Notes on Assignment 6
Applications of Stack (cont.)
Queues

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, April 5, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

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)

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

- Start from the code we have already written.
- 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" until you have your code working.
  - Until a couple of years ago, over a decade after the ANSI C++ standard was published, some compilers did not handle friends that are templates correctly. Additionally, they are syntactically difficult to handle.
  - If you find yourself in a situation where a global function or other class needs to access an object's private data, then you should temporarily make that data public, or else add public accessor functions.

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 …*

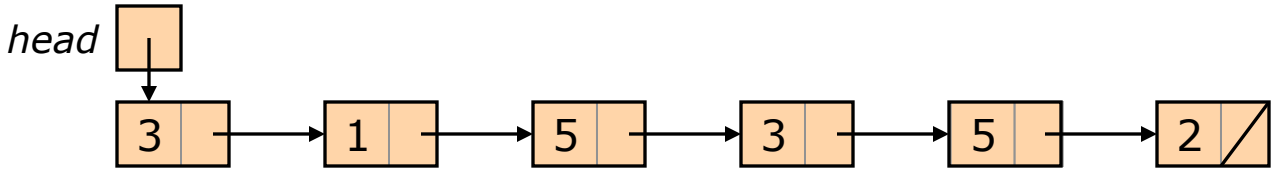
Notes on Assignment 6
Reversing a Linked List [1/3]

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

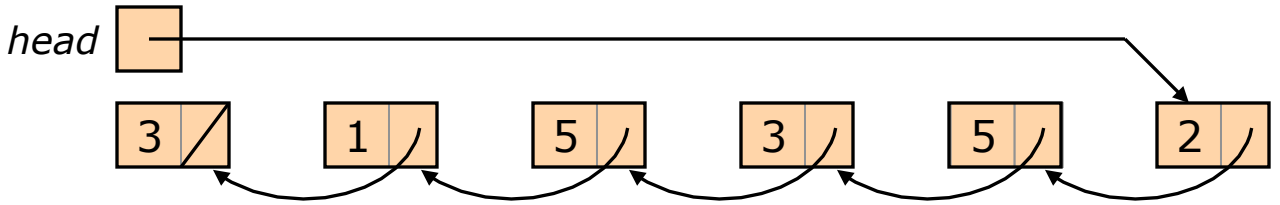
A Singly Linked List can be reversed:

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

Consider the following Linked List.

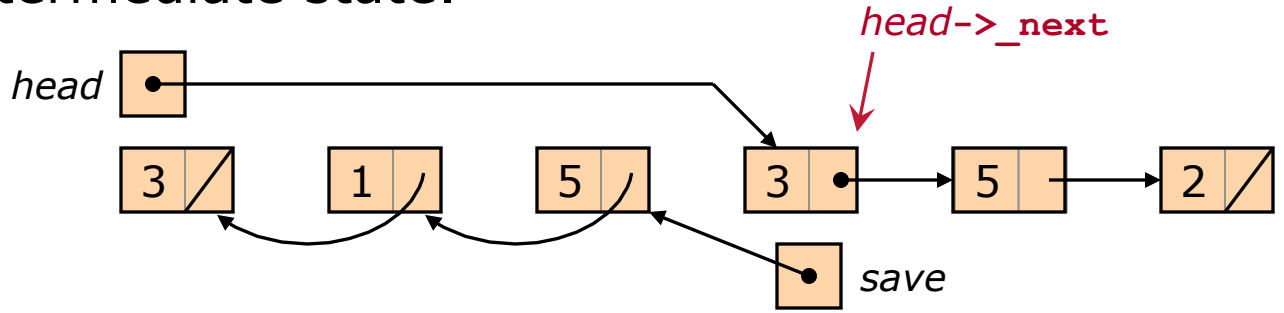


We want to reverse the pointers, leaving data items alone, resulting in the following Linked List, which has the same nodes.



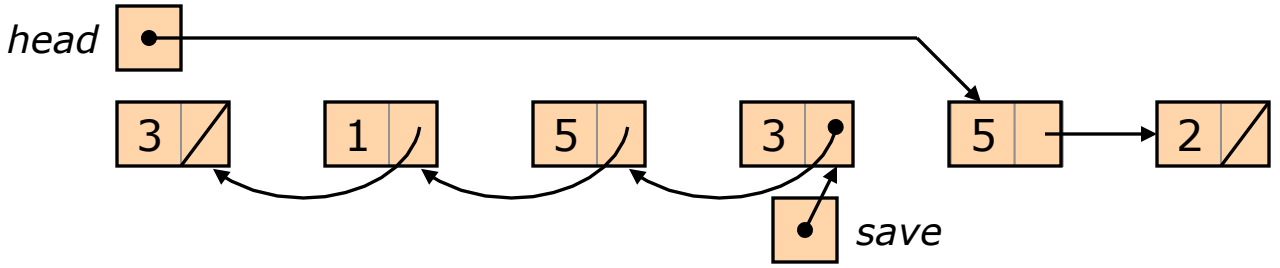
Notes on Assignment 6
Reversing a Linked List [2/3]

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.

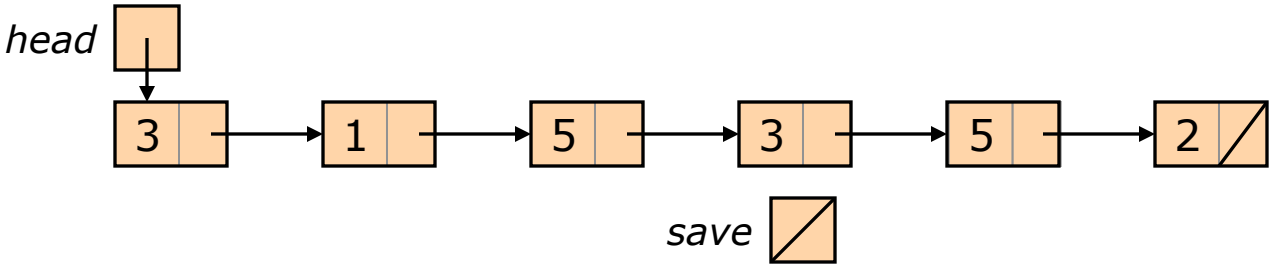


- So, what does one iteration do? Answer: A 3-pointer rotate operation, with *head*, *head*->**next_**, and *save* (marked with dots).

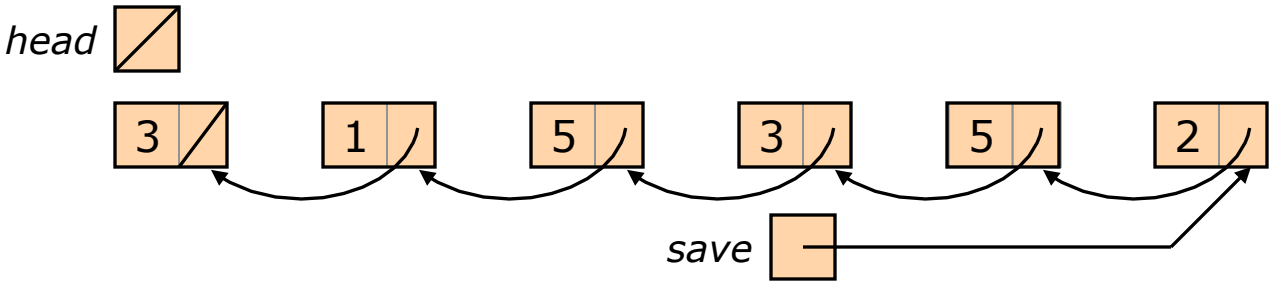
Notes on Assignment 6
Reversing a Linked List [3/3]

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*.

Applications — Eliminating Recursion: Refresher [1/2]

From the "Eliminating Recursion" slides:

Fact. Every recursive function can be rewritten as an iterative function that uses essentially the same algorithm.

- Think: How does the system help you do recursion?
  - It provides a **Stack**, used to hold return addresses for function calls, and values of automatic local variables.
- We can implement such a Stack ourselves. We need to be able to store:
  - Values of automatic local variables, including parameters.
  - The return value (if any).
  - Some indication of where we have been in the function.
- Thus, we can eliminate recursion by mimicking the system's method of handling recursive calls using Stack frames.

Stacks
Applications — Eliminating Recursion: Refresher [2/2]

To rewrite **any** recursive function in iterative form:

"Brute-force" method

- Declare an appropriate Stack.
  - A Stack item holds all automatic variables, an indication of what location to return to, and the return value (if any).
- Replace each automatic variable with its field in the top item of the Stack.
  - Set these up at the beginning of the function.
- Put a loop around the *rest* of the function body: "`while (true) { … }`".
- Replace each recursive call with:
  - Push an object with parameter values and current execution location on the Stack.
  - Restart the loop ("`continue`").
  - A label marking the current location.
  - Pop the stack, using the return value (if any) appropriately.
- Replace each "`return`" with:
  - If the "return address" is the outside world, really `return`.
  - Otherwise, set up the return value, and skip to the appropriate label ("`goto`"?).

This method is primarily of theoretical interest.

- *Thinking* about the problem often gives better solutions than this.
- We will look at this method further when we study **Stacks**. ← **NOW**

Applications — Eliminating Recursion: Example [1/6]

Here is function **fibo** from **fibo1.cpp**.

```
bignum fibo(int n)
{
    // BASE CASE
    if (n <= 1)
        return bignum(n);

    // RECURSIVE CASE
    return fibo(n-2) + fibo(n-1);
}
```

Let's use the "brute force" recursion elimination procedure to produce a non-recursive version.

- I have already written this. In the following slides we examine the code.

Stacks
Applications — Eliminating Recursion: Example [2/6]

When we eliminate recursion, all local values will be stored on our Stack. For convenience I rewrote function **`fibo`** so that computed values have names.

```
bignum fibo(int n)
{
    bignum v1, v2;

    // BASE CASE
    if (n <= 1)
        return bignum(n);

    // RECURSIVE CASE
    v1 = fibo(n-2);  // Recursive call #1
    v2 = fibo(n-1);  // Recursive call #2
    return v1 + v2;  // Return the result
}
```

Applications — Eliminating Recursion: Example [3/6]

We need a Stack. It holds:

- All variables and necessary temporary values.
  - These are **`n`**, **`v1`**, **`v2`**, and the return value.
- Some indication of where to return.
  - Three possibilities: the outside world, recursive call #1, recursive call #2.

We can use a **`struct`** for our Stack frame:

```
struct FiboStackFrame {
    int n;                  // Parameter
    bignum v1;              // Result of recursive call #1
    bignum v2;              // Result of recursive call #2
    bignum returnValue;     // Value to return
    int returnAddr;         // Return address:
                            //     0: outside world
                            //     1: recursive call #1
                            //     2: recursive call #2
};
```

Applications — Eliminating Recursion: Example [4/6]

We need to create our Stack when we enter function **`fibo`**.

```
std::stack<FiboStackFrame> s;
```

Then we can store our local variables there.

- So, for example, "**`n`**" becomes "**`s.top().n`**".

We need variables to hold values during Stack operations.

- Some will be **`int`**s and some will be **`bignum`**s.

```
int tmpi;
bignum tmpb;
```

After setting up the initial values, we enter a big **`while`** loop.

Stacks
Applications — Eliminating Recursion: Example [5/6]

To make a recursive call:

- We set up the Stack and restart the loop (**`continue`**).
- We must enable the function to return here. Use a **label**, and return to it with "**`goto`**".

For example, here is "**`v1 = fibo(n-2);`**":

```
    tmpi = s.top().n - 2;
    s.push(FiboStackFrame());  // Make new stack frame
    s.top().n = tmpi;          // Set parameter
    s.top().returnAddr = 1;    // Set return address
                               //  (recursive call #1)
    continue;                  // Do "recursive call"
label1:                        // Place to return to
    tmpb = s.top().returnValue;
    s.pop();
    s.top().v1 = tmpb;         // Put returned value in v1
```

Applications — Eliminating Recursion: Example [6/6]

To "return":

- If we were called by the outside world, then really **`return`**.
- Otherwise, set up the return value, and **`goto`** the appropriate location.
  - Note: As on the previous slide, I pop the Stack *after* returning.

For example, here is "**`return bignum(n);`**":

```
s.top().returnValue = bignum(s.top().n);
if (s.top().returnAddr == 1)        // Back to recursive call #1
    goto label1;
else if (s.top().returnAddr == 2)   // Back to recursive call #2
    goto label2;
else                                // Back to outside world
{
    tmpb = s.top().returnValue;
    s.pop();
    return tmpb;
}
```

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**.

Queues
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.

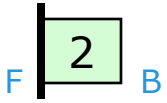
Thus, a Queue is another restricted version of a Sequence.

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

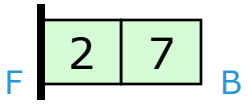
Queues
What a Queue Is — Illustration

1. Start:
   an empty Queue.

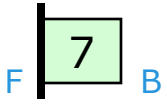
2. Enqueue 2. →



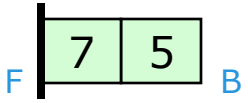
3. Enqueue 7. →



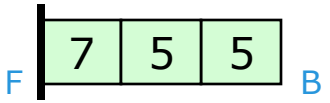
4. Dequeue. →



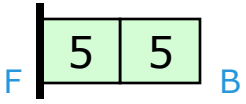
5. Enqueue 5. →



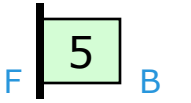
6. Enqueue 5. →



7. Dequeue. →



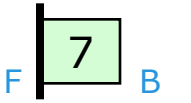
8. Dequeue. →



9. Dequeue. →

Queue is empty again.

10. Enqueue 7. →



11. Etc. …

Compare this with Stack!

Queues
What a Queue Is — Waiting

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

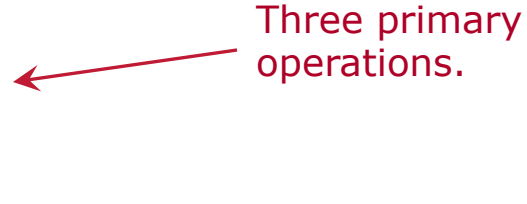
- Items that need to be processed are enqueued.
- When we are able to process an item, we dequeue it and process it.
- As long as the processor keeps going, no item languishes forever. They are all processed eventually.

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

Queues
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.
  - 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.

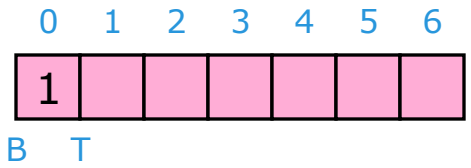
- We would need to use a Sequence type that has fast insertion at one end and fast removal at the other end.
  - NOT a (smart) array.
  - *Maybe* a Singly Linked List …
    - With the right interface. We would need to maintain an iterator to the last element. We can then insert at the end and remove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

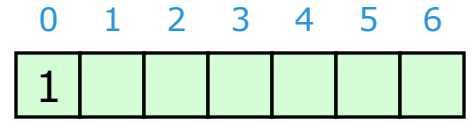
Consider a Stack based on an array with top & bottom markers.

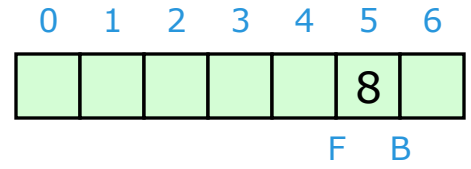
- Begin with a single item on the Stack: a "1" in array element 0.
- Do **`push(8)`** five times, and **`pop()`** five times.
- Result: Exactly the Stack we started with.



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.



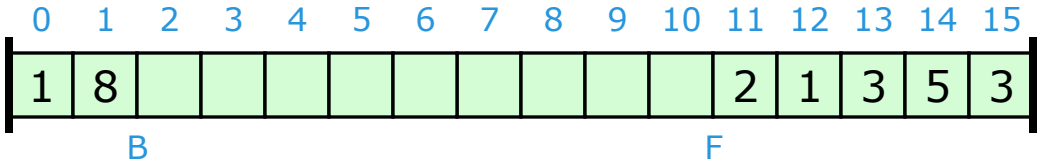


This "crawling data" can make Queues trickier to implement than Stacks.

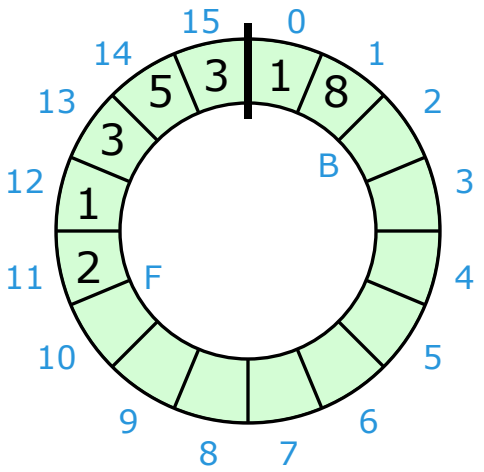
Queues
Implementation — #2: Circular Buffer [1/3]

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 + queue_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 — #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.