

Node-Based Structures, cont'd

Linked Lists

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, April 4, 2008

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview

Sequences and Their Implementations

Major Topics

- ✓ • Data Abstraction
- ✓ • Sequence Data
- ✓ • Interfaces to Data
- ✓ • Data Structure Implementation
- ✓ • Exception Safety
- ✓ • Allocation & Efficiency
- ✓ • Generic Containers
- 1/2 ✓ • Node-Based Structures
 - Linked Lists
 - Sequences in Practice

Review

Allocation & Efficiency

An operation is **amortized-constant-time** if k operations require $O(k)$ time.

- Thus, over many consecutive operations, the operation averages constant time.
- *Not* the same as constant-time average case.
- Quintessential amortized-constant-time operation: insert-at-end for a well written (smart) array.

Fixing Class **Sequence**

- Our original design did not allow for efficient insert-at-end.
 - Reallocate-and-copy would happen every time.
- The revised design had three data members: size, capacity, and the array pointer.
- Having a “capacity” member allows us to keep a record of how much memory is allocated. Then we can allocate extra so that we do not need to reallocate every time.
- Result: amortized-constant-time insert-at-end.

Review

Generic Containers — Exception Neutrality

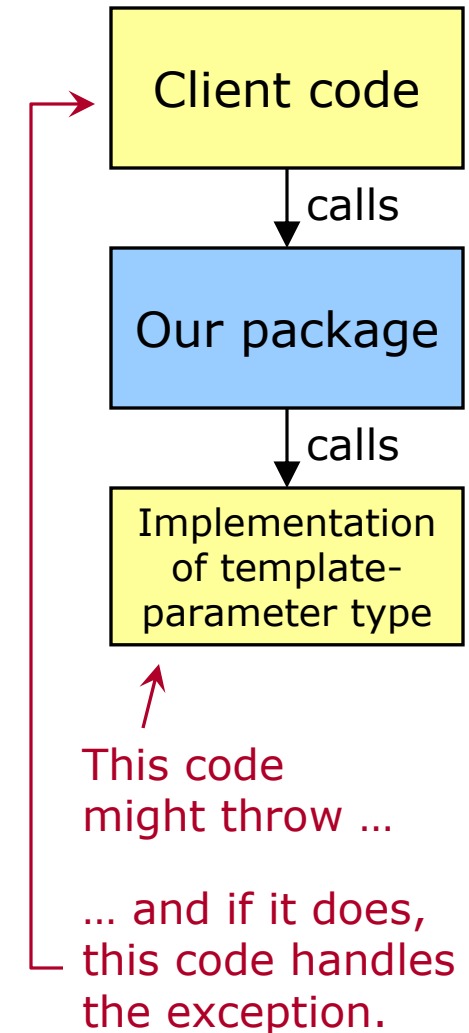
A **generic container** is a container that can hold a client-specified data type.

- In C++ we usually implement a generic container using a **class template**.

A function that allows exceptions thrown by a client's code to propagate unchanged, is said to be **exception-neutral**.

When exception-neutral code calls a client-provided function that may throw, it does one of two things:

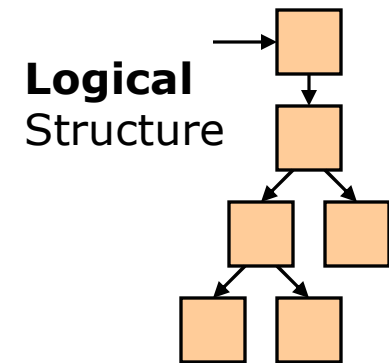
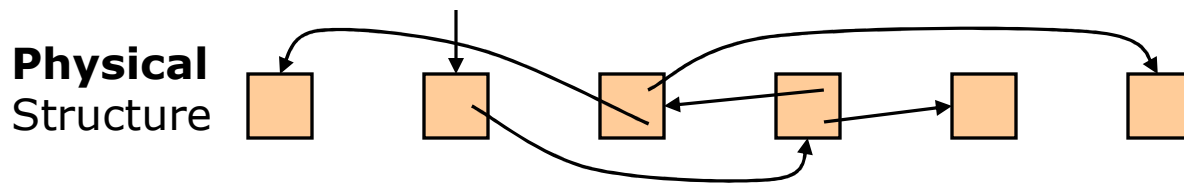
- Call the function outside a try block, so that any exceptions terminate our code immediately.
- Or, call the function inside a try block, then catch all exceptions, do any necessary clean-up, and re-throw.



Review

Node-Based Structures

A **node** is generally a small block of memory that is referenced via a pointer, and which may reference other nodes via pointers.

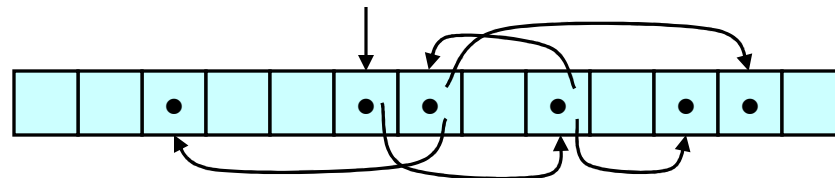


When we use a node-based data structure:

- We may not store data in contiguous memory.
- Memory-management overhead becomes significant.
- To “find”, we follow a chain of pointers. Look-up can be slow.
- Operations that require rearrangement can be very fast.

Node are usually separately allocated, but might be in an array.

- The main difference is who manages the memory, and when.



Node-Based Structures, cont'd

Classes

When we define a class to implement a node-based data structure, we usually need to define several classes:

- The main **container class**, representing the structure as a whole.
- A class representing a **node**.
 - This might be a private member of the main class.
 - Typically client code does not deal with this class.
- Possibly an **iterator** class.
 - Iterators to node-based structures are almost never pointers, because `operator++`, for example, needs to go to the next **logical** node, not the next **physical** memory location.

Despite the multiple classes being defined, we are implementing only a single package.

- Thus, multiple header & source files are generally not necessary.
- We can make all of these classes friends without breaking encapsulation.

Node-Based Structures, cont'd

Pointers & Ownership

Think of nodes as resources to be owned & managed.

- Who owns them?
 - Always document ownership (here: in the class invariants).
- Internal pointers in a node-based structure will usually be owning pointers.
 - A node is typically owned by the node that points to it.
 - Thus, a node's destructor should free all nodes that it points to.

This can make destroying a node-based structure **easy**.

- Each node is responsible for destroying the nodes it owns.
- Thus, to destroy the whole structure, all we need to do is destroy the nodes that are not owned by other nodes.
- And there is usually just one of these.

Linked Lists

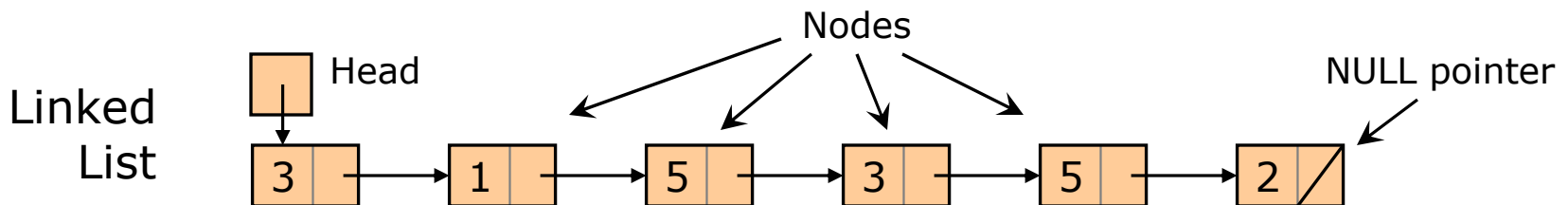
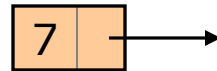
Introduction — The Idea

Our first node-based data structure is a **Linked List**.

- A Linked List is used to implement a Sequence.

What is a *Linked List*?

- A Linked List is composed of an ordered collection of nodes.
- Each node contains:
 - A data item.
 - A pointer to the next node.
- The **head** of the Linked List generally has a pointer to the first node.
 - And possibly other information (size, maybe?).
- There must be some way of identifying the last node.
 - Typically, we make its pointer NULL.



More precisely, this is a **Singly Linked List**.

- Later, we will see **Doubly Linked Lists**, which also have previous-node pointers.

Linked Lists

Introduction — Why? [1/3]

Why not always use (smart) arrays?

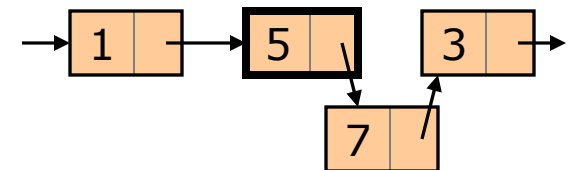
- One important reason: we can often insert and remove much faster with a Linked List.

Inserting

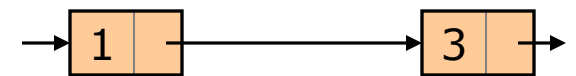
- Inserting an item at a given position in an array is $O(n)$.
- Inserting an item at a given position (think “iterator”) in a Linked List is $O(1)$.
 - Insert *after* node referenced by iterator.
- Example: insert a “7” after the bold node.

Removing

- Removing the item at a given position from an array is $O(n)$.
- Removing the item at a given position from a Linked List is $O(1)$.
 - We need an iterator to the *previous* item.
- Example: Remove the item in the bold node.



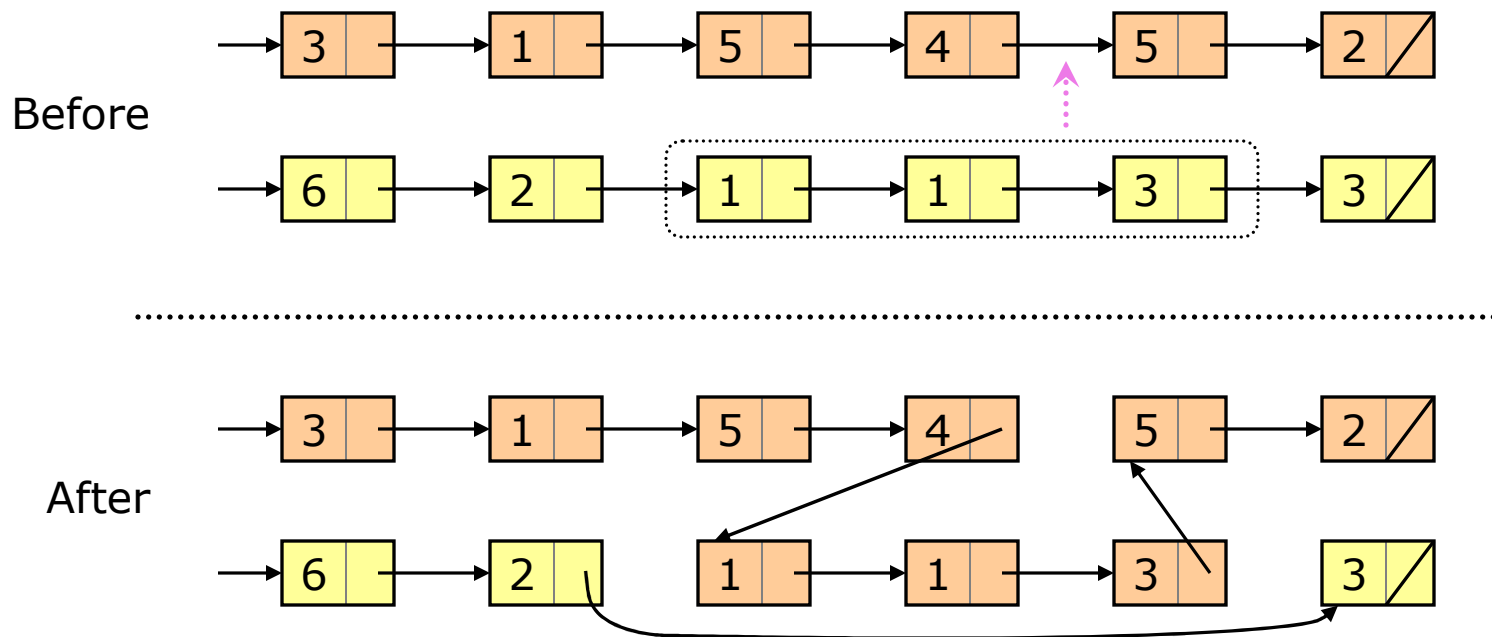
.....



Linked Lists

Introduction — Why? [2/3]

More interestingly, with Linked Lists, we can do a fast **splice**:

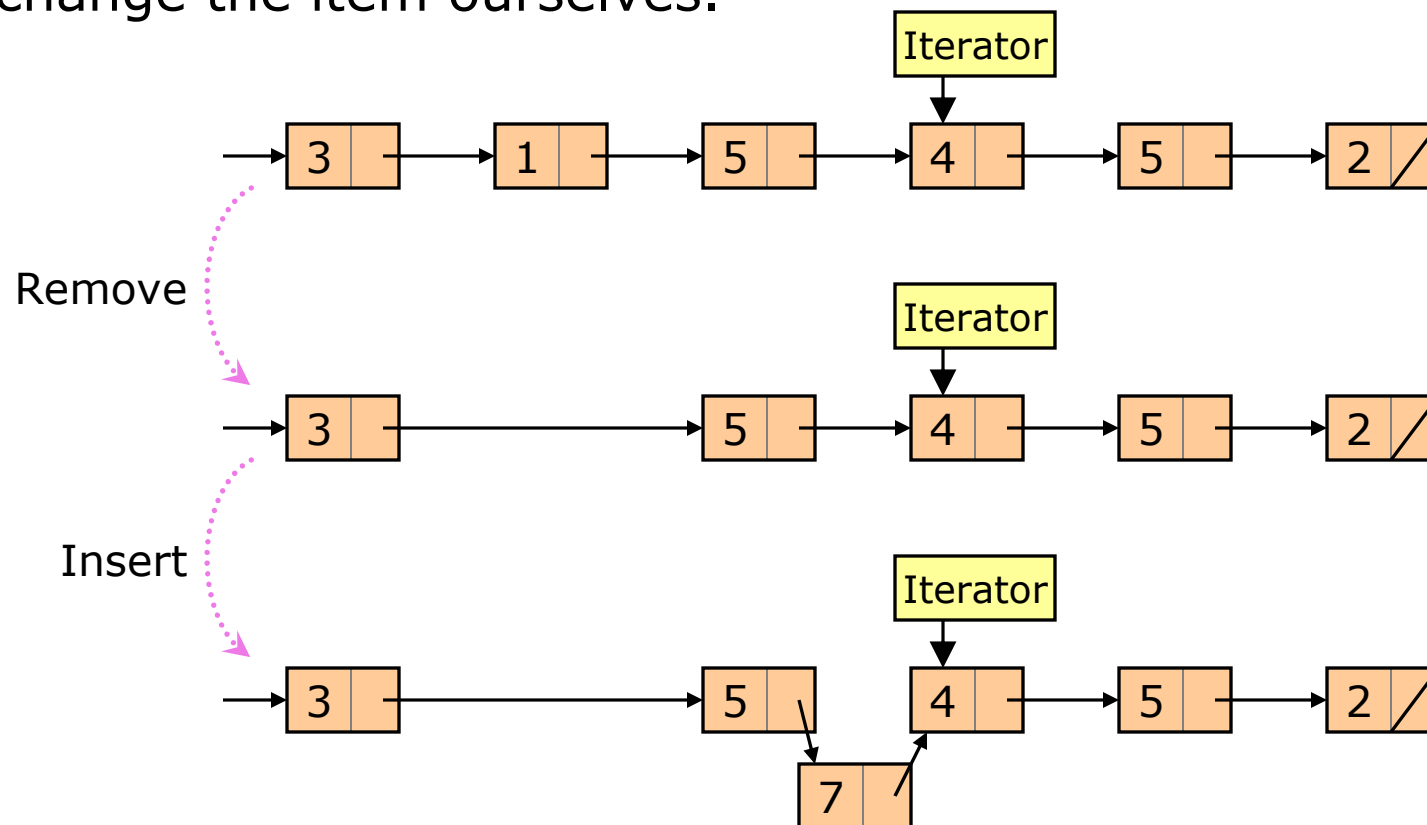


Note: If we allow for efficient splicing, then we cannot efficiently keep track of a Linked List's size.

Linked Lists

Introduction — Why? [3/3]

Further, with Linked Lists, iterators, pointers, and references to items will always stay valid and never change what they refer to, as long as the Linked List exists — unless we remove or change the item ourselves.



Linked Lists

Comparison With Arrays [1/4]

What is the order of each of the following when using (1) a smart-array implementation of a Sequence, and (2) a Linked-List implementation? Assume good design and good algorithms.

- Look-up an item by index.
- Search in a sorted Sequence.
- Search in an unsorted Sequence.
- Sort a Sequence.
- Insert an item at a given position.
- Remove an item at a given position.
- Splice part of one Sequence into another.
- Insert item at beginning of Sequence.
- Remove item at beginning of Sequence.
- Insert item at end of Sequence.
- Remove item at end of Sequence.
- Traverse a Sequence (iterate through all items, in order).
- Make a copy of an entire Sequence.
- Swap two Sequences.

What other issues arise when comparing the two data structures?

Linked Lists

Comparison With Arrays [2/4]

	Smart Array	Linked List
Look-up by index	$O(1)$	$O(n)$
Search sorted	$O(\log n)$	$O(n)$
Search unsorted	$O(n)$	$O(n)$
Sort	$O(n \log n)$	$O(n \log n)$
Insert @ given pos	$O(n)$	$O(1)^*$
Remove @ given pos	$O(n)$	$O(1)^*$
Splice	$O(n)$	$O(1)$
Insert @ beginning	$O(n)$	$O(1)$
Remove @ beginning	$O(n)$	$O(1)$
Insert @ end	$O(1)$ or $O(n)^{**}$ amortized const	$O(1)$ or $O(n)^{***}$
Remove @ end	$O(1)$	$O(1)$ or $O(n)^{***}$
Traverse	$O(n)$	$O(n)$
Copy	$O(n)$	$O(n)$
Swap	$O(1)$	$O(1)$

*For Singly Linked Lists, we mean inserting or removing just *after* the given position.

- Doubly Linked Lists can help.

** $O(n)$ if reallocation occurs. Otherwise, $O(1)$. Amortized constant time.

- Pre-allocation can help.

***For $O(1)$, need a pointer to the end of the list. Otherwise, $O(n)$.

- This is tricky.
- Doubly Linked Lists can help.

Find faster
with an array

Rearrange faster
with a Linked List

Linked Lists

Comparison With Arrays [3/4]

Other Issues

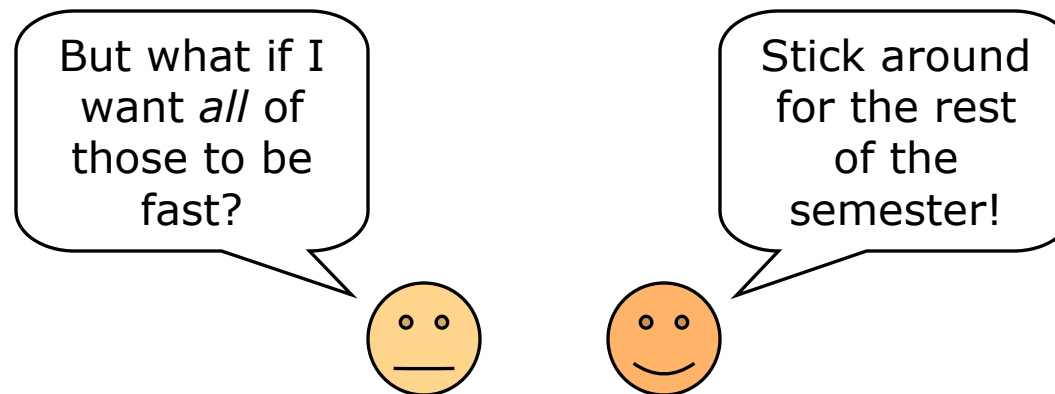
- ☹️ Linked Lists use **more memory**.
- ☹️ When order is the same, Linked Lists are almost always **slower**.
 - Arrays might be 2–10 times faster.
- ☹️ Arrays keep consecutive items in **nearby memory locations**.
 - Many algorithms have the property that when they access a data item, the following accesses are likely to be to the same or nearby items.
 - This property of an algorithm is called **locality of reference**.
 - Once a memory location is accessed, a memory cache will automatically load nearby memory locations. With an array, these are likely to hold nearby data items.
 - Thus, when a memory cache is used, an array can have a significant speed advantage over a Linked List, when used with an algorithm that has good locality of reference.
- 😊 With an array, iterators, pointers, and references to items can be **invalidated** by reallocation. Also, insert/remove can change the item they reference.

Linked Lists

Comparison With Arrays [4/4]

The Moral of the Story

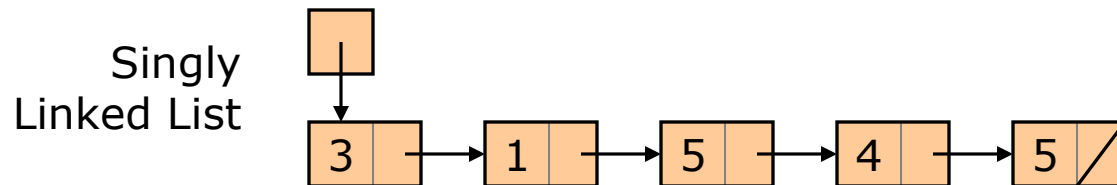
- Two kinds of design decisions affect the efficiency of code:
 - Choice of algorithm.
 - Choice of data structure.
- The latter often has the greater impact.
- Again:
 - Use arrays when we want fast look-up/search.
 - Use Linked Lists when we want fast insert & delete (by iterator).



Linked Lists

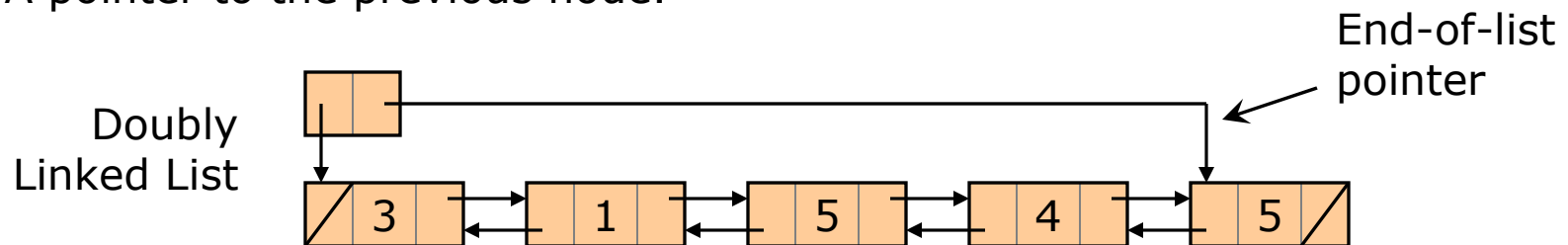
Variations — Doubly Linked Lists [1/3]

The kind of Linked List we have been discussing contains one pointer per node. Thus, it is called a **Singly Linked List**.



In a **Doubly Linked List**, each node has a data item & **two pointers**:

- A pointer to the next node.
- A pointer to the previous node.



Doubly Linked Lists often have an end-of-list pointer.

- This can be efficiently maintained, resulting in constant-time insert and remove at the end.

Linked Lists

Variations — Doubly Linked Lists [2/3]

Doubly Linked Lists have essentially all the advantages of (Singly) Linked Lists, plus some more.

- They allow efficient insert/remove at both ends of the list.
 - We can maintain an end-of-list pointer without trouble.
- They allow efficient insert-before-this-node and remove-this-node.
- They allow efficient backwards iteration.

However

- Doubly Linked Lists are a *little* slower than Singly Linked Lists.
 - Constant-time operations remain $O(1)$, but the constant is a little larger.
- Doubly Linked Lists still cannot do both splice and size efficiently.

The Bottom Line

- Doubly Linked Lists are generally considered to be a good basis for a general-purpose generic container type.
 - Singly-Linked Lists are not. Remember all those asterisks?

The C++ STL has Doubly Linked Lists: the `std::list` class template.

Linked Lists

Variations — Doubly Linked Lists [3/3]

	Smart Array	Doubly Linked List
Look-up by index	$O(1)$	$O(n)$
Search sorted	$O(\log n)$	$O(n)$
Search unsorted	$O(n)$	$O(n)$
Sort	$O(n \log n)$	$O(n \log n)$
Insert @ given pos	$O(n)$	$O(1)$
Remove @ given pos	$O(n)$	$O(1)$
Splice	$O(n)$	$O(1)$
Insert @ beginning	$O(n)$	$O(1)$
Remove @ beginning	$O(n)$	$O(1)$
Insert @ end	$O(1)$ or $O(n)^*$ amortized const	$O(1)$
Remove @ end	$O(1)$	$O(1)$
Traverse	$O(n)$	$O(n)$
Copy	$O(n)$	$O(n)$
Swap	$O(1)$	$O(1)$

With Doubly Linked Lists, we can get rid of most of our asterisks.

* $O(n)$ if reallocation occurs. Otherwise, $O(1)$. Amortized constant time.

- Pre-allocation can help.

Find faster
with an array

Rearrange faster
with a Linked List

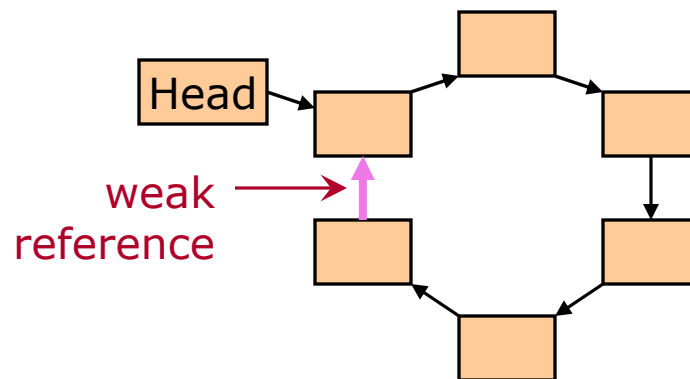
Linked Lists

Variations — Circular Linked Lists

Using nodes and pointers, we can arrange data structures in just about any way we want.

An interesting variation on a Linked List is a **Circular Linked List**.

- Here, we arrange our nodes in a circle.



- Ownership issues get trickier.
 - Destroy the head and ... nothing else gets destroyed?
 - One (somewhat icky) solution: use a **weak reference**.
- This structure generally not aimed at Sequence data.
 - What is it good for?

Linked Lists Implementation

Now we discuss how to implement a Linked List.

A sample Linked List implementation was sketched out in class:

- *Main class: `LinkedList`*
 - *Data member: `Node * head_.`*
- *Node class: `LinkedList::Node` (is a struct)*
 - *Private member of class `LinkedList`.*
 - *Data members: `value_type item_, Node * next_.`*
 - *Constructor taking `item` & `next`, for convenience.*
 - *Destructor: `deletes next_.`*
- *Iterator class: `LinkedList::iterator`*
 - *Public member of class `LinkedList`.*
 - *Data members: `Node * ptr_.`*
 - *Has `* operator`: returns `ptr_->item_.`*
 - *Has `++ operator`: does `ptr_ = ptr_-> next_.`*