Queues

Intro and Implementations

Queues

Like stacks, queues are simple but useful data structures

- they are ordered containers where entries can only be inserted at one end (the rear or end of the queue) and removed from another (called the front)

- a queue is a first-in, first-out (FIFO data structure; entries are removed from the stack in the reverse order of their insertion

Picturing a queue is easy (when stealing images from Google):

front of the queue

end of the queue

The STL queue

The STL has a queue class you can use

- like other STL containers, it is a template class and can contain items of any type
- we'll later see several different implementations of queue classes

Member functions

(constructor)	Construct queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
front	Access next element (public member function)
back	Access last element (public member function)
push	Insert element (public member function)
pop	Delete next element (public member function)

The STL stack

What does the following code output?

```
string text = "Data structures";
queue<char> letters;

for (size_t i = 0; i < text.length(); i++) {
    letters.push( text[i] );
}

while (!letters.empty()) {
    cout << letters.front();
    letters.pop();
}
```

Queue Implementation: Array Version

Specification

Queue: Array Version

Template parameters, typedefs, and member constants:

```
// template parameter
template <typename Item>


// Item is the type of items in the queue, and must be
// a built-in type, or a class that provides:
//   - instantiation via a default constructor
//   - instantiation via a copy constructor
//   - assignment operator   (x = y)
```

Queue: Array Version

Template parameters, typedefs, and member constants:

```
// alias for the template parameter
typedef Item value_type;
```

Queue: Array Version

Template parameters, typedefs, and member constants:

```
// data type of variables that track the queue's size
typedef std::size_t size_type;
```

Queue: Array Version

Template parameters, typedefs, and member constants:

```
// the maximum capacity for any queue
static const size_type CAPACITY = 30;

// queue<Item>::CAPACITY is the maximum capacity of
// any queue; once CAPACITY is reached, further pushes
// (enqueues) are forbidden
```

Queue: Array Version

Constructors:

```
// creates an empty queue
queue();

// postcondition:
//   the queue has been initialized as an empty queue
```

Queue: Array Version

Modification member functions:

```
// adds @entry to the rear of the queue
void push(const Item& entry);

// precondition:
//   size() < CAPACITY
// postcondition:
//   A new copy of @entry has been added to the rear of
//   the queue
```

Queue: Array Version

Modification member functions:

```
// removes the first item in the queue
void pop();

// precondition:
//   size() > 0
// postcondition:
//   The top (first) item in the queue has been removed.
```

Queue: Array Version

Constant member functions:

```
// returns the front item in the queue
Item front() const;

// precondition:
//   size() > 0
// postcondition:
//   The return value is the first (top) item of the
//   queue, but the queue is unchanged. This differs
//   slightly from the STL queue, where the front
//   function returns by reference.
```

Queue: Array Version

Constant member functions:

```
// returns the total number of items in the queue
size_type size() const;

// postcondition:
//   The return value is the total number of items in
//   the queue
```

Queue: Array Version

Constant member functions:

```
// returns true if the queue is empty, false otherwise
bool empty() const;

// postcondition:
//   The return value is true if the queue is empty, and
//   false otherwise
```

Queue: Array Version

Value semantics:

```
// queue<Item> objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

Queue Implementation: Array Version

Implementation

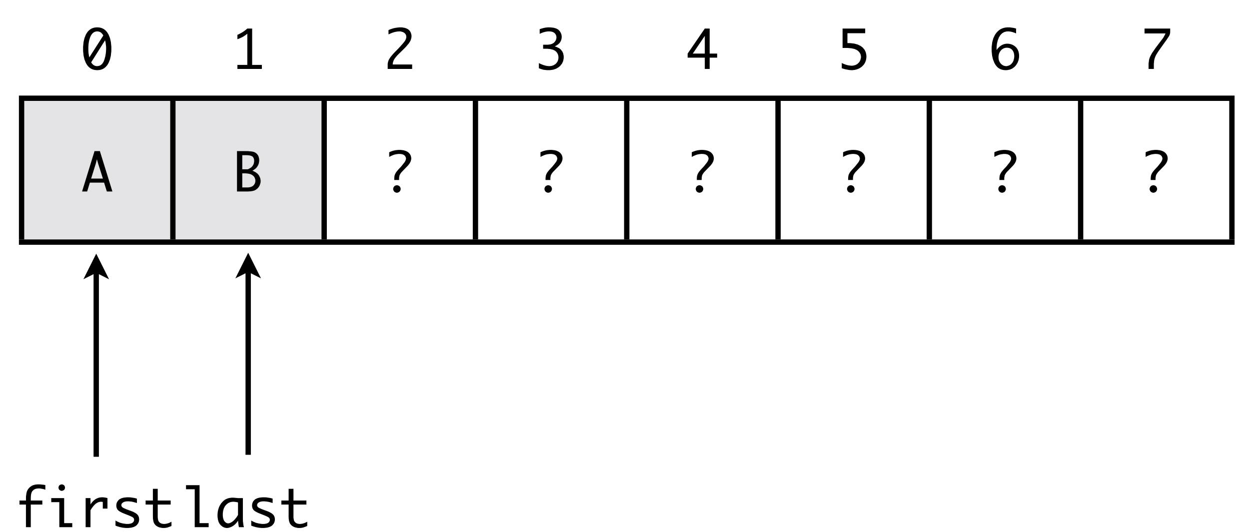
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`



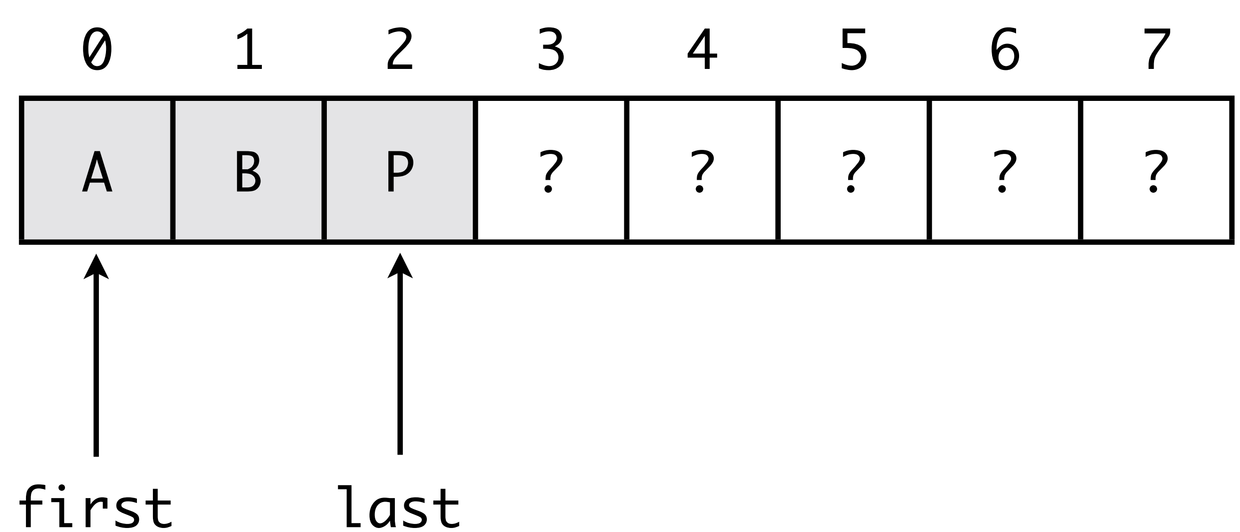
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`




```
queue.push('P');
```

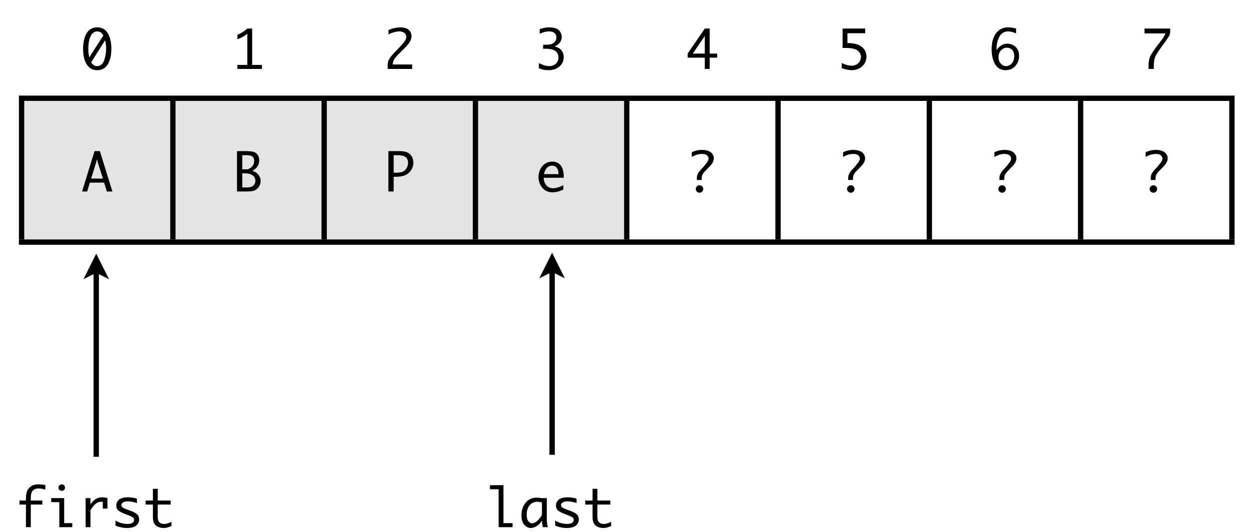
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`




```
queue.push('e');
```

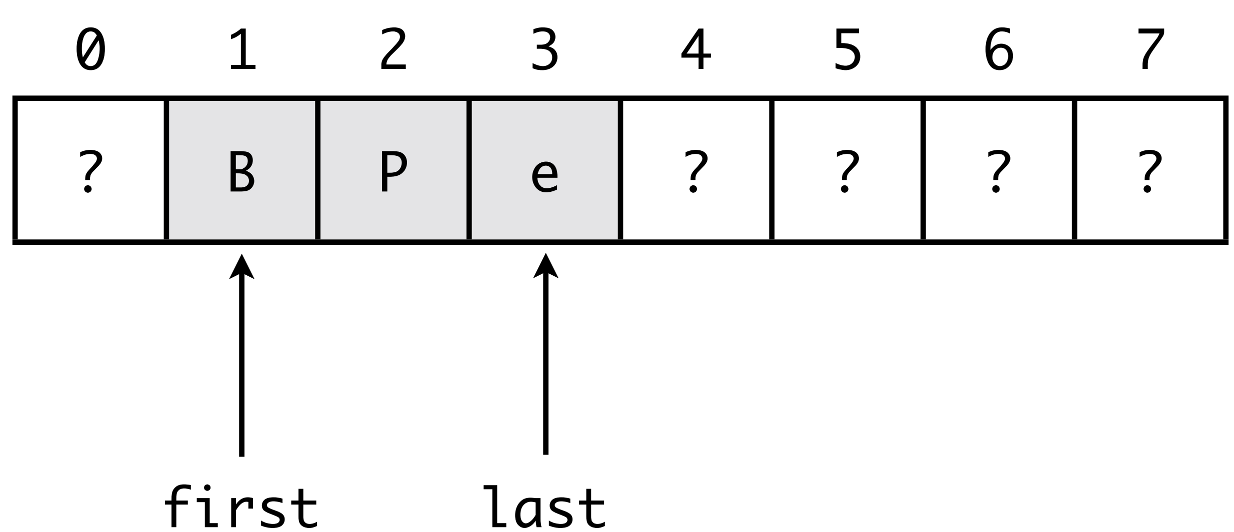
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as “circular”...

- we keep track of two indexes: `first` and `last`




```
queue.pop();
```

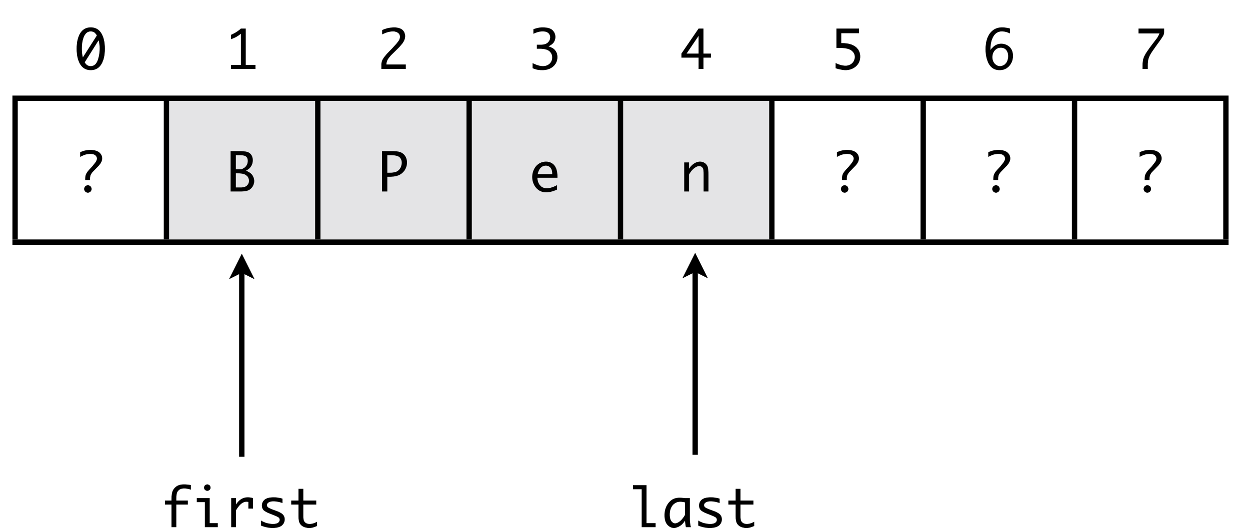
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`




```
queue.push('n');
```

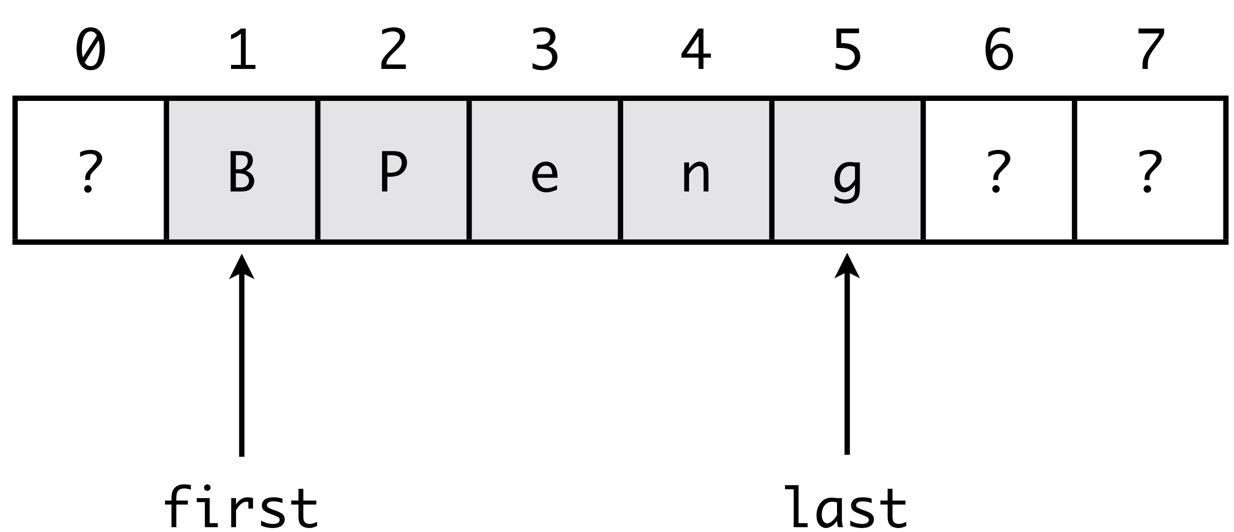
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`




```
queue.push('g');
```

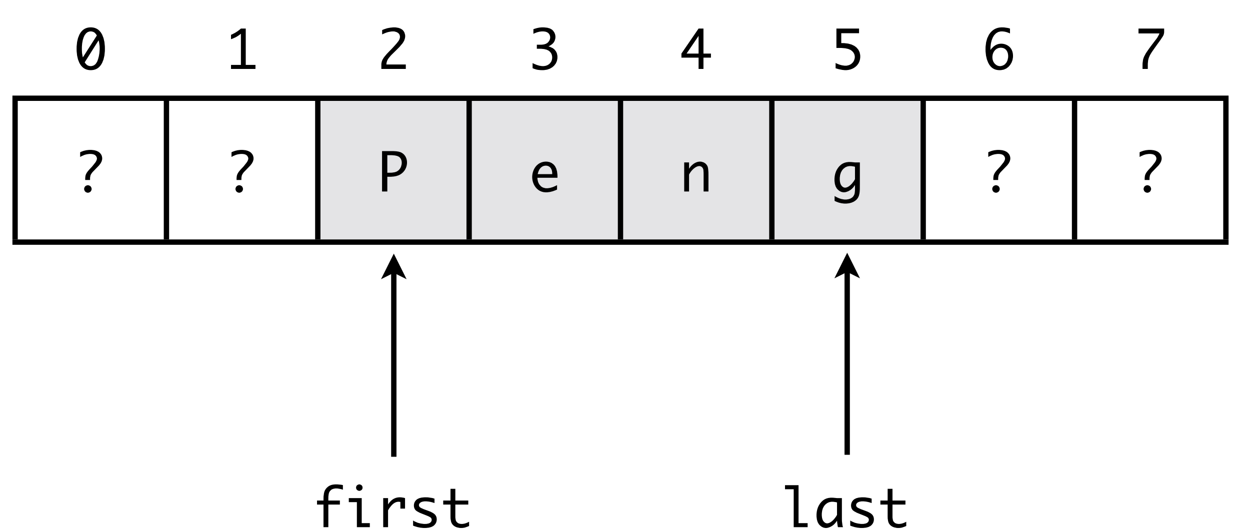
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`



`queue.pop();`

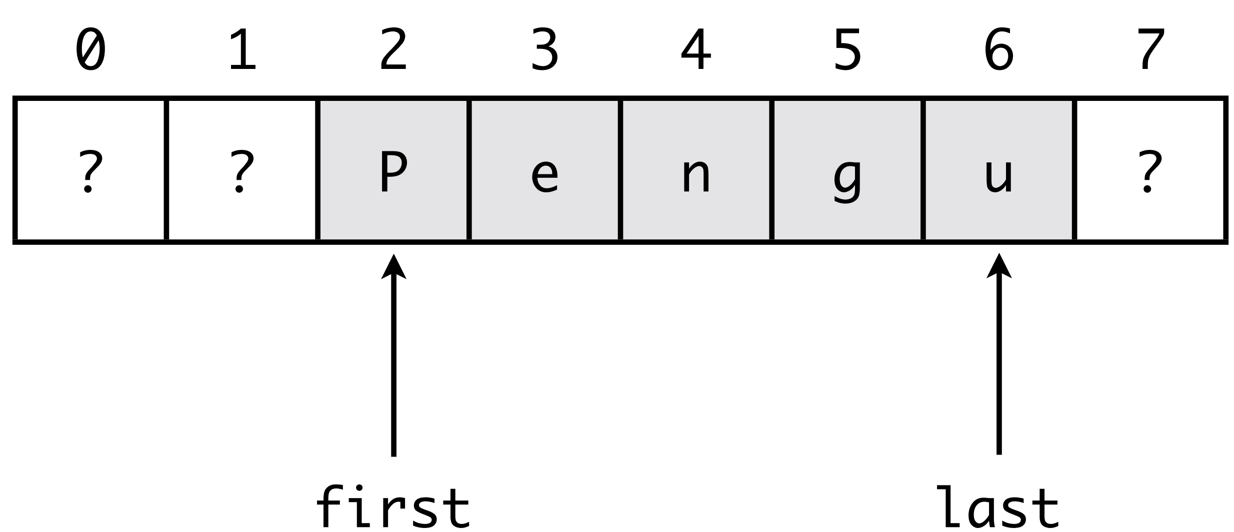
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`




```
queue.push('u');
```

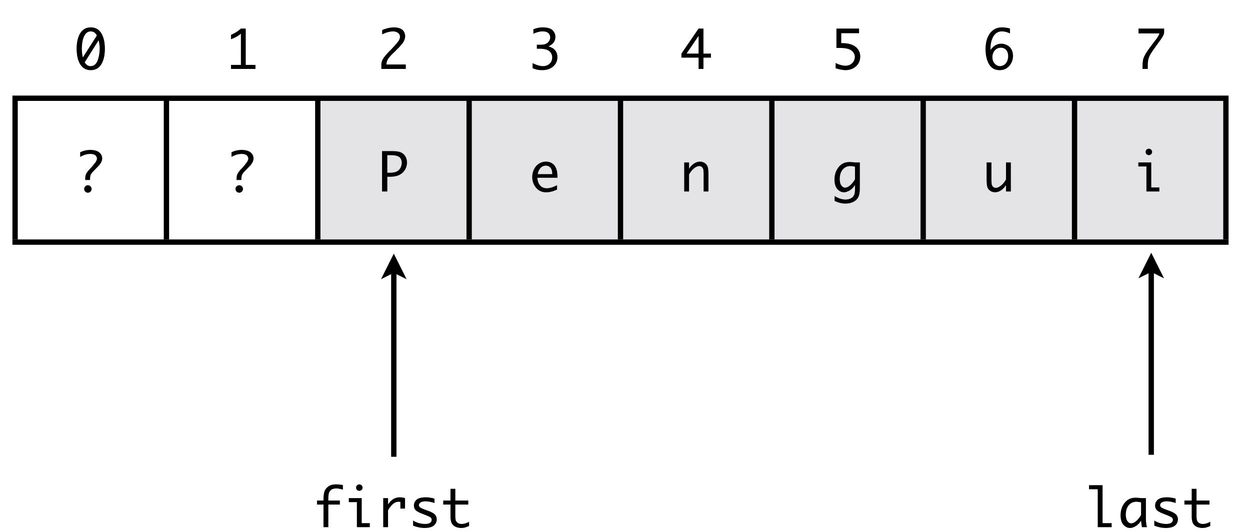
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`




```
queue.push('i');
```

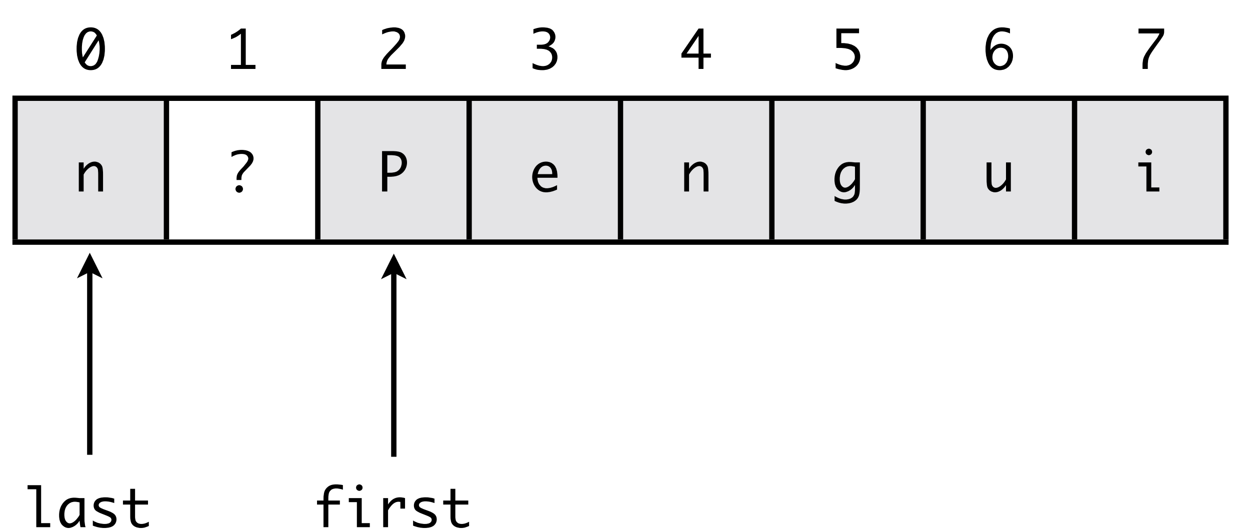
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we keep track of two indexes: `first` and `last`




```
queue.push('n');
```

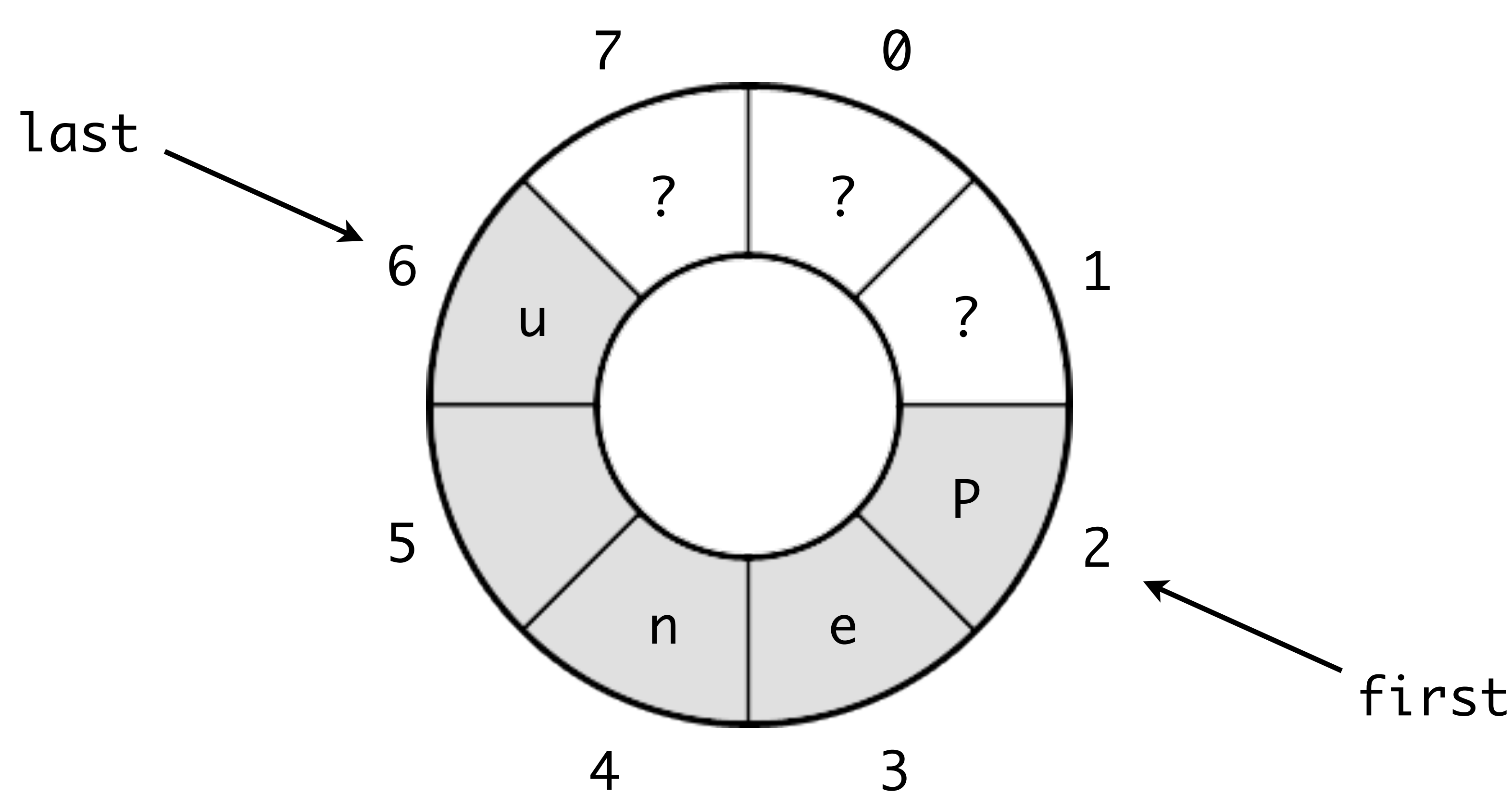
Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- essentially, we can think of the array as being a continuous loop:



Queue: Array Version

Like before, our implementation will use a partially filled array

- unlike before, though, we need access to BOTH ends of the array

One solution is to treat the array as "circular"...

- we'll be calculating the next index in a couple different places.

So, we'll use a helper function (a private method) to make life easier:

```
// returns the next index after @i in the array
size_type next_index(size_type i) const {
    return (i + 1) % CAPACITY;
}
```

Queue: Array Version

Starting the header file:

```
// queue.h header file
// specification documentation
#pragma once
#include <cstdlib>

namespace CS262 {
    template <typename Item>
    class queue { };
}

#include "queue.cpp"
```

Queue: Array Version

Starting the implementation file:

```
// queue.cpp implementation file
// INVARIANT (coming soon)
#include <cassert>
using namespace std;

namespace CS262 {
    // member implementations
}
```

Queue: Array Version

Type definitions and member constants:

```
template <typename Item>
class queue {
    public:
        // typedefs and constants

    private:
        Item data[CAPACITY];
        size_type first; // index of front item
        size_type last;  // index of rear item
        size_type used;  // number of items in queue
};
```

Queue: Array Version

Document invariant in implementation file:

```
// queue.cpp implementation file
// This file is included in the header file and not
//   compiled separately.
// INVARIANT for queue<Item> class:
// 1. The number of items in the queue is stored in the
//    member variable used
// 2. For a non-empty queue, the items are stored in a
//    circular array beginning at data[front] and
//    continuing through data[last].
// 3. For an empty queue, last is some valid index, and
//    first is always equal to next_index(last).
```

Queue: Array Version

The constructor (inline implementation):

```
// creates an empty queue
queue() : first(0), last(CAPACITY - 1), used(0) { }
```

Queue: Array Version

The size method (inline implementation):

```
// returns the total number of items in the queue
size_type size() const { return used; }
```

Queue: Array Version

The empty method (inline implementation):

```
// returns true if the queue is empty, false otherwise
bool empty() const { return (used == 0); }
```

Queue: Array Version

The push function prototype:

```
// adds @entry to the rear of the queue
void push(const Item& entry);
```

How will this function be implemented?

```
template <typename Item>
void queue<Item>::push(const Item& entry) {
    assert(size() < CAPACITY);
    last = next_index(last);
    data[last] = entry;
    used++;
}
```

Queue: Array Version

The pop function prototype:

```
// removes the first item in the queue
void pop();
```

How will this function be implemented?

```
template <typename Item>
void queue<Item>::pop(const Item& entry) {
    assert(!empty());
    first = next_index(first);
    used--;
}
```

Queue: Array Version

The front function prototype:

```
// returns the front item in the queue
Item front() const;
```

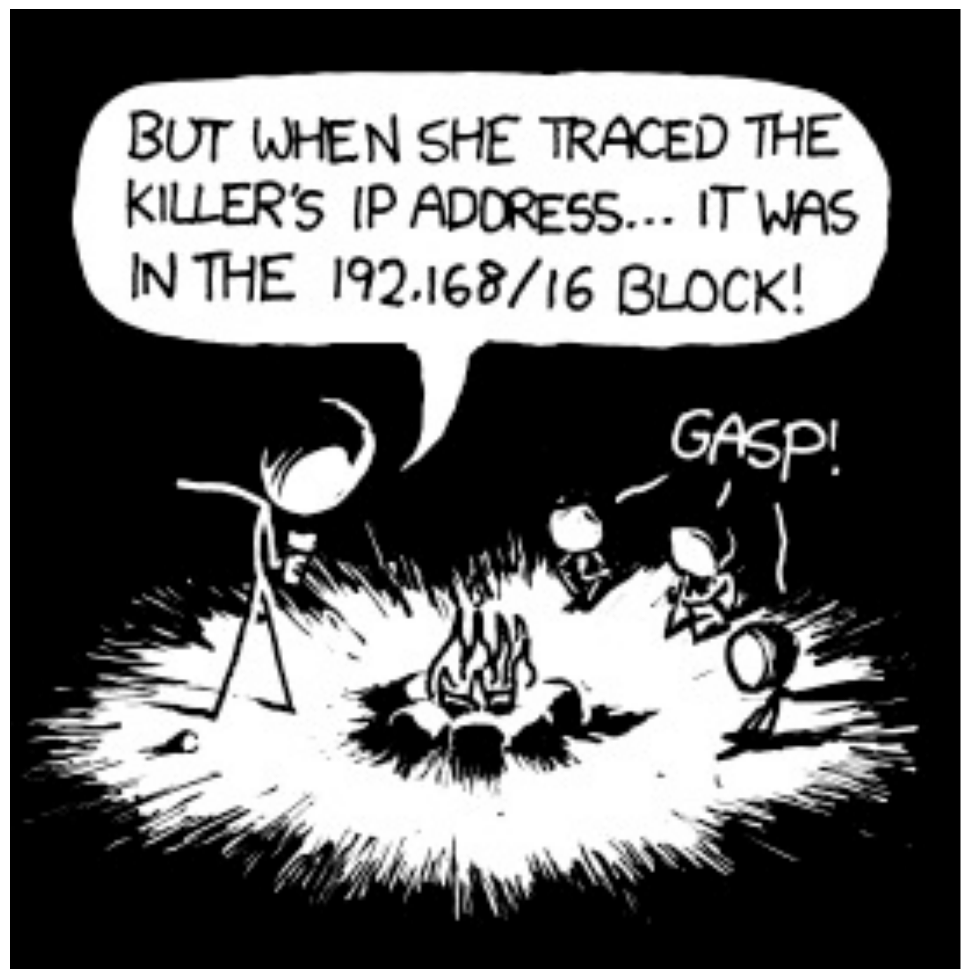
How will this function be implemented?

```
template <typename Item>
Item queue<Item>::front() const {
    assert(!empty());
    return data[first];
}
```

Queue: Array Version

The array version doesn't use dynamic memory...

- this means that the automatic versions of the copy constructor, assignment operator, and destructor are just fine! Hooray!



Queue Implementation: Linked-List Version

Specification

Queue: Linked-List Version

Template parameters, typedefs, and member constants:

```
// template parameter
template <typename Item>

// Item is the type of items in the queue, and must be
// a built-in type, or a class that provides:
//  - instantiation via a default constructor
//  - instantiation via a copy constructor
//  - assignment operator   (x = y)
```

Queue: Linked-List Version

Template parameters, typedefs, and member constants:

```
// alias for the template parameter
typedef Item value_type;
```

Queue: Linked-List Version

Template parameters, typedefs, and member constants:

```
// data type of variables that track the queue's size
typedef std::size_t size_type;
```

Queue: Linked-List Version

Constructors:

```
// creates an empty queue
queue();

// postcondition:
//   the queue has been initialized as an empty queue
```

Queue: Linked-List Version

Modification member functions:

```
// adds @entry to the rear of the queue
void push(const Item& entry);

// postcondition:
//   A new copy of @entry has been added to the rear of
//   the queue
```

Queue: Linked-List Version

Modification member functions:

```
// removes the first item in the queue
void pop();

// precondition:
//   size() > 0
// postcondition:
//   The top (first) item in the queue has been removed.
```

Queue: Linked-List Version

Constant member functions:

```
// returns the front item in the queue
Item front() const;

// precondition:
//   size() > 0
// postcondition:
//   The return value is the first (top) item of the
//   queue, but the queue is unchanged. This differs
//   slightly from the STL queue, where the front
//   function returns by reference.
```

Queue: Linked-List Version

Constant member functions:

```
// returns the total number of items in the queue
size_type size() const;

// postcondition:
//   The return value is the total number of items in
//   the queue
```

Queue: Linked-List Version

Constant member functions:

```
// returns true if the queue is empty, false otherwise
bool empty() const;

// postcondition:
//   The return value is true if the queue is empty, and
//   false otherwise
```

Queue: Linked-List Version

Value semantics:

```
// queue<Item> objects may be:
//   assigned using operator =
//   copied via the copy constructor
```

Queue: Linked-List Version

Dynamic memory usage:

```
// If there is insufficient dynamic memory, then the
// following functions throw bad_alloc:
//    copy constructor
//    push
//    operator =
```

Queue Implementation: Linked-List Version

Implementation

Queue: Linked-List Version

Starting the header file:

```
// queue.h header file
// specification documentation
#pragma once
#include <cstdlib>
#include "Node.h"

namespace CS262 {
    template <typename Item>
    class queue { };
}
#include "queue.cpp"
```

Queue: Linked-List Version

Starting the implementation file:

```
// queue.cpp implementation file
// INVARIANT (coming soon)
#include <cassert>
using namespace std;

namespace CS262 {
    // member implementations
}
```

Queue: Linked-List Version

Type definitions and member constants:

```
template <typename Item>
class queue {
    public:
        // typedefs and constants

    private:
        Node<Item>* front_ptr;
        Node<Item>* last_ptr;
        size_type used;
};
```

Queue: Linked-List Version

Document invariant in implementation file:

```
// queue.cpp implementation file
// This file is included in the header file and not
//   compiled separately.
// INVARIANT for queue<Item> class:
// 1. The number of items in the queue is stored in the
//    member variable used
// 2. The items in the queue are stored in a linked
//    list, with the front of the queue stored at the
//    head node and the rear of the queue stored at the
//    final node.
```

Queue: Linked-List Version

Document invariant in implementation file:

```
// queue.cpp implementation file
// This file is included in the header file and not
//   compiled separately.
// INVARIANT for queue<Item> class (continued):
// 3. The member variable front_ptr is the head pointer
//    of the linked list of items. For a non-empty
//    queue, the member variable rear_ptr is the tail
//    pointer of the linked list; for an empty list, we
//    don't care what's stored in rear_ptr.
```

Queue: Linked-List Version

The constructor (inline implementation):

```
// creates an empty queue
queue() : front_ptr(NULL), used(0) { }
```

Queue: Linked-List Version

The size method (inline implementation):

```
// returns the total number of items in the queue
size_type size() const { return used; }
```

Queue: Linked-List Version

The empty method (inline implementation):

```
// returns true if the queue is empty, false otherwise
bool empty() const { return (used == 0); }
```

Queue: Linked-List Version

The push function prototype:

```
// adds @entry to the rear of the queue
void push(const Item& entry);
```

How will this function be implemented?

- we have to handle two cases...
- the first is when the queue is initially empty
- and the second is when it is not empty

Queue: Linked-List Version

A push function implementation:

```
template <typename Item>
void queue<Item>::push(const Item& entry) {
    Node<Item>* node = new Node<Item>(entry);

    if (empty()) {
        front_ptr = rear_ptr = node;
    } else {
        rear_ptr = rear_ptr->next = node;
    }
}
```

Queue: Linked-List Version

The pop function prototype:

```
// removes the first item in the queue

void pop();
```

How will this function be implemented?

- we need to check the precondition,
- remove and delete the first node,
- and finally decrement used...

Queue: Linked-List Version

A pop function implementation:

```
template <typename Item>
void queue<Item>::pop(const Item& entry) {
    assert(!empty());

    Node<Item>* remove_ptr = front_ptr;
    front_ptr = front_ptr->next;
    delete remove_ptr;

    used--;
}
```

Queue: Linked-List Version

The front function prototype:

```
// returns the front item in the queue
Item front() const;
```

How will this function be implemented?

```
template <typename Item>
Item queue<Item>::front() const {
    assert(!empty());
    return front_ptr->data;
}
```

Stack: Array Version

The linked-list version <u>does</u> use dynamic memory

- we need to implement our own copy constructor, assignment operator, and destructor
- this should be completely within your abilities by now...

COMIC!