CSE 12: Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Seven
16 July 2012
Stacks and queues.
Stacks and queues.

- Let’s now bring in two more fundamental data structures into the course.

- So far we have covered lists -- array-based lists and linked-lists.

- These are both linear data structures -- each element in the container has at most one successor and one predecessor.

- Lists are most frequently used when we wish to store objects in a container, and probably never remove them from it.

- E.g., if Amazon uses a list to store its huge collection of customers, it has no intention of “removing” a customer (except at program termination).
Stacks and queues

- Stacks and queues, on the other hand, are examples of linear data structures in which every object inserted into it will generally be removed:
  - The stack/queue is intended only as “temporary” storage.
  - Both stacks and queues allow the user to add and remove elements.
  - Where they differ is the order in which elements are removed relative to when they were added.
Stacks.
Stacks

- Stacks are *last-in-first-out* (LIFO) data structures.
- The classic analogy for a “stack” is a pile of dishes:
  - Suppose you’ve already added dishes A, B, and C to the “stack” of dishes.

If you try to remove a middle dish, you get that annoying clanging sound.
Stacks

- Stacks are *last-in-first-out* (LIFO) data structures.
- The classic analogy for a “stack” is a pile of dishes:
  - Suppose you’ve already added dishes A, B, and C to the “stack” of dishes.
  - Now you add one more, D.

If you try to remove a middle dish, you get that annoying clanging sound.
Stacks

• Stacks are *last-in-first-out* (LIFO) data structures.

• The classic analogy for a “stack” is a pile of dishes:
  • Suppose you’ve already added dishes A, B, and C to the “stack” of dishes.
  • Now you add one more, D.
  • Now you remove one dish -- you get *D* back.

If you try to remove a middle dish, you get that annoying clanging sound.
Stacks

• Stacks are *last-in-first-out* (LIFO) data structures.

• The classic analogy for a “stack” is a pile of dishes:

  • Suppose you’ve already added dishes A, B, and C to the “stack” of dishes.
  
  • Now you add one more, D.
  
  • Now you remove one dish -- *you get D back*.
  
  • If you remove another, you get C, and so on.

• With stacks, you can only add to/remove from the *top* of the stack.

If you try to remove a middle dish, you get that annoying clanging sound.
Usage example of stacks

```java
Stack<String> stack = new Stack<String>();
stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");
...
String s;
s = stack.pop(); // returns "d"
```

**push** adds an object to the stack

**pop** both gets and removes the “last” object from the stack
Usage example of stacks

```java
Stack<String> stack = new Stack<String>();
stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");
...
String s;
s = stack.pop(); // returns "d"
s = stack.pop(); // returns "c"
```

push adds an object to the stack

pop both gets and removes the “last” object from the stack
Usage example of stacks

```java
Stack<String> stack = new Stack<String>();
stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");

push adds an object to the stack

... String s;
s = stack.pop(); // returns "d"
s = stack.pop(); // returns "c"
s = stack.pop(); // returns "b"

pop both gets and removes the "last" object from the stack
```
Usage example of stacks

```java
Stack<String> stack = new Stack<String>();
stack.push("a");
stack.push("b");
stack.push("c");
stack.push("d");

String s;
s = stack.pop(); // returns "d"
s = stack.pop(); // returns "c"
s = stack.pop(); // returns "b"
s = stack.pop(); // returns "a"
```

push adds an object to the stack

pop both gets and removes the “last” object from the stack
Stacks

- Stacks find many uses in computer science, e.g.:
  - Implementing procedure calls.

- Consider the following code:
  ```java
  void f () {
    _num = 4;
    g();
    _num++;
  }
  void g () {
    h();
    _num = 7;
  }
  void h () {
    System.out.println("Yup!");
  }
  ```
  How does the CPU know to “jump” from f to g, g to h, then h back to g, and finally g back to f?
Von Neumann machine

- On all modern machines, a program’s instructions and its data are stored together somewhere in the computer’s long sequence of bits (Von Neumann architecture).

- Just by “glancing” at the contents of computer memory, one would have no idea whether a certain byte contains code or data -- it’s all just bits.

- To keep track of which instruction in memory is currently being executed, the CPU maintains an Instruction Pointer (IP).
Code execution

Address          Memory

0
4
8
12
16
20
24
28
32
36
40
...

• Suppose the IP is 8:
  • Then the next instruction to execute is \_num=4;
  • The CPU then advances the IP to the next instruction (4 bytes later) to 12.

Monday, July 16, 12
Code execution

Address  Memory

0           4
4           _num=4;
8           call g();
12          _num++;
16          return;
20          
24          call h();
28          _num = 7;
32          return;
36          ...println("yup!");
40          return;
...

- The next instruction is call g().
- The CPU must now “move” the IP to address 24 (start of g’s code) so g can start.
• g has now started.
• The first thing g does is call h.
• We have to move the IP again.
Code execution

Address | Memory
---|---
0 | num=4
4 | call g();
8 | _num++;
12 | return;
16 | call h();
20 | _num = 7;
24 | return;
28 | ...println("yup!");
32 | return;
36 | ...println("yup!");
40 | return;
...

- h now prints out “yup!”.
• The return instructions tells the CPU to move the IP back to where it was before the current method was called.

• But where is that?
### Code execution

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>_num=4;</td>
</tr>
<tr>
<td>12</td>
<td>call g();</td>
</tr>
<tr>
<td>16</td>
<td>_num++;</td>
</tr>
<tr>
<td>20</td>
<td>return;</td>
</tr>
<tr>
<td>24</td>
<td>call h();</td>
</tr>
<tr>
<td>28</td>
<td>_num = 7;</td>
</tr>
<tr>
<td>32</td>
<td>return;</td>
</tr>
<tr>
<td>36</td>
<td>...println(&quot;yup!&quot;);</td>
</tr>
<tr>
<td>40</td>
<td>return;</td>
</tr>
</tbody>
</table>

- The return call at address 40 *should* cause the CPU to jump to address 28 -- the next instruction in g.
Code execution

- We then execute `_num=7;`
Code execution

• And now we have to return to where the caller of g left off (address 16).
### Code execution

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>_num=4;</td>
</tr>
<tr>
<td>12</td>
<td>call g();</td>
</tr>
<tr>
<td>16</td>
<td>_num++;</td>
</tr>
<tr>
<td>20</td>
<td>return;</td>
</tr>
<tr>
<td>24</td>
<td>call h();</td>
</tr>
<tr>
<td>28</td>
<td>_num = 7;</td>
</tr>
<tr>
<td>32</td>
<td>return;</td>
</tr>
<tr>
<td>36</td>
<td>...println(&quot;yup!&quot;);</td>
</tr>
<tr>
<td>40</td>
<td>return;</td>
</tr>
</tbody>
</table>

- How does the CPU know which address to “return” to?
- We need some kind of data structure to manage the “return addresses” for us.
What we need is a last-in-first-out data structure ("stack") to remember all the return addresses:

- **Rule 1**: Before method x calls method y, method x first adds its "return address" to the stack.
- **Rule 2**: When method y "returns" to its caller, it removes the top of the stack and sets the IP to that address.

Let’s see this work in practice...
Code execution

- “Return address” stack:
Code execution

- "Return address" stack:

```plaintext
num

16
(bottom of stack)

"push" 16 onto stack
```
Code execution

• “Return address” stack:

“push” 28 onto stack

28
16
(bottom of stack)
Code execution

Address | Memory
--- | ---
0 | ![Code execution diagram](diagram.png)
4 | _num=4;
8 | call g();
12 | _num++; return;
16 | call h();
20 | _num = 7; return;
24 | call h();
28 | println("yup!");
32 | return;
36 | ...
40 | ...
...

- "Return address" stack:

28
16

(bottom of stack)
Code execution

- “Return address” stack:

  “pop” 28 off the stack...

  28
  16
  (bottom of stack)
Code execution

- "Return address" stack:
  ...and jump to that address.

- Memory
  - Address
    - 0: _num=4;
    - 4: call g();
    - 8: _num++;
    - 12: return;
    - 16: call h();
    - 20: _num = 7;
    - 24: return;
    - 28: println("yup!");
    - 32: return;
    - 36: ...
    - 40: return;

- IP (bottom of stack)
Code execution

- "Return address" stack:

  "pop" 16 off the stack...

  (bottom of stack)
Code execution

Address  Memory

0    7
4
8    _num=4;
call g();
12
16    _num++;
20    return;
24    call h();
28    _num = 7;
32    return;
36    ...println("yup!");
40    return;
...

• “Return address” stack:

...and jump to that address.

(bottom of stack)
Stack ADT

• To support the last-in-first-out adding/removal of elements, a stack must adhere to the following interface:

```java
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 ();

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

```java
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.

```java
Stack stack;
...
stack.push(y);
stack.push(z);
stack.push(w);
```

```
T[] _underlyingStorage;
_numElements: 7
```

```
0
```

```
_stack.push(y);
_stack.push(z);
_stack.push(w);
```
Array-based stacks

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

```
T[] _underlyingStorage;

_numElements: 8
```

```plaintext
0  a  b  c  x  y  y  z  w  q
```

Bottom  Top

_numElements - 1
Array-based stacks

• In every call to peek():
  • The element stored at index \texttt{numElements} - 1 is saved to a local variable \texttt{top}.
  • \texttt{top} is returned.
Array-based stacks

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

```plaintext
T[] _underlyingStorage

Array-based stack

a b c
int _numElements: 3

Linked list-based stack

Node a <-> Node b <-> Node c
_null

_bottom or _head
_top or _tail
```
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.:

```java
_stack.push(d);
```
Linked list-based stacks

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

```java
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 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 “converts” from the interface of one ADT -- the one we’re trying to implement -- to 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 DoublyLinkedListList12) to satisfy the Stack ADT interface specification...
Stack as adaptation of linked list

class StackImpl<T> implements Stack<T> {
    private DoublyLinkedList _list;
    StackImpl() {
        _list = new DoublyLinkedList();
    }

    void push(T o) {
        _list.addToBack(o);
    }

    T pop() {
        return _list.removeBack();
    }
    ...
}
Queues.
Queues

• Queues are a first-in-first-out (FIFO) data structure used typically for temporary data storage.

• Instead of add, get, and remove methods, queues offer enqueue and dequeue methods.

• The first object to be enqueued is the first object to be dequeued.

• Similarly to a train entering a tunnel, the first car to enter the tunnel is the first car to exit the tunnel.
Usage example of queues

Queue<String> queue = new Queue<String>();
queue.enqueue("a");
queue.enqueue("b");
queue.enqueue("c");
queue.enqueue("d");

String s;
s = queue.dequeue();  // returns "a"
s = queue.dequeue();  // returns "b"
...
Queue example

• Consider enrollment lists for a UCSD course. Suppose max enrollment = 80:

    class Course {
        private static final int MAX_ENROLLMENT = 80;
        private List<Student> _enrolledStudents;
        private Queue<Student> _waitingList;
        ...
        boolean enroll (Student s) {
            ...
        }
        void addToWaitingList (Student s) {
            ...
        }
        void drop (Student s) {
            ...
        }
    }
Queue example

- A student can enroll only if course size is less than max enrollment:

```java
boolean enroll (Student s) {
    if (_enrolledStudents.size() == MAX_ENROLLMENT) {
        return false;  // course full -- can’t enroll!
    }
    _enrolledStudents.add(s);
}
```
Queue example

• If course is full, students can place their name on a waiting list:

```java
void addToWaitingList (Student s) {
    _waitingList.enqueue(s);
}
```
Queue example

- If a student drops the course, then we can enroll a student from the waiting list:

```java
void drop (Student s) {
    _list.remove(s);
    if (_waitingList.size() > 0) {
        _enrolledStudents.add(_waitingList.dequeue());
    }
}
```

The Queue interface ensures that the first Student to be dequeued is always the first student who enqueued.
Queue ADT

• The interface for a Queue ADT looks as follows:

```java
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;

    // Returns number of elements in queue
    int size ();
}
```
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 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:

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

```c
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T[] _underlyingStorage;</td>
<td>o1</td>
<td>o2</td>
<td>o3</td>
<td>o4</td>
<td>o5</td>
<td>o6</td>
<td>o7</td>
</tr>
<tr>
<td>_frontIdx</td>
<td>_backIdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
```
Array-based queue

- `enqueue(o)`: Append to the back of the array:
  - This is easy:
    ```
    _backIdx++; 
    _underlyingStorage[_backIdx] = o;
    ```
  - Example: `queue.enqueue(o8);`

```
_T[] _underlyingStorage;
_ frontIdx
↓
  0 1 2 3 4 5 6 7
_ backIdx
↓
  o1 o2 o3 o4 o5 o6 o7 o8
```
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...

```c
T[] _underlyingStorage;
_frontIdx
_0
_1
_2
_3
_4
_5
_6
_7
_backIdx
```

Monday, July 16, 12
dequeue() -- Attempt #1

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

```java
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;
```
• One possibility is to “shift down” by 1 the entire queue after the front has been removed:

```java
definal 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();`

![Diagram showing dequeue operation](image)
dequeue() -- Attempt #1

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

```java
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;
```

- Example: `queue.dequeue();`

```
T[] _underlyingStorage;
02 03 04 05 06 07 08
_0 _frontIdx  _frontIdx
_1 _backIdx
```

_FrontIdx_ never changes -- always 1!
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.

```java
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.

```java
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.

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

- Example: `
  _queue.dequeue();
`
enqueue() -- Attempt #2

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

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

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

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

```plaintext
_T[] _underlyingStorage;
_ frontIdx
0 1 2 3 4 5 6 7 8 9

  o2  o3  o4  o5  o6  o7  o8  o9
  _backIdx
```
Let’s consider this implementation strategy when `enqueue(o)` and `dequeue()` are called many times...

```c
_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...

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

```
_T[] _underlyingStorage;
_frontIdx

_o3 o4 o5 o6 o7 o8 o9 o10

_backIdx
```

Monday, July 16, 12
Let’s consider this implementation strategy when `enqueue(o)` and `dequeue()` are called many times...

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

```plaintext
T[] _underlyingStorage;
```

![Diagram](image)
Let's consider this implementation strategy when `enqueue(o)` and `dequeue()` are called many times...

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

![Diagram showing the dequeue operation and indices]
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”.

```
T[] _underlyingStorage;
  0 1 2 3 4 5 6 7 8 9
  _frontIdx \downarrow
  _backIdx \downarrow
```

Monday, July 16, 12
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 \texttt{MAX\_LENGTH}.

  • Note -- in general, \texttt{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”.

Monday, July 16, 12
deque() -- 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.
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;`
Similar logic applies to iterating “backwards”:

Each time we wish to retrieve the “previous” element, we return
\(_\text{ringBuffer}[\text{currentIdx}]\);

We then must “decrement” \text{currentIdx}.

1. If \text{currentIdx} > 0, then: \text{currentIdx}--;
2. If \text{currentIdx} == 0, then: \text{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.

```plaintext
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```
• 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.

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

```
    7 0 1
   6 f c
   5 2 _backIdx
   4 e d
   3 _frontIdx
```
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();
```
**enqueue() -- 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.

```plaintext
enqueue(f);
enqueue(g);
dqueue();
enqueue(h);
enqueue(i);
dqueue();
```
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();
```
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()), 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 \texttt{dequeue()} is called frequently enough (compared to \texttt{enqueue(o)}), the ring buffer will never get full.

```plaintext
enqueue(f);
enqueue(g);
dequeue();
enqueue(h);
enqueue(i);
dequeue();
```
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 \texttt{dequeue()} is called frequently enough (compared to \texttt{enqueue(o)}), the ring buffer will never get full.

\begin{verbatim}
enqueue(f);  
enqueue(g);  
dequeue();   
enqueue(h);  
enqueue(i);  
dequeue();  
\end{verbatim}
deque() -- Attempt #3

• Using a ring buffer as the underlying storage, a queue can be implemented so that both enqueue(o) and dequeue() operate efficiently.

• 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.