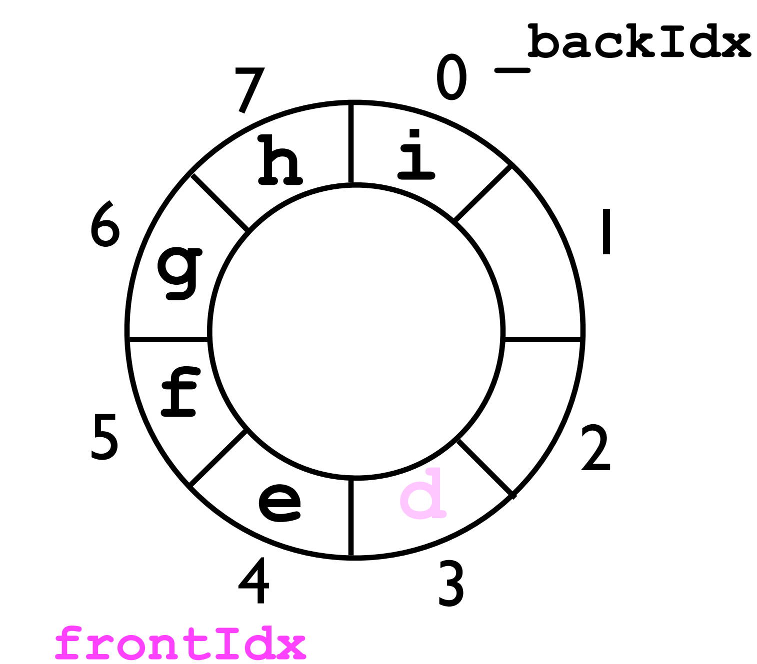
```
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```



dequeue() -- Attempt #3

- Ring buffers are useful when implementing queues because they allow us to keep "moving the queue to the right" *without actually requiring infinite storage*.
- Consider the queue below (initially `_frontIdx = 2` and `_backIdx = 4`).
- We can call enqueue and dequeue repeatedly -- the queue will appear to "slide around" the ring buffer.
- As long as `dequeue()` is called frequently enough (compared to `enqueue(o)`), the ring buffer will never get full.

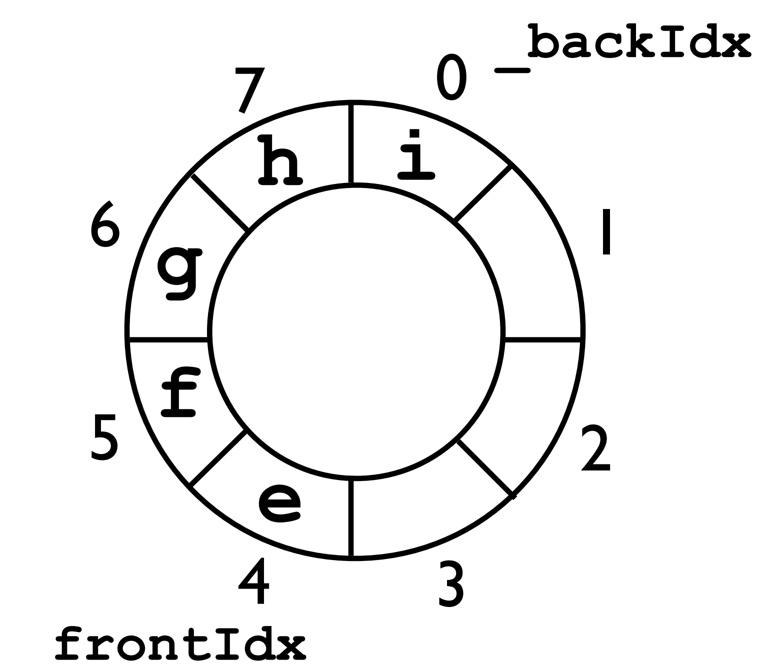
```
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```



dequeue() -- Attempt #3

- Ring buffers are useful when implementing queues because they allow us to keep "moving the queue to the right" *without actually requiring infinite storage*.
- Consider the queue below (initially `_frontIdx = 2` and `_backIdx = 4`).
- We can call enqueue and dequeue repeatedly -- the queue will appear to "slide around" the ring buffer.
- As long as `dequeue()` is called frequently enough (compared to `enqueue(o)`), the ring buffer will never get full.

```
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```



dequeue() -- Attempt #3

- Using a ring buffer as the underlying storage, a queue can be implemented so that both `enqueue(o)` and `dequeue()` have time cost *O*(1).
- The disadvantage compared to a linked list-based implementation is that the maximum length of the queue must be known in advance.
  - When the queue is "full" and the user calls `enqueue(o)`, then either:
    - The queue will **block** -- hang until some other program/thread calls dequeue; or
    - Throw an exception.
  - With linked lists, the queue can grow arbitrarily long.