Notes on Assignment 6
Queues

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, November 9, 2009

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

Notes on Assignment 6
Suggestions

In Assignment 6 you will be writing a Linked List and basing a Stack on it.

Suggestions (none of these are required)

- Start from the code we have already written.

See **`linked_list.cpp`**, *on the web page.*

- Where you have a choice about putting functionality in the Linked List class or the Stack class, then put it in the Linked List class.
  - In my experience, this makes life easier.
- Do not use "friends" in this assignment.
  - Even now, over a decade after the ANSI C++ standard was published, some compilers do not handle friends that are templates correctly.
  - If you find yourself in a situation where a global function or other class needs to access an object's private data, then you may make that data public, or else add public accessor functions. Restrictions on adding extra public members to a class are lifted, for this assignment.

Notes on Assignment 6
Implementing Operations

Linked List: Copy Constructor

- Warning: The "obvious" copying method (add new nodes at the beginning of the copied list) gets you a backwards list.
- Method #1: Maintain a node pointer that proceeds through the new list. Add new nodes at the end.
- Method #2: Copy backwards, then call **`reverse`** (see below).

Linked List: Copy Assignment

- Use the swap trick, as in Assignment 5.

Linked List: Destructor

- Everything destroys what it owns.
- Every destructor is one line long.

Stack Big: Three

- If you have done the above right, then you can just use the silently written versions here.

Linked List: Reverse

- Only change pointers. Set each pointer to point to the previous node.
- *See the next slide …*

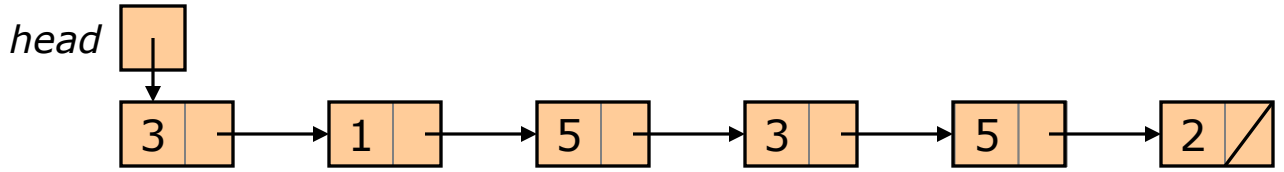
Notes on Assignment 6
Reversing a Linked List [1/3]

Your Linked List class must have a member function that reverses the order of the data.

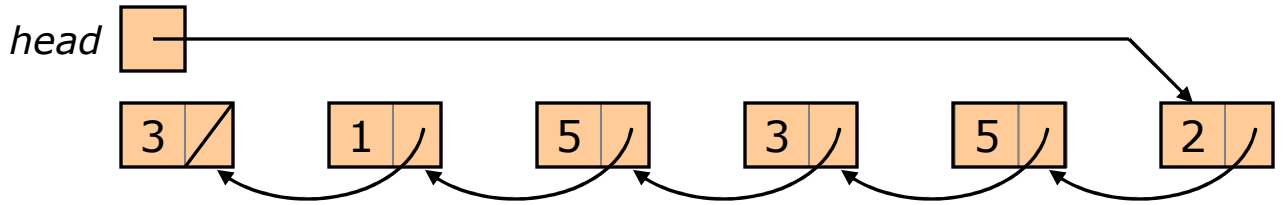
A Singly Linked List can be reversed:

- In place.
- Using no value-type operations.

Consider the following Linked List.

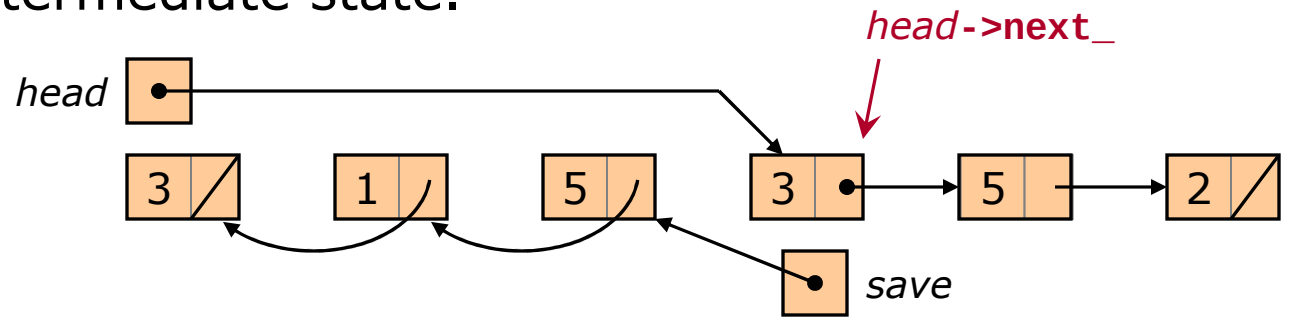


We want to reverse the pointers, leaving data items alone, resulting in the following Linked List, which has the same nodes.



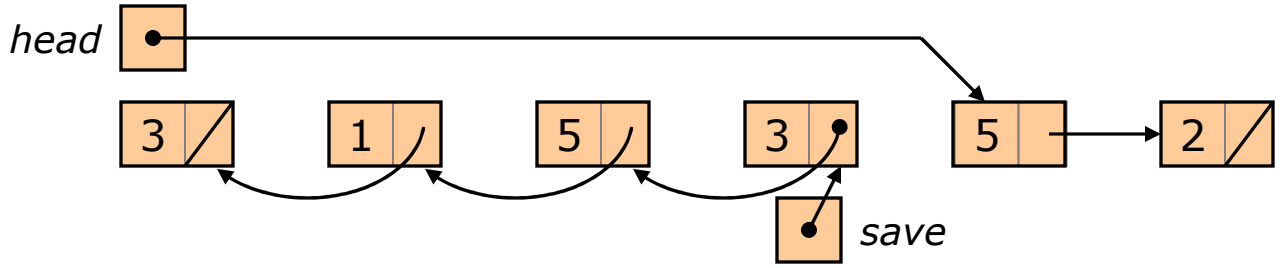
Notes on Assignment 6
Reversing a Linked List [2/3]

We need a loop. What does one iteration of this loop do? Consider an intermediate state.



- We need to save the start of the new reversed list. That is what the variable "*save*" is for.

Now consider the data one iteration later.

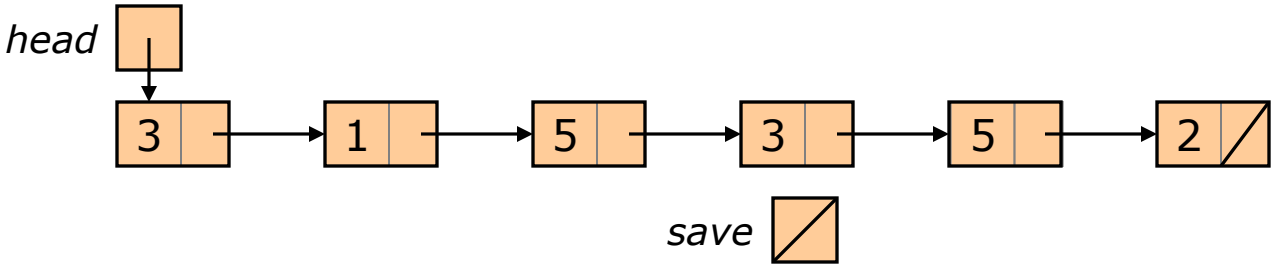


- So, what does one iteration do? Answer: A 3-pointer rotate operation, with *head*, *head*->**next**_, and *save* (marked with dots).

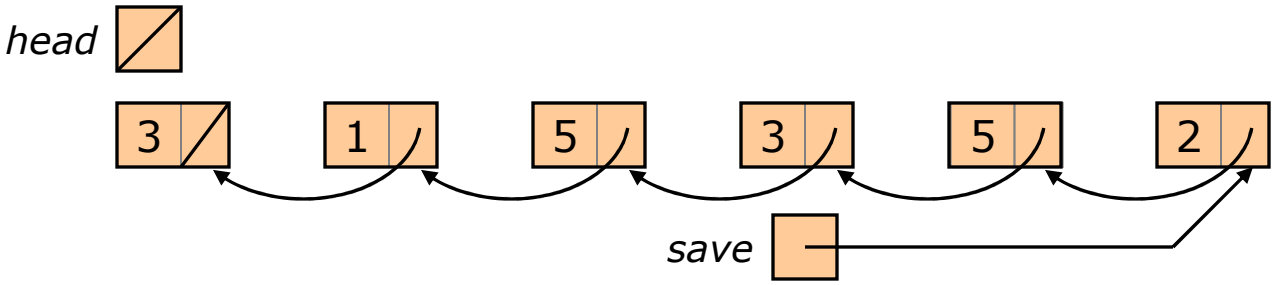
Notes on Assignment 6
Reversing a Linked List [3/3]

Other things to do:

- Start, before the loop, by setting *save* `= NULL`.
  - Here is the situation as the loop begins.



- Keep iterating as long as *head* `!= NULL`.
  - Here is what the situation should be when the loop terminates.



- Finish, after the loop, by setting *head* `=` *save*.

Unit Overview
Handling Data & Sequences

Major Topics

- ✓ Data abstraction
- ✓ Introduction to Sequences
- ✓ Smart arrays
  - ✓ Array interface
  - ✓ Basic array implementation
  - ✓ Exception safety
  - ✓ Allocation & efficiency
  - ✓ Generic containers
- ✓ Linked Lists
  - ✓ Node-based structures
  - ✓ More on Linked Lists
- ✓ Sequences in the C++ STL
- ✓ Stacks
- Queues

Review
Stacks — What a Stack Is, Implementation

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**.

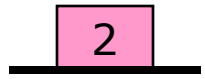
Three primary operations:

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

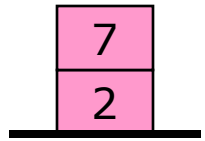
A Stack can be implemented simply as a wrapper around some existing Sequence type.

1. Start:
   Empty Stack.

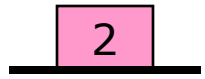
2. Push 2. →



3. Push 7. →



4. Pop. →



5. Pop.
   Empty again. →

6. Push 5. →

Review
Stacks — In the C++ STL, Applications

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

- This is a "container adapter". The data is stored in some other container.

`vector`, **`deque`**, or **`list`**

```
std::stack<T, container<T> >
std::stack<T>  // = std::stack<T, std::deque<T> >
```

We looked at two applications of Stacks.

- Expression evaluation.
  - A Stack can be used to do a very simple evaluator for **Reverse Polish Notation**.
    - Normal (infix): (2 + 3) * (7 – 5). RPN (postfix): 2 3 + 7 5 – *.

See **`rpn.cpp`**, *on the web page.*

- Eliminating recursion.
  - We used the "brute force method" to eliminate the recursion in **`fibo1.cpp`**. Result: long, slow, but correct.

See **`fibo6.cpp`**, *on the web page.*

Queues
What a Queue Is — Idea [1/2]

Our fourth ADT is **Queue**. This is yet another container ADT; that is, it holds a number of values, all the same type.

- 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**.

What a Queue Is — Idea [2/2]

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.

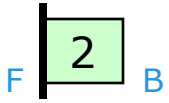
Thus, a Queue is another restricted version of a Sequence.

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

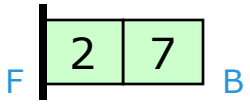
Queues
What a Queue Is — Illustration

1. Start:
   an empty Queue.

2. Enqueue 2. →

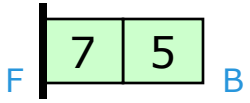


3. Enqueue 7. →

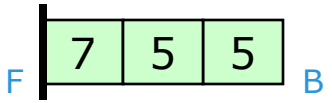


4. Dequeue. →

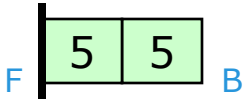
5. Enqueue 5. →



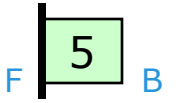
6. Enqueue 5. →



7. Dequeue. →



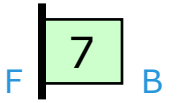
8. Dequeue. →



9. Dequeue. →

Queue is empty again.

10. Enqueue 7. →



11. Etc. …

Compare this with Stack!

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
What a Queue Is — ADT

As with a Stack, 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.

Three primary operations.

  - 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 — #1: Sequence Wrapper

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

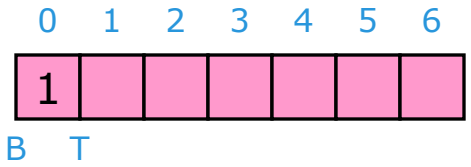
- We would need to use a Sequence type that has fast insertion at one end and fast removal at the other end.
  - NOT a (smart) array.
  - *Maybe* a Singly Linked List …
    - With the right interface. We would need to maintain an iterator to the last element. We can then insert at the end and remove 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 — #2??: Array + Markers

Suppose we try something simpler: put our data in an array with markers indicating the ends.

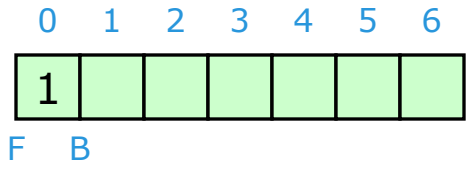
Consider a Stack based on an array with top & bottom markers.

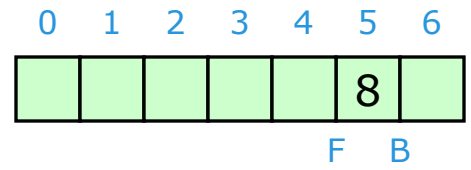
- Begin with a single item on the Stack: a "1" in array element 0.
- Do **push(8)** five times, and **pop()** five times.
- Result: Exactly the Stack we started with.



Now consider a Queue based on an 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.



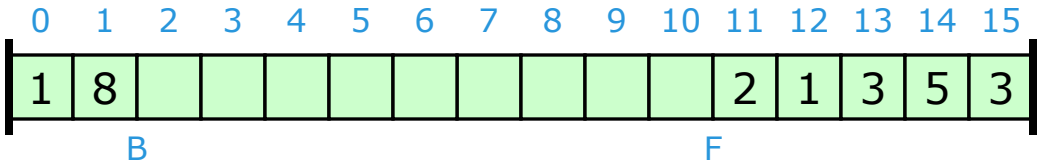


This "crawling data" can make Queues trickier to implement than Stacks.

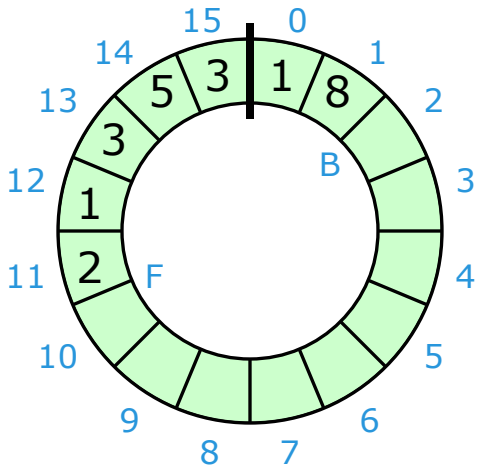
Queues
Implementation — #2: Circular Buffer [1/3]

When we store a Queue in an array with markers, we can deal with "crawling data" using a **circular buffer**.

- A circular buffer is just an ordinary Sequence. However, we think of the ends as being joined.
- We also have markers indicating the front and back of the Queue.
- We generally do not expand or contract the Sequence itself when the Queue expands or contracts; we just move the markers.
- Note that we might still need to expand the Sequence if it fills up.



Physical Structure



Logical Structure

Implementation — #2: 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.

Implementation — #2: 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
In the C++ STL — 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.

`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 **`push_back`**, **`pop_front`**, and **`front`**).
- In particular *container* can be **`std::deque`** or **`std::list`**.
  - But not **`std::vector`** or **`std::basic_string`**; these have no **`pop_front`**.

container defaults to **`std::deque`**.

```
std::queue<T>  // = std::queue<T, std::deque<T> >
```

Queues
In the C++ STL — 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'll call you on the phone at 3 p.m. (we both stop everything at the agreed time 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.

Unit Overview
The Basics of Trees

We now begin a unit covering a very different basis for ADTs & data structures: **trees**.

Major Topics

- Introduction to Trees
- Binary Trees
- Binary Search Trees
- Treesort

Material is in Chapter 10 in the text.

After this, we look at Tables & Priority Queues.

- Some of the more interesting kinds of trees (Binary Heaps, 2-3 Trees, 2-3-4 Trees, Red-Black Trees, AVL Trees) will be covered in this next unit.