Node-Based Structures
More on Linked Lists

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, March 29, 2013

Chris Hartman

Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu

Based on material by Glenn G. Chappell

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
Exception Safety [1/2]

Safety: Does function ever signal an error condition, and if so:

- Are data left in a usable state?
- Do we know something about that state?
- Are resource leaks avoided?

Basic Guarantee ← Minimum standard

- Data remain in a usable state, and resources are never leaked, even in the presence of exceptions.

Strong Guarantee ← Preferred

- If the operation throws an exception, then it makes no changes that are visible to the client code.

No-Throw Guarantee ← Required in some special situations

- The operation never throws an exception.

Each guarantee includes the previous one.

Review
Exception Safety [2/2]

To make sure code is exception-safe:

- Look at **every** place an exception might be thrown.
- For each, make sure that, if an exception is thrown, either
  - we terminate normally and meet our postconditions, or
  - we throw and meet our guarantees.

A bad design can force us to be unsafe.

- Thus, good design is part of of exception safety.
- An often helpful idea is that every module has exactly one well defined responsibility (the **Single Responsibility Principle**).
- In particular: A non-const member function should not return an object by value.

Review
Allocation & Efficiency [1/2]

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 (which averages over *all possible inputs*)
- Quintessential amortized-constant-time operation: insert-at-end for a well written smart array.
- Amortized constant time is not something we can easily compare with (say) logarithmic time.

Review
Allocation & Efficiency [2/2]

Improving **`SmArray`**

- 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: We can do amortized-constant-time insert-at-end.

Review
Generic Containers [1/2]

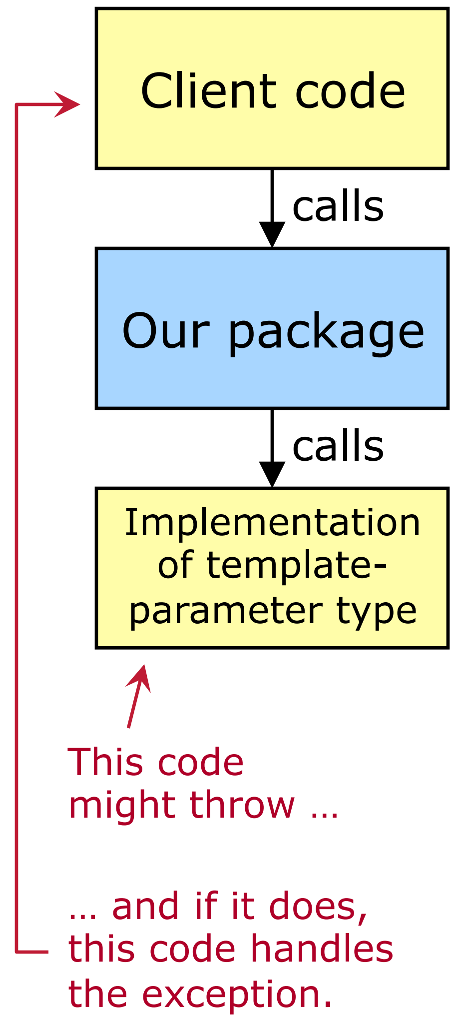
A **generic container** is a container that can hold a client-specified data type.

- 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
Generic Containers [2/2]

We can use catch-all, clean-up, re-throw to get both exception safety and exception neutrality.

```
arr = new MyType[size];
try
{
    std::copy(a, a+size, arr);
}
catch (...)
{
    delete [] arr;
    throw;
}
```

Called outside any `try` block. If this fails, we exit immediately, throwing an exception.

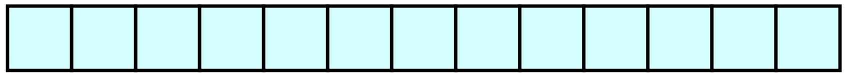
Called inside a `try` block. If this fails, we need to deallocate the array before exiting.

This helps us meet the Basic Guarantee (also the Strong Guarantee if this function does nothing else).

This makes our code exception-neutral.

Node-Based Structures
Introduction [1/2]

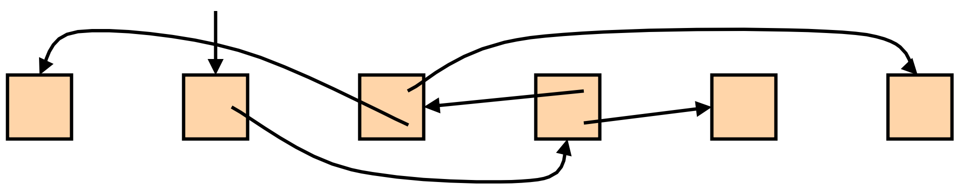
Our fundamental building block for data structures has been the **array**.



- Items are stored in contiguous memory locations.
- Look-up operations are usually very fast.
- Operations that require rearrangement (insert, remove, etc.) can be slow.

Now we begin looking at data structures built out of **nodes**.

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

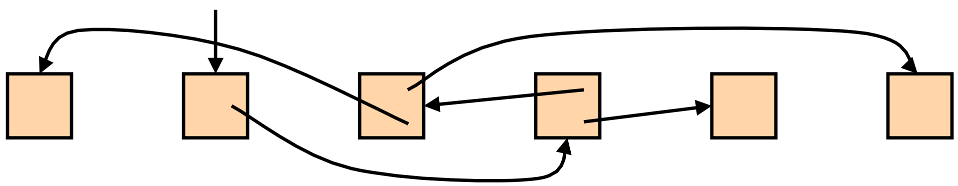


- Node-based structures do not necessarily store data in contiguous memory.
  - Memory-management changes significantly.
- To find a node, we follow a chain of pointers. Look-up can be slow.
- Operations that require rearrangement can be very fast.

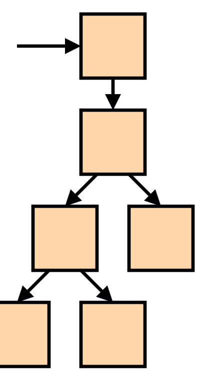
Node-Based Structures
Introduction [2/2]

When we draw pictures of node-based data structures, the positions of nodes in the picture usually have nothing to do with their positions in memory.

For example, if a structure is stored like this …

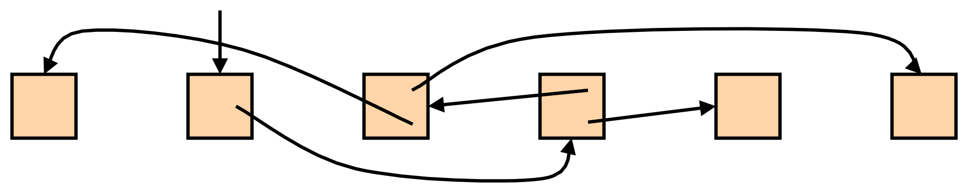


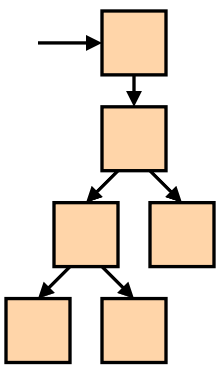
… then we might draw it like this:



Node-Based Structures
Implementation — Storage Options

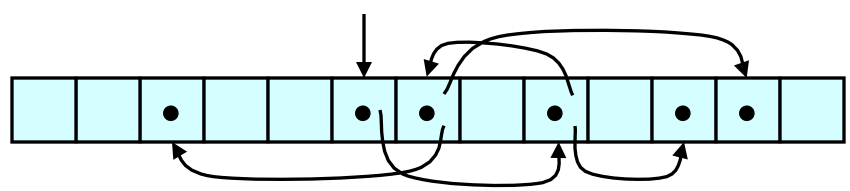
We normally store nodes in separately allocated blocks of memory.





However, we might allocate an **array of nodes**.

- Replace pointers with array indices, if desired.



Both of these have the above **logical** structure.

This is still a node-based data structure.

- It still has most of the pros & cons of node-based structures.
- The main differences involve **memory management**: who does it & when it gets done.

Node-Based Structures Implementation — Classes

When we define a class to implement a node-based data structure, we may wish to write 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.
  - If there are different kinds of nodes, then we might want several node classes. Sometimes inheritance can be helpful here.
- 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
Implementation — 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**.

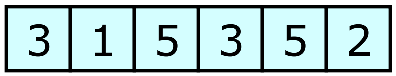
- Each node is responsible for destroying the nodes it owns.
- Thus, to destroy the whole structure, all we need to do is destroy the nodes that are not owned by other nodes.
- And there is usually just one of these.

More on Linked Lists
Refresher [1/2]

Earlier in the semester, we looked briefly at the simplest node-based structure: a **Linked List**.

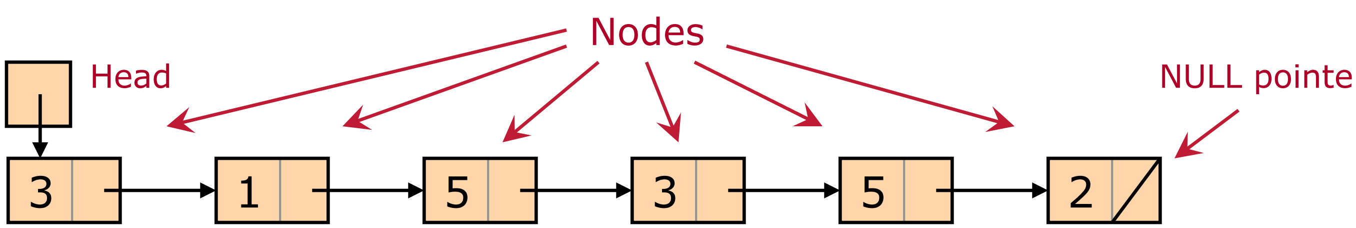
- Like an array, a Linked List is a structure for storing a sequence of items.

Array



- A Linked List is composed of **nodes**. Each has a single data item and a pointer to the next node.

Linked List



- These pointers are the **only** way to find the next data item. Thus, unlike an array, we cannot quickly skip to (say) the 100th item in a Linked List. Nor can we quickly find the previous item.
- A Linked List is a one-way sequential-access data structure. Thus, its natural iterator is a **forward iterator**, which has only the **++** operator.

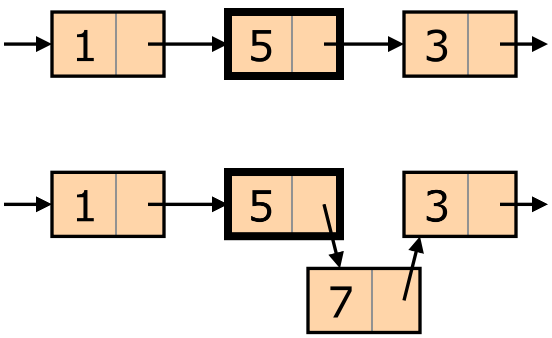
More on Linked Lists
Refresher [2/2]

Why not always use (smart) arrays?

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

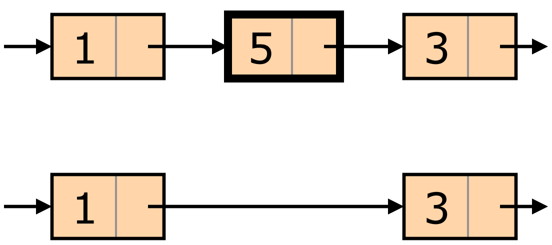
Inserting

- Inserting an item at a given position in an array is slow-ish.
- Inserting an item at a given position (think "iterator") in a Linked List is very fast.
- Example: insert a "7" after the bold node.



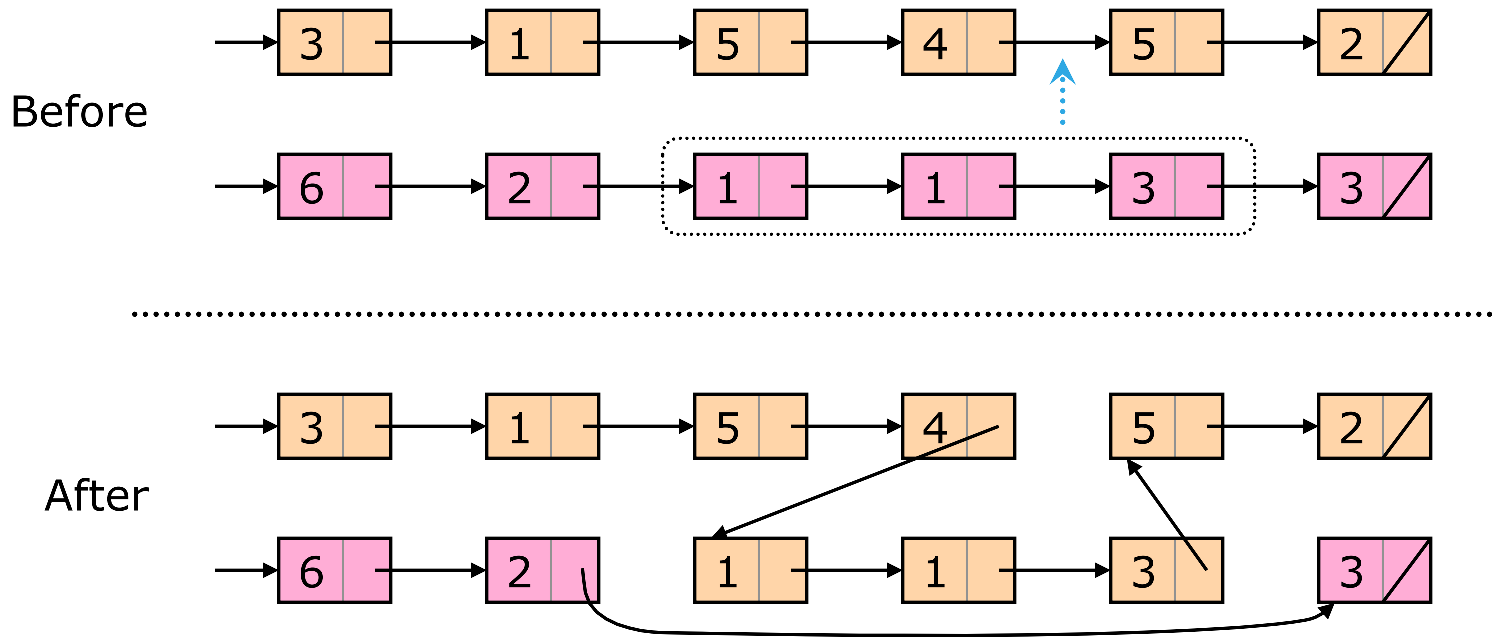
Removing

- Removing the item at a given position from an array *is also slow-ish.*
- Removing the item at a given position from a Linked List is very fast.
  - We need an iterator to the *previous* item.
- Example: Remove the item in the bold node.



More on Linked Lists
More Advantages [1/2]

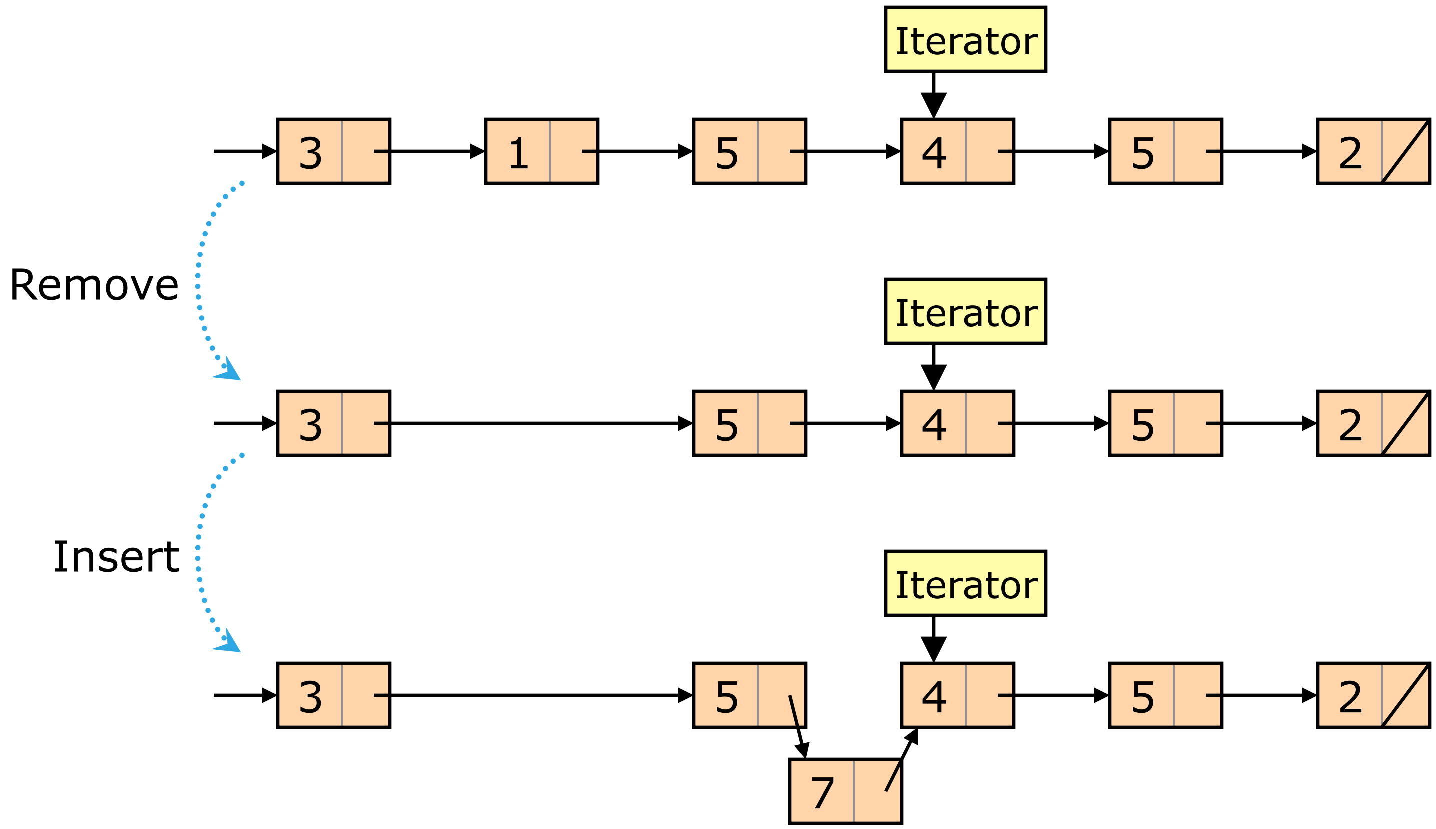
With Linked Lists, we can do a fast **splice**:



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

More on Linked Lists
More Advantages [2/2]

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.



More on Linked Lists
Comparison With Arrays [1/4]

1. What is the order of each of the following when using (a) a smart-array implementation of a Sequence, and (b) 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.
2. What other issues arise when comparing the two data structures?

More on Linked Lists
Comparison With Arrays [2/4]

	Smart Array	Linked List
Look-up by index	$\boldsymbol{O(1)}$	$\boldsymbol{O(n)}$
Search sorted	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(n)}$
Search unsorted	$O(n)$	$O(n)$
Sort	$O(n \log n)$	$O(n \log n)$
Insert @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$*
Remove @ given pos	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$*
Splice	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Insert @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
Remove @ beginning	$\boldsymbol{O(n)}$	$\boldsymbol{O(1)}$
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	$O(1)$	$O(1)$

Find faster with an array

Rearrange faster with a Linked List

*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.
- And, for remove @ end, it is basically impossible.
- Doubly Linked Lists can help.

More on 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, the CPU cache can **prefetch** nearby memory locations. With an array, these are likely to hold nearby data items.
  - Thus, because of cache prefetching, 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.

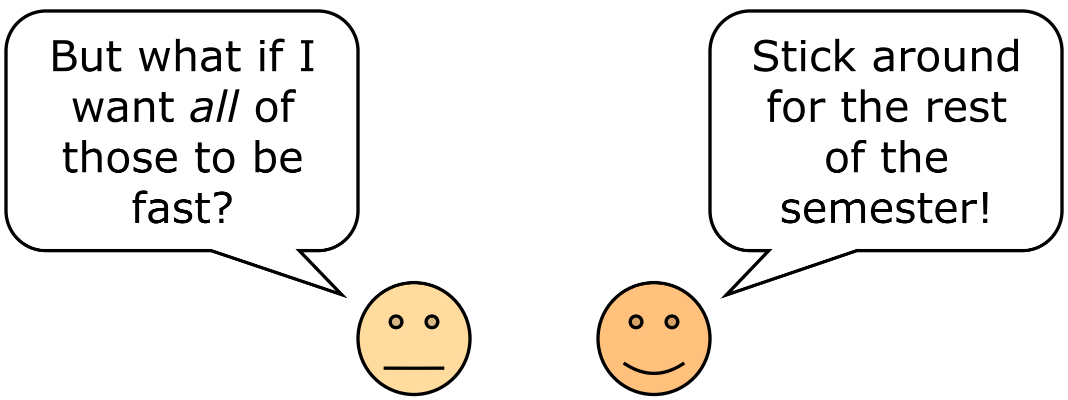
More on Linked Lists
Comparison With Arrays [4/4]

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.

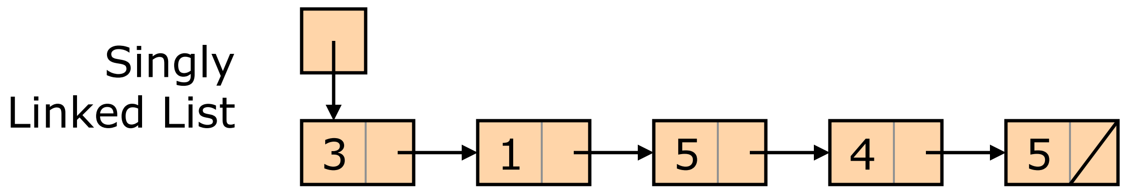
Again:

- Use arrays when we want fast look-up/search.
- Use Linked Lists when we want fast insert & delete (by iterator).



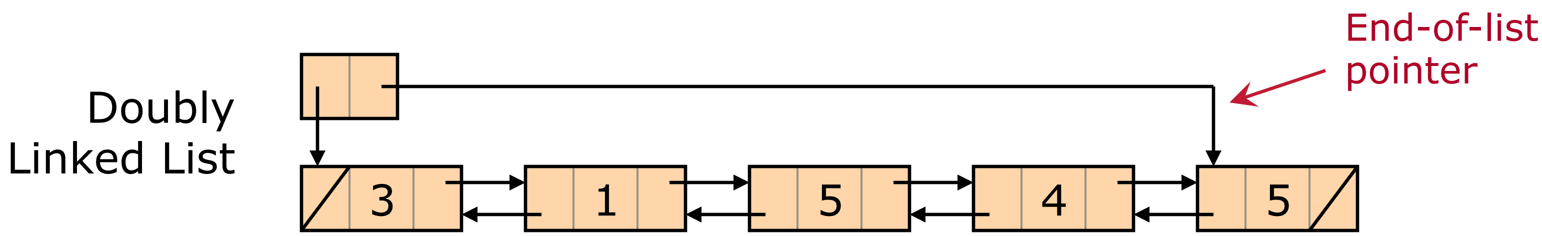
More on Linked Lists
Variations — Doubly Linked List [1/3]

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.

More on Linked Lists
Variations — Doubly Linked List [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 uses Doubly Linked Lists: the **`std::list`** class template.

(C++11 also has **`std::forward_list`**, a singly linked list.)

More on Linked Lists
Variations — Doubly Linked List [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)

Find faster with an array

Rearrange faster with a Linked List

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

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

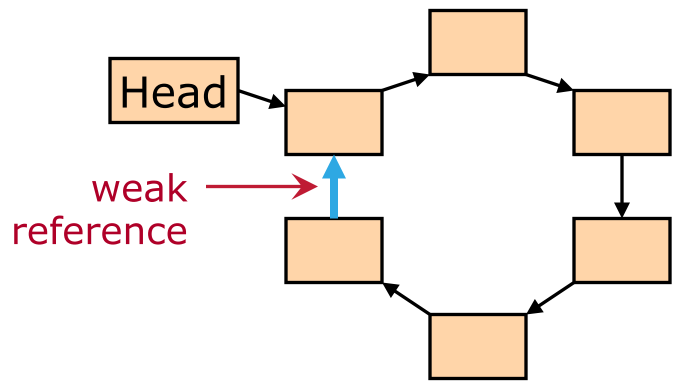
- Pre-allocation can help.

More on Linked Lists
Variations — Circular Linked List

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**.

More on Linked Lists
Implementation

Two approaches to implementing a Linked List:

- A Linked List package to be used by others.
- An internal-use Linked List: part of some other package, and not exposed to client code.

How would these be different?

- In particular, what classes might we define in each case?
  - In the first case, many classes: container, node, iterator.
  - In the second case, a node class, but possibly nothing else.

TO DO

- Update the internal-use Singly Linked List begun long ago, to include:
  - Exception-safety information.
  - An insert-at-beginning operation.

Sequences in the C++ STL
Generic Sequence Types — Introduction

The C++ STL has four generic Sequence container types.

- Class template **`std::vector`**.
  - A “smart array”.
  - Much like the Assignment 5 package, but with more member functions.
- Class template **`std::basic_string`**.
  - Much like **`std::vector`**, but aimed at character string operations.
  - Mostly we use **`std::string`**, which is really **`std::basic_string<char>`**.
  - Also **`std::wstring`**, which is really **`std::basic_string<std::wchar_t>`**.
- Class template **`std::list`**.
  - A Doubly Linked List.
    - Note: The Standard does not specify implementation. It specifies the semantics and order of operations. These were written with a Doubly Linked List in mind, and a D.L.L. is the usual implementation.
- Class template **`std::deque`**.
  - Deque stands for **D**ouble-**E**nded **QUE**ue.
  - Say “deck”.
  - Like **`std::vector`**, but a bit slower. Allows fast insert/remove at both beginning and end.

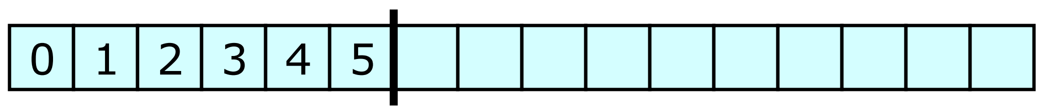
Sequences in the C++ STL
Generic Sequence Types — `std::deque` [1/4]

We are familiar with smart arrays and Linked Lists. How is **`std::deque`** implemented?

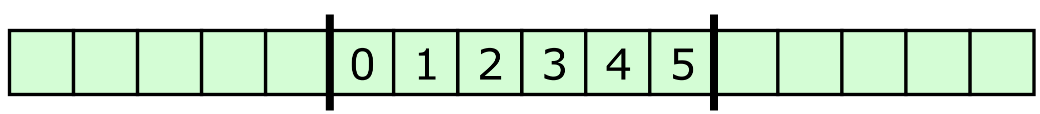
- There are two big ideas behind it.

First Idea

- A **`vector`** uses an array in which data are stored at the beginning.



  - This gives linear-time insert/remove at beginning, constant-time remove at end, and, if we do it right, amortized-constant-time insert at end.
- What if we store data in the middle? When we reallocate-and-copy, we move our data to the middle of the new array.

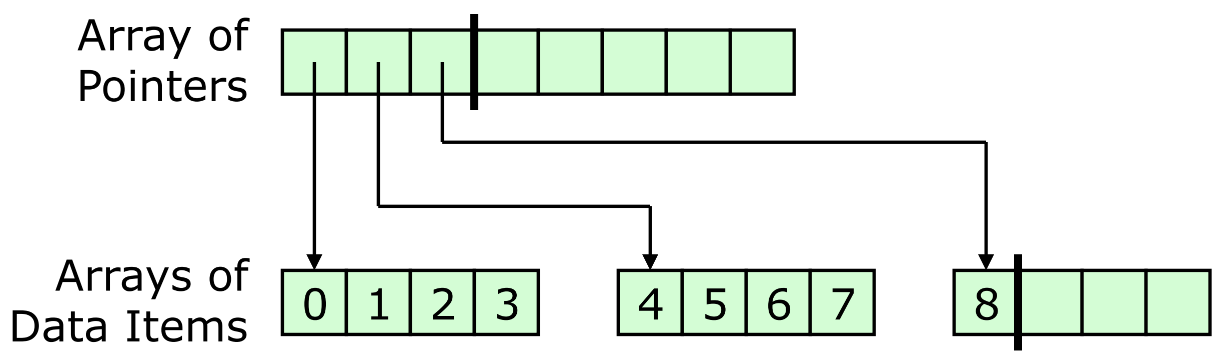


- Result: Amortized $O(1)$ insert, and $O(1)$ remove, at **both** ends.

Sequences in the C++ STL
Generic Sequence Types — `std::deque` [2/4]

Second Idea

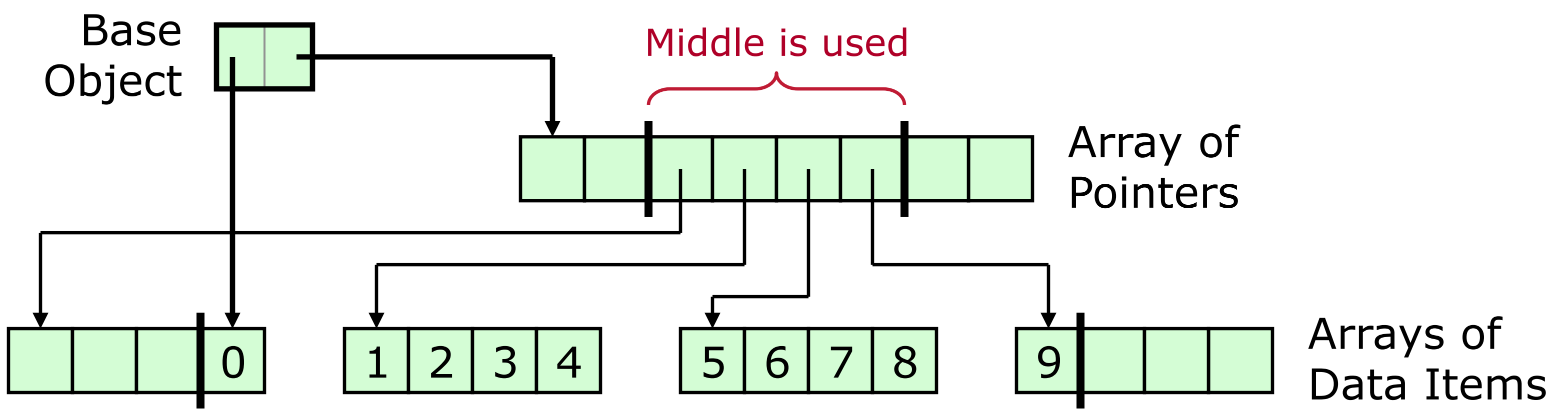
- Doing reallocate-and-copy for a `vector` requires calling either the copy constructor or copy assignment for **every data item**.
  - For large, complex data items, this can be time-consuming.
- Instead, let our array be an **array of pointers to arrays**, so that reallocate-and-copy only needs to move the pointers.
  - This still lets us keep most of the locality-of-reference advantages of an array, when the data items are small.



Sequences in the C++ STL
Generic Sequence Types — `std::deque` [3/4]

An implementation of **`std::deque`** typically uses both of these ideas.

- It probably uses an array of pointers to arrays.
  - This *might* go deeper (array of pointers to arrays of pointers to arrays).
- The arrays may not be filled all the way to the beginning or the end.
- Reallocate-and-copy moves the data to the middle of the new array of pointers.



Thus, **`std::deque`** is an array-ish container, optimized for:

- Insert/remove at either end.
- Possibly large, difficult-to-copy data items.

The cost is complexity, and a slower [but still $O(1)$] look-up by index.

Sequences in the C++ STL
Generic Sequence Types — `std::deque` [4/4]

Essentially, **`std::deque`** is an array.

- Iterators are random-access.
- But it has some complexity to it, so it is a slow-ish array.

Like **`vector`**, **`deque`** *tends* to keep consecutive items in nearby memory locations.

- So it tends to avoid cache misses, when used with algorithms having good locality of reference.

However:

- Insertions at the beginning do not require items to be moved up.
  - We speed up insert-at-beginning by allocating extra space before existing data.
- Reallocate-and-copy leaves the data items alone.
  - We also speeds up insertion by trading value-type operations for pointer operations.
  - Pointer operations can be much faster than value-type operations. A **`std::deque`** can do reallocate-and-copy using a raw memory copy, with no value-type copy ctor calls.

The Bottom Line

- A **`std::deque`** is generally a good choice when you need fast insert/remove at both ends of a Sequence.
- Especially if you also want fast-ish look-up.
- Some people also recommend **`std::deque`** whenever you will be doing a lot of resizing, but do not need fast insert/remove in the middle.

Sequences in the C++ STL
Generic Sequence Types — Efficiency [1/2]

We measure efficiency by counting steps. How do we count steps for a generic container type?

- We count both built-in operations and value-type operations.
- However, we typically expect that the most time-consuming operations are those on the value type.

The C++ Standard, on the other hand, counts **only** value-type operations.

- For example, "constant time" in the Standard means that at most a constant number of value-type operations are performed.

Sequences in the C++ STL
Generic Sequence Types — Efficiency [2/2]

	`vector`, `basic_string`	`deque`	`list`
Look-up by index	Constant	Constant	Linear
Search sorted	Logarithmic	Logarithmic	Linear
Insert @ given pos	Linear	Linear	Constant
Remove @ given pos	Linear	Linear	Constant
Insert @ beginning	Linear	Linear/ Amortized Constant*	Constant
Remove @ beginning	Linear	Constant	Constant
Insert @ end	Linear/ Amortized Constant**	Linear/ Amortized Constant*	Constant
Remove @ end	Constant	Constant	Constant

*Only a constant number of value-type operations are required.

- The C++ standard counts only value-type operations. Thus, it says that insert at the beginning or end of a **`std::deque`** is constant time.

**Constant time if sufficient memory has already been allocated.

All have $O(n)$ traverse, copy, and search-unsorted, $O(1)$ swap, and $O(n \log n)$ sort.

Sequences in the C++ STL
Generic Sequence Types — Common Features

All STL Sequence containers have:

- **`iterator`**, **`const_iterator`**
  - Iterator types. The latter acts like a pointer-to-const.
  - **`vector`**, **`basic_string`**, **`deque`** have random-access iterators.
  - **`list`** has bidirectional iterators.
- *iterator* **`begin()`**, *iterator* **`end()`**
- *iterator* **`insert(`***iterator*, *item***`)`**
  - Insert before. Returns position of new item.
- *iterator* **`erase(`***iterator***`)`**
  - Remove this item. Returns position of following item.
- **`push_back(`***item***`)`**
  - Insert at the end.
- **`clear()`**
  - Remove all items.
- **`resize(`***newSize***`)`**
  - Change the size of the container.
  - Not the same as **`vector::reserve`**, which sets capacity.

In Addition

`vector`, **`deque`**, **`list`** have:

- **`pop_back()`**
  - Remove at the end.
- *reference* **`front()`**, *reference* **`back()`**
  - Return reference to first, last item.

`deque`, **`list`** have:

- **`push_front(`***item***`)`**, **`pop_front()`**
  - Insert & remove at the beginning.

`vector`, **`basic_string`**, **`deque`** have:

- *reference* **`operator[](`***index***`)`**
  - Look-up by index.

`vector` has:

- **`reserve(`***newCapacity***`)`**
  - Sets capacity to at least the given value.

And there are other members …