Queues

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, April 11, 2008

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview
Stacks & Queues

Major Topics

- ✓ Stacks
- Queues

Review
Stacks [1/2]

A Stack is:

- A kind of container.
- A Last-In-First-Out (LIFO) structure.
- A restricted version of a Sequence.

Conceptually, a Stack carries out the idea of **top-down design**.

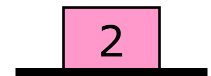
Three primary operations:

- **push**
- **pop**
- **getTop**

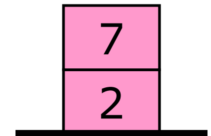
A Stack can be implemented simply as a wrapper around some existing Sequence type.

1. Start:
   Empty Stack.

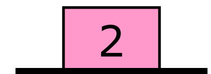
2. Push 2.



3. Push 7.

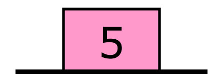


4. Pop.



5. Pop.
   Empty again.

6. Push 5.



Review
Stacks [2/2]

The STL has a Stack: **`std::stack`**, in **`<stack>`**.

- STL documentation does not call **`std::stack`** a "container", but rather a "container adapter".
- This is because **`std::stack`** is explicitly a **wrapper** around some other container.

You get to pick what that container is.

- You say "**`std::stack<T,`** *container***`<T> >`**".
  - "**`T`**" is the value type.
  - "*container*" can be **`std::vector`**, **`std::deque`**, or **`std::list`**.
  - "*container***`<T>`**" can be **any** standard-conforming container with member functions **`back`**, **`push_back`**, **`pop_back`**, **`empty`**, **`size`**, along with comparison operators (**`==`**, **`<`**, etc.).
- *container* defaults to **`std::deque`**.
  - You can say just "**`std::stack<T>`**" to get "**`std::stack<T, std::deque<T> >`**".

Queues
Overview

Our next ADT is Queue.

- Say “Q”.
- A Queue is …
  - … very similar to a Stack in **definition**,
  - … somewhat different from a Stack in **implementation**, and
  - … very different from a Stack in **application**.

Topics

- What a Queue is.
- The ADT Queue interface.
- Implementing a Queue.
- Queues in the C++ STL.
- Applications of Queues.

Queues
What a Queue Is — Idea

A **Queue** is a kind of container.

- As usual, it stores a number of values, all of the same type.

A *Queue* is a First-In-First-Out (FIFO) structure.

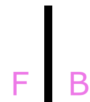
- What we do with a Queue:
  - **Enqueue**: add a new value at the *back*.
    - Say “N Q”.
  - **Dequeue**: Remove a value at the *front*.
    - Say “D Q”.
- The first item added is the first removed.
  - Think of people standing in line. (This is also a good way to remember which end is “front” and which is “back”.)
- Some people use other words for “enqueue” & “dequeue”.
  - “Push” and “pop”, for example.

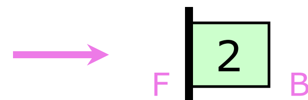
So a Queue is another restricted version of a Sequence.

- We can only insert at one end and remove at the other.
- We (usually) cannot iterate through the contents.

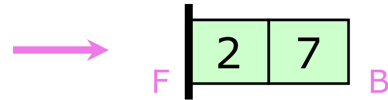
Queues
What a Queue Is — Illustration

1. Start:
   an empty Queue.

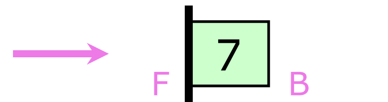




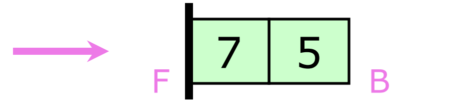
2. Enqueue 2.



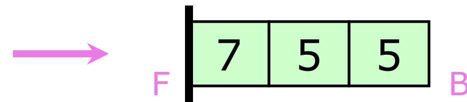
3. Enqueue 7.



4. Dequeue.

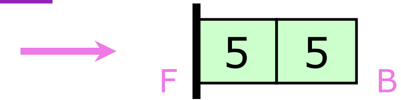


5. Enqueue 5.

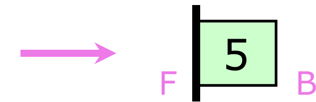


6. Enqueue 5.

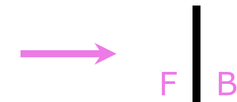
7. Dequeue.



8. Dequeue.

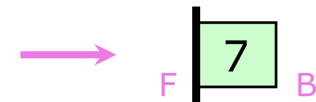


9. Dequeue.



   Queue is empty again.

10. Enqueue 7.



11. Etc. …

Compare this with Stack!

Queues
What a Queue Is — Waiting

Conceptually, a Queue carries out the idea of **waiting in line**.

- Items that need to be processed are enqueued.
- When we are able to process an item, we dequeue it and process it.
- As long as the processor keeps going, no item languishes forever. They are all processed eventually.

In practice, nearly every use of a Queue has this idea behind it.

Queues
Interface

As with Stacks, there is essentially only one good interface to a Queue:

- Data
  - A sequence of data items.
- Operations
  - **getFront**. Look at front item.
  - **enqueue**. Add an item to the back.
  - **dequeue**. Remove front item.
  - The text adds:
    - **dequeue(queueItemType & queueFront)**. Returns old front item in reference param.
  - To avoid errors we need information about empty state (or size):
    - **isEmpty**. Returns true if queue is empty.
  - Then, of course, we need bookkeeping:
    - **create**.
    - **destroy**.
    - Again, I will add the usual **copy** operations.

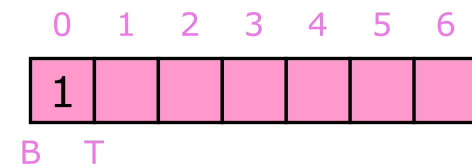
Queues
Implementation — Warning

Despite their similarities, Queues are trickier to implement than Stacks.

- One reason is that the data in a Queue "crawls" around.

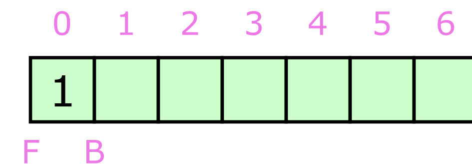
Consider a Stack implemented using a **fixed-size** array with top & bottom markers.

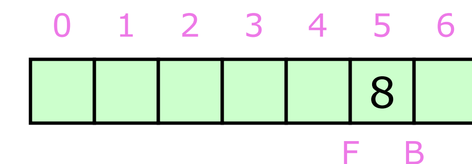
- Begin with a single item on the Stack: a "1" in array element 0.
- Do **`push(8)`** five times, and **`pop()`** five times.
- Result: Exactly the Stack we started with.



Consider a Queue implemented using a **fixed-size** array with front & back markers.

- Begin with a single item in the Queue: a "1" in array element 0.
- Now do **`enqueue(8)`** five times and **`dequeue()`** five times.
- Result: The single data item in the Queue is an **`8`** in array item 5.
- If the size of the array is as pictured, then two more **`enqueue`** operations will result in the data "crawling" off the end of the array.





Queues
Implementation — Sequence Wrapper

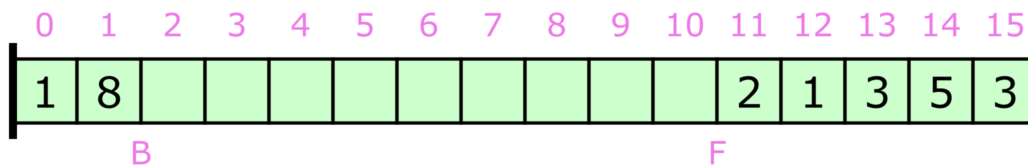
As with a Stack, a Queue is often implemented as a wrapper around a Sequence type.

- We would need to use a Sequence type that has fast insertion at one end and fast removal at the other end.
  - NOT a (smart) array.
  - *Maybe* a Singly Linked List …
    - With the right interface. We would need to keep an iterator to the last element. We can then insert at the end and remove at the beginning. Since we never do remove-at-end, we can always update the iterator when it changes.
  - A Doubly Linked List works.
  - Something like **`std::deque`** works.
- As with a Stack, it is likely that the Queue operations are essentially already implemented.
  - We typically only need to write a bunch of one-line functions.

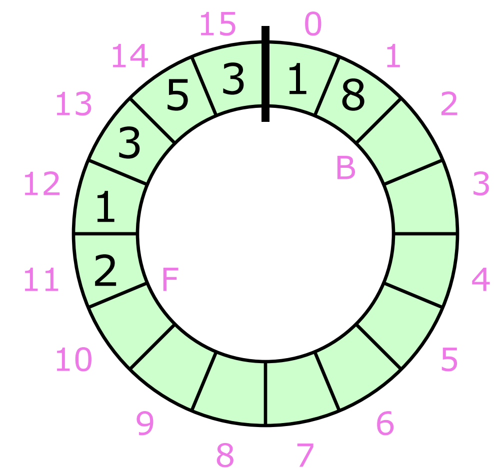
Queues
Implementation — Circular Buffer [1/3]

Another way to implement a Queue is to use a **circular buffer**.

- In practice, a circular buffer is just an ordinary Sequence. However, we think of the ends as being joined.



Physical Structure



Logical Structure

Queues
Implementation — Circular Buffer [2/3]

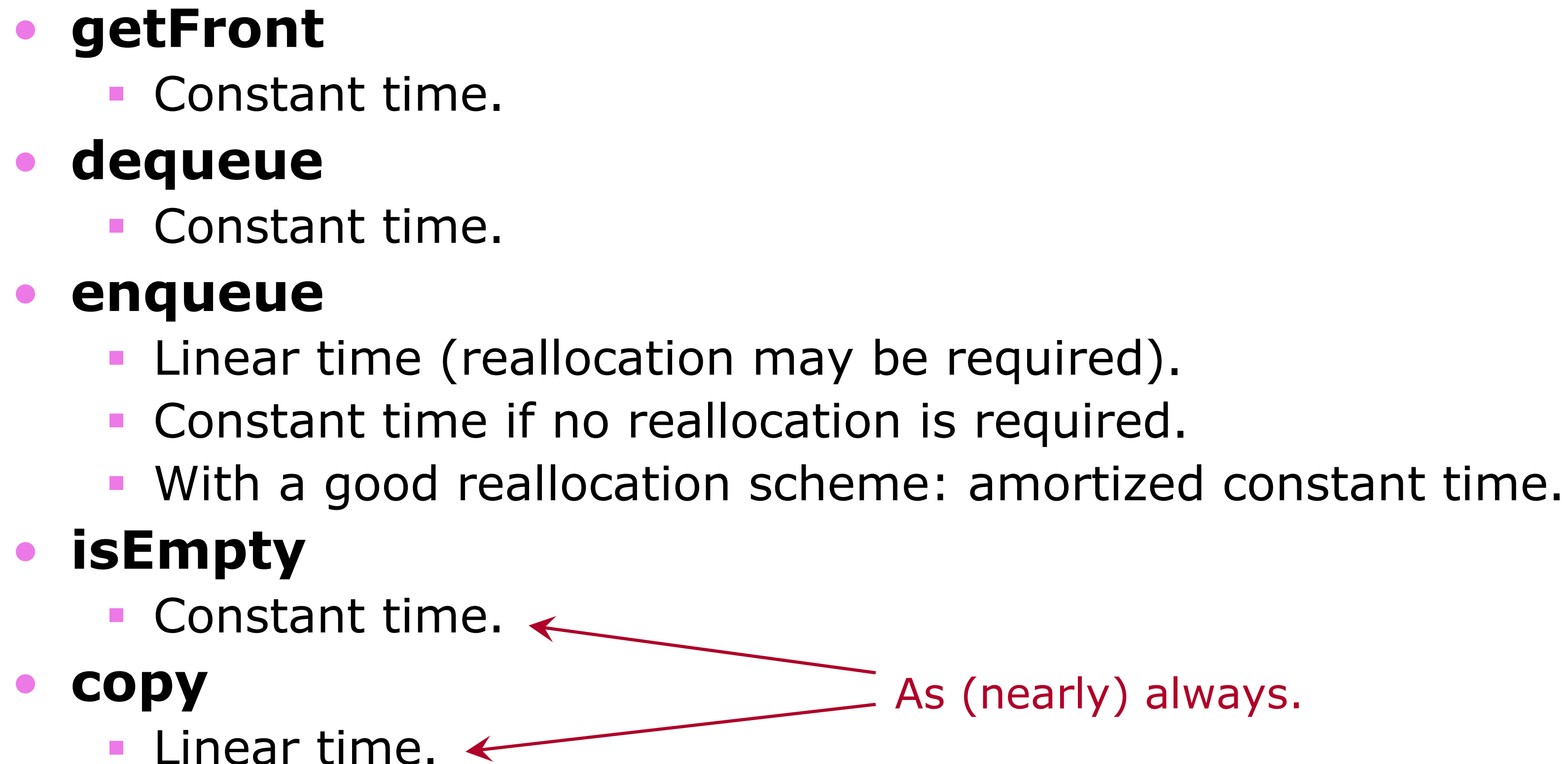
A circular buffer can be simply an array. We need to know:

- The number of elements in the array.
- The subscript of the front item.
  - When dequeuing, we do
    `frontsubs = (frontsubs + 1) % array_size`.
- The size of the Queue (that is, the number of items in it).
  - The subscript of the back item is
    `(frontsubs + size - 1) % array_size`, if **`queue_size != 0`**.

This is a good way of implementing a Queue that will never exceed some smallish size. For a Queue that can get large:

- We may want to add automatic reallocation.
- This works in much the same way it does for smart arrays.
- When reallocating, be careful to copy items to the right places.

Queues
Implementation — Circular Buffer [3/3]



What is the order of each of the following operations for a Queue implemented using an array-based circular buffer?

- **getFront**
  - Constant time.
- **dequeue**
  - Constant time.
- **enqueue**
  - Linear time (reallocation may be required).
  - Constant time if no reallocation is required.
  - With a good reallocation scheme: amortized constant time.
- **isEmpty**
  - Constant time.
- **copy**
  - Linear time.

Queues
`std::queue` — Introduction

The STL has a Queue: **`std::queue`**, in **`<queue>`**.

- Again, STL documentation calls **`std::queue`** a "container adapter", not a "container".

As with **`std::stack`**, **`std::queue`** is a wrapper around a container that you choose.

- You say "**`std::queue<T,`** *container***`<T> >`**".
  - "**`T`**" is the value type.
  - "*container***`<T>`**" can be **any** standard-conforming container with value type **`T`** and the required member functions (including **`pop_front`**).
  - In particular "*container*" can be **`std::deque`**, or **`std::list`**.
    - But not **`std::vector`**! Why not? It has no **`pop_front`**.
- *container* defaults to **`std::deque`**.
  - You can say just "**`std::queue<T>`**" to get "**`std::queue<T, std::deque<T> >`**".

Queues
`std::queue` — Notes

Efficiency issues for **`std::queue`** are just like **`std::stack`**.

- Good overall performance is gotten from **`std::deque`**.
- Good worst-case performance is gotten from **`std::list`**, at the expense of memory management overhead.

Functions in **`std::queue`**.

- Enqueue is "**`push`**".
- Dequeue is "**`pop`**".
- GetFront is "**`front`**".
- And comparison operators are defined, etc.

About **`std::deque`**.

- It seems that **`std::deque`** exists primarily to serve as a basis for **`std::queue`** (and **`std::stack`**).
- I have never had occasion to use **`std::deque`** by itself.
  - But maybe you will …

Queues
Applications

Queues are used to mediate **asynchronous communications**.

- *Synchronous* = coordinated in time.
  - Example: I call you on the phone (we both stop everything and deal with the call).
- *Asynchronous* = not coordinated in time.
  - Example: I send you an e-mail (and you read and answer it when you can).
  - More relevant example: Computer sends document to printer. Printer prints it when it can.

The “waiting in line” behavior of Queues makes asynchronous communication work.

- Sender enqueues a message whenever it has one to send.
- Receiver dequeues a message whenever processing capability is available.
- All messages eventually get processed.

Applications of Queues often involve requests to use some limited resource (an I/O channel, a device, etc.), with requests waiting in line.

- In a print Queue, print jobs wait to be printed.
- In a program with a graphical user interface, user input is often processed in the form of “events”. An event might be a mouse click or a keypress. Events will be stored in an event Queue.