Node-Based Structures
More on Linked Lists

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, October 23, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Review

Review
Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

Unit Overview
Data Handling & Sequences

Major Topics

- ✓ Data abstraction
- ✓ Introduction to Sequences
- ✓ Interface for a smart array
- ✓ Basic array implementation
- ✓ Exception safety
- ✓ Allocation & efficiency
- ✓ Generic containers

Smart Arrays

- Node-based structures
- More on Linked Lists

Linked Lists

- Sequences in the C++ STL
- Stacks
- Queues

Review
Allocation & Efficiency [1/2]

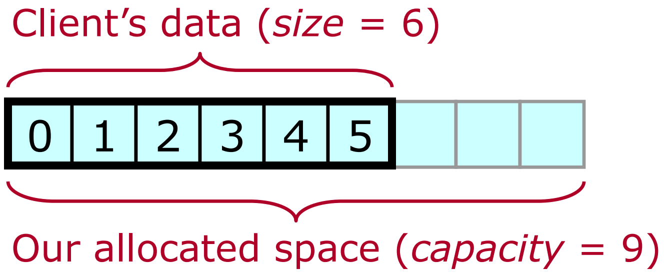
Originally, our resizable array class had two data members:

- Size
- Pointer to array

This design is inappropriate for a resizable array. It cannot keep track of the amount of extra allocated space, if any. Thus, it must do a **reallocate-and-copy** *every time* it is resized larger.

The fix is to add an additional data member to hold the **capacity**: the amount of allocated space.

When this is done, the **size** is still the space used by the client's dataset, but it may be smaller than the capacity.


Client's data (*size* = 6)
0
1
2
3
4
5
Our allocated space (*capacity* = 9)

Review
Allocation & Efficiency [2/2]

An operation is **amortized constant-time** if *k* consecutive 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*.
- Amortized constant-time is not something we can compare with (say) logarithmic time.

This is our last efficiency-related terminology.

Insert-at-end for a well written smart array is amortized constant-time:

- Store both currently used size and allocated capacity.
- When space runs out, reallocate-and-copy with capacity increased by a **constant factor** (doubled, for example).

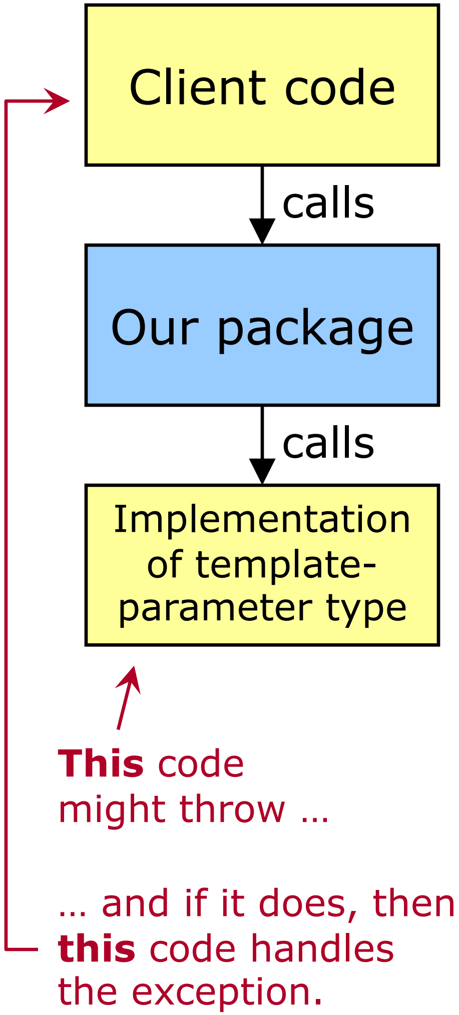
Note. Insert-at-end is still linear time!

Review
Generic Containers

When we write a generic container, our value type is specified by the client code. Its member functions may throw. We generally have no idea how to handle these exceptions. Only the client code knows.

A function that allows exceptions thrown by client-provided code to propagate unchanged to the caller, is said to be **exception neutral**.

This is our last exception-related terminology.



Client code

calls

Our package

calls

Implementation of template-parameter type

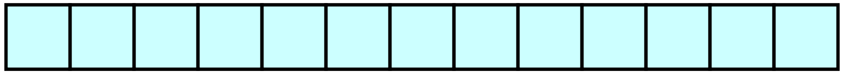
This code might throw …

… and if it does, then **this** code handles the exception.

Node-Based Structures

Node-Based Structures
Introduction [1/3]

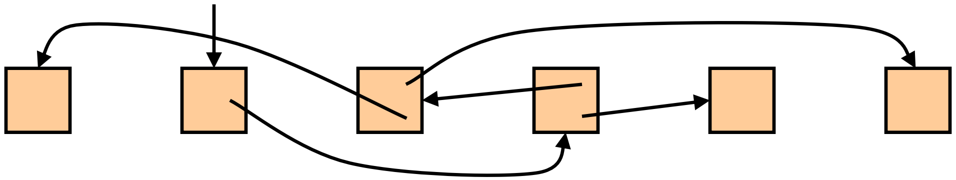
Our primary building block for data structures has been the array.



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

Many data structures are, instead, built out of *nodes*.

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

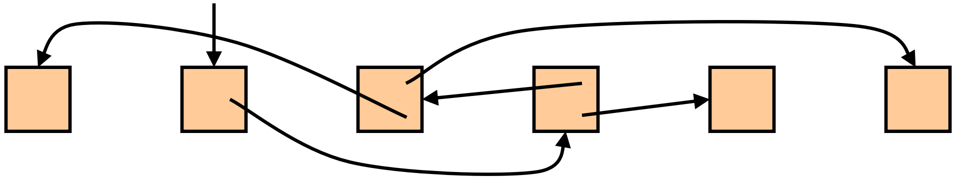


- Memory-management changes significantly.
- To find a node, follow a chain of pointers. Look-up can be slow.
- Operations that require rearrangement *might* be very fast.

Node-Based Structures
Introduction [2/3]

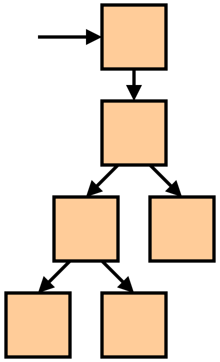
When we draw pictures of node-based data structures, the positions of nodes in the picture usually have nothing to do with their positions in memory.

For example, if a structure is stored like this …



Physical Structure

… then we might draw it like this:



Logical Structure

Node-Based Structures Introduction [3/3]

Think of nodes as resources to be owned & managed.

- Who owns them?
  - It is good to document ownership (here: in the class invariants).
- Internal pointers in a node-based structure are typically owning pointers.
  - A node is typically owned by the node that points to it.
  - A node’s destructor typically frees 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.
- To destroy the whole structure, all we need to do is destroy the nodes that are not owned by other nodes.
- And there is often just one of these.

Handling ownership is particularly easy if we use *smart pointers* …

Node-Based Structures
Smart Pointers — Overview

In 2011, **smart pointer** class templates were added to the C++ Standard Library. These use RAII to handle ownership of dynamic objects automatically.

`std::unique_ptr<T>` (`<memory>`)

- One-owner-at-a-time ownership of a dynamic object of type `T`.
- The destructor of an owning `unique_ptr` destroys the object pointed to.
- Movable but not copyable. Moving transfers ownership.

`std::shared_ptr<T>` (`<memory>`)

- Allows shared ownership of a dynamic object of type `T`.
- Uses a **reference count**. Destroys object when the count hits `0`.
  - "The last one to leave turns out the lights."
- Copyable. Copying grants shared ownership.

Node-Based Structures
Smart Pointers — Creation

A default-constructed smart pointer does not point to anything.

```
unique_ptr<Foo> unp;
```

We *can* pass a pointer returned by `new` to the constructor of a `unique_ptr`/`shared_ptr`, which will then do the `delete` for us.

```
Foo * p = new Foo(5, "xy", 3.2);  // Do not delete p
unique_ptr<Foo> unp(p);           // unp does the delete
```

But there is a better way: call `make_unique`/`make_shared`, passing constructor arguments.

Arguments for `Foo` constructor.

```
auto unp = make_unique<Foo>(5, "xy", 3.2);
    // Both new and delete are done for us
```

Node-Based Structures
Smart Pointers — Use like a Pointer

Dereference a `unique_ptr/shared_ptr` just like a regular pointer.

```
cout << *unp << endl;
```

The arrow operator is also available.

```
unp->bar();  // Member function of the referenced object
```

If you need a regular (non-smart) pointer to the object a `unique_ptr/shared_ptr` points to, call member function `get`.

```
Foo * p = unp.get();
p->bar();
```

Node-Based Structures
Smart Pointers — Empty

A `unique_ptr`/`shared_ptr` that does not point to anything is said to be **empty**. This corresponds to a null pointer.

Test whether a smart pointer is empty by treating it like a `bool`.

```
if (unp) // Is unp nonempty (does it point to anything)?
{
    unp->bar();
}
```

Similarly, "`if (!unp)`" checks whether `unp` is empty.

Unlike with a regular pointer, this test works on a smart pointer that has not been set to a particular value—including a default-constructed smart pointer.

Member function `reset` relinquishes ownership *early* and makes the smart pointer empty. If the referenced object had only one owner, then it is destroyed.

```
unp.reset();  // Relinquish ownership of object;
              //  unp is now empty
```

Node-Based Structures
Smart Pointers — Transferring & Sharing Ownership

To transfer ownership, pass or return a `unique_ptr` by value. This `unique_ptr` must be an Rvalue; `std::move` may be required.

```
unique_ptr<Foo> makeAFoo()
{
    return make_unique<Foo>(5, "xy", 3.2);
}
```

`make_unique` returns an Rvalue, so we do not need to use `std::move` here.

```
auto unp = makeAFoo();
```

If we are not transferring ownership, then a `unique_ptr` should be passed by reference or reference-to-const.

A `shared_ptr` may be passed by value arbitrarily. Passing by value shares ownership.

Node-Based Structures
Smart Pointers — Philosophy

Programmers have found that shared ownership is rarely needed. It is true that, whenever you might use `unique_ptr`, it will also work to use `shared_ptr`. On the other hand, using `unique_ptr` helps the compiler find bugs for you (and it is more efficient!).

Suggestions

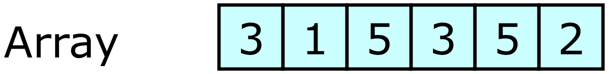
- When you want a smart pointer, start by using `unique_ptr` and `make_unique`.
- Pass a smart pointer by reference or reference-to-const when you do not want to transfer/share ownership.
- If code does not compile because it tries to copy a `unique_ptr`:
  - If you want to transfer ownership, then you might simply need to wrap the `unique_ptr` in `std::move`.
  - If you do not want to transfer ownership, then you can call `get` to obtain a non-owning regular pointer and use that instead.
  - If it turns out that you really need shared ownership, then do a global search/replace: `unique` → `shared`

More on Linked Lists

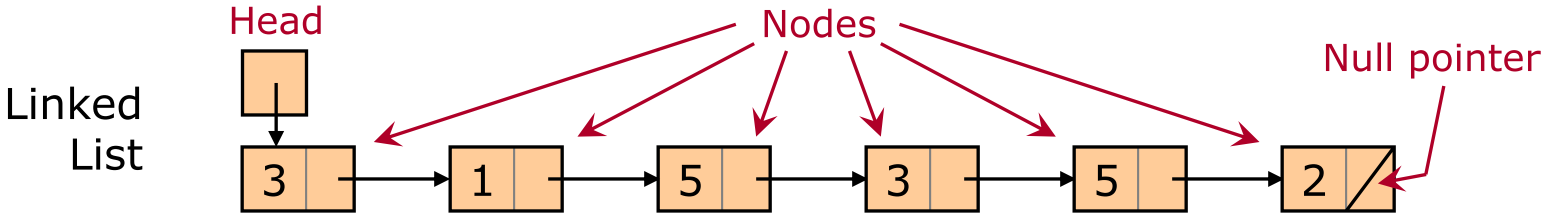
More on Linked Lists
Refresher [1/2]

Earlier, we looked briefly at a simple node-based structure: the **Linked List** (or **Singly Linked List**).

- Like an array, a Linked List stores a sequence of data items.



- A Linked List is made of **nodes**. Each has a single data item and a pointer to the next node, or a null pointer at the end of the list.



- These pointers are the only way to find the next item. Unlike with an array, we cannot quickly find (say) the 100,000th item in a Linked List. Nor can we quickly find the previous item.
- A Linked List is a one-way sequential-access structure. So its iterators are **forward iterators**, which have only the ++ operator.

More on Linked Lists
Refresher [2/2]

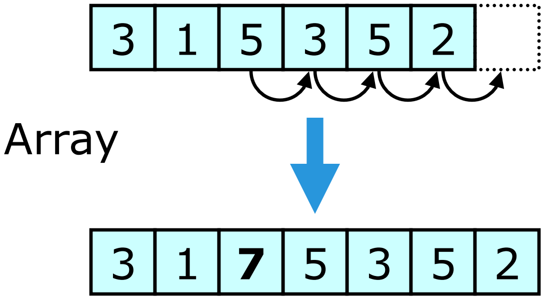
Why not always use (smart) arrays?

One reason: Linked Lists support fast insertion.

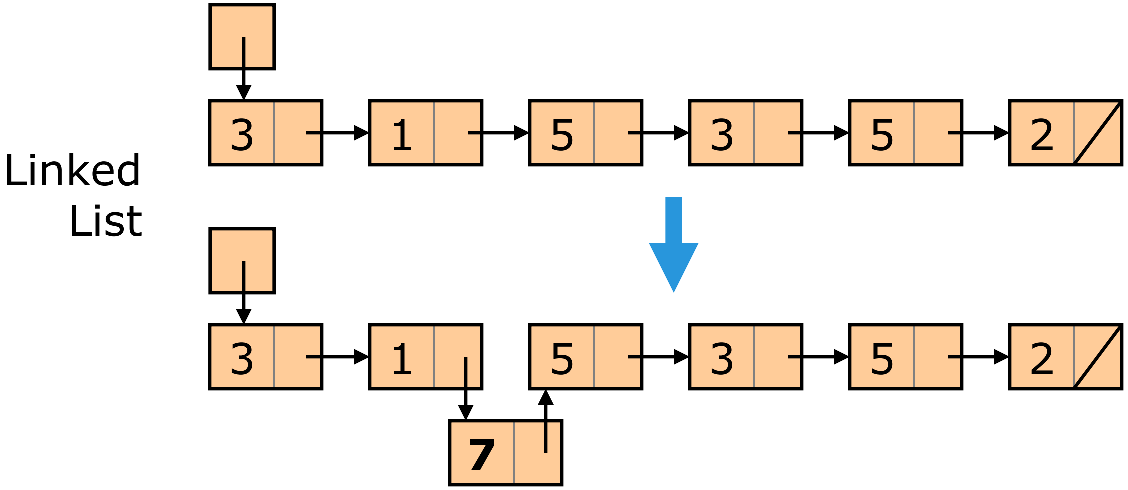
Suppose we have a sequence 3, 1, 5, 3, 5, 2.
We wish to insert a 7 before the first 5.

With an array, we move all later items up.

With a Linked List, *if we know the proper location*, insertion is very fast.

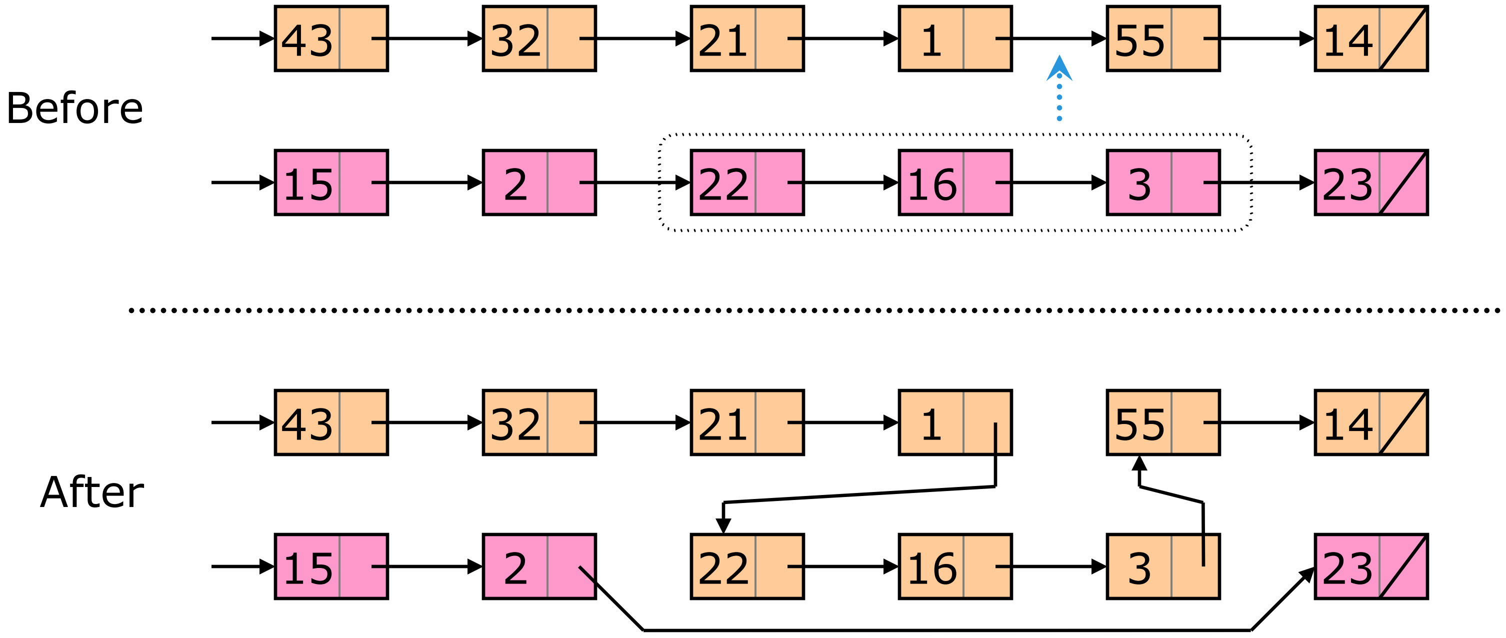


For long sequences, the speed difference can be huge.



More on Linked Lists
More Advantages [1/2]

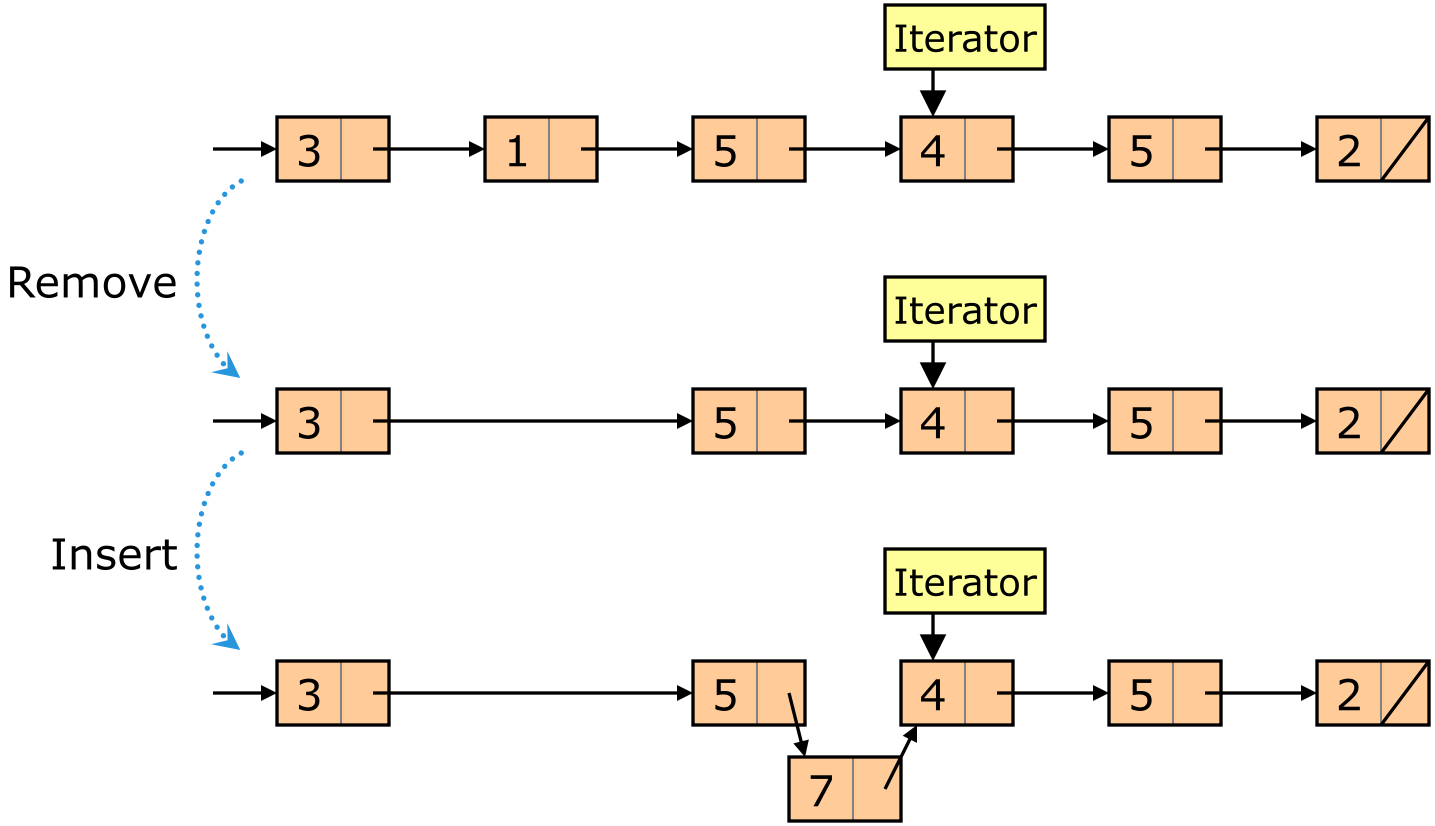
With Linked Lists, we can also do a fast **splice**:



Note that, 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 referenced.



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

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 ("position": think *iterator*)
- Remove an item at a given position.
- Splice part of one Sequence into another.
- Insert item at the beginning of a Sequence.
- Remove item at the beginning of a Sequence.
- Insert item at the end of a Sequence.
- Remove item at the end of a Sequence.
- Traverse a Sequence (iterate through all items, in order).

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	**$O(1)$**	**$O(n)$**
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	$O(n)$** amortized const	$O(1)$ or $O(n)$***
Remove @ end	$O(1)$	$O(1)$ or $O(n)$***
Traverse	$O(n)$	$O(n)$

Find faster
with an array

Rearrange faster
with a Linked List

*For Singly Linked Lists, insert/remove just *after* the given position.

- Doubly Linked Lists can help.

**$O(1)$ if no reallocate-and-copy.

- Pre-allocate to ensure this.

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

- This can be tricky.
- And, for remove @ end, it is mostly 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 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.
  - 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]

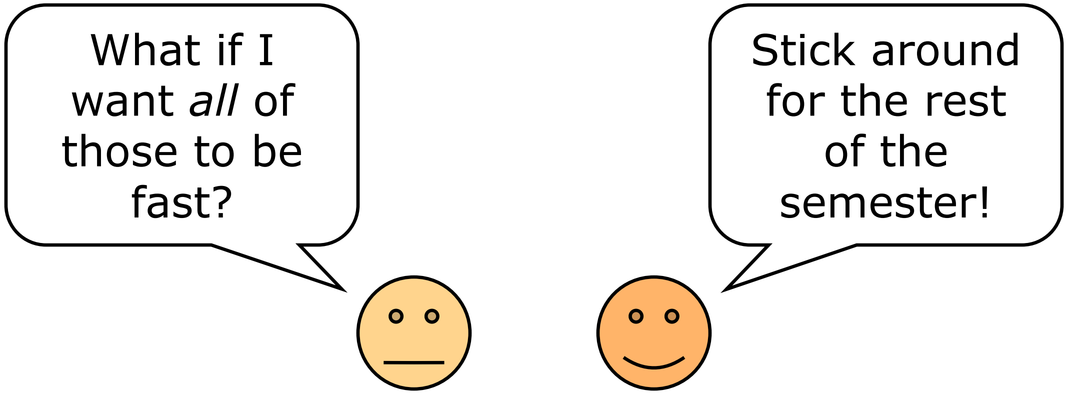
A Moral for Our Story

- Two kinds of design decisions affect the efficiency of code:
  - Deciding how to *process* data (algorithms).
  - Deciding how to *store* data (data structures).

The latter often has the greater impact.

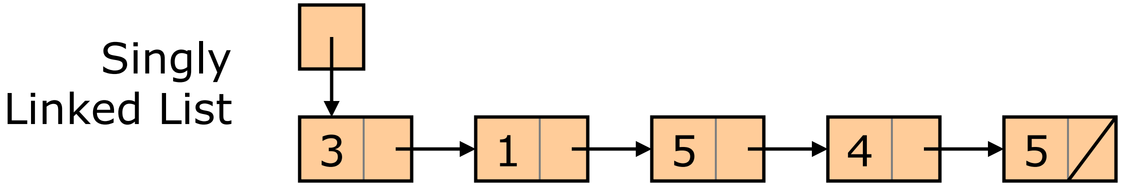
Very rough guidelines:

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

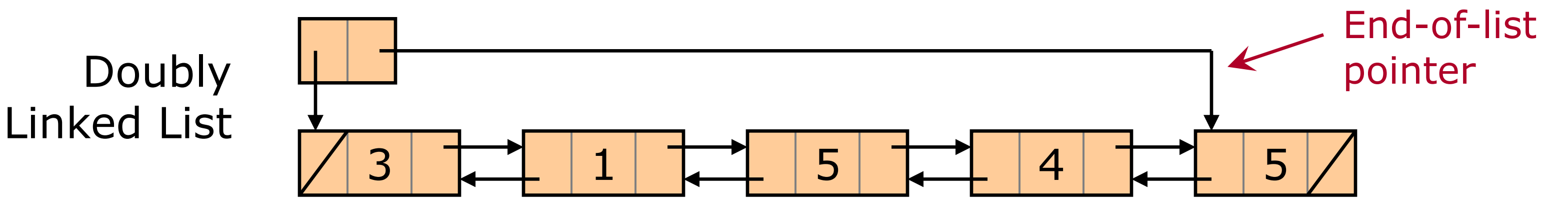

What if I want *all* of those to be fast?
Stick around for the rest of the semester!

More on Linked Lists
Doubly Linked Lists [1/3]

We have been discussing the **Singly Linked List**.



Recall: in a **Doubly Linked List**, each node has two pointers, next node (null at the end) and previous node (null at the beginning).



End-of-list pointer

A Doubly Linked List typically has an end-of-list pointer. This can be efficiently maintained, resulting in constant-time insert and remove at the end of the list.

More on Linked Lists
Doubly Linked Lists [2/3]

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

- An end-of-list pointer can be maintained without trouble.
- They allow efficient insert/remove at both ends of the list.
- They allow efficient insert-before-this-node and remove-this-node.
- They allow efficient reverse iteration.

However, Doubly Linked Lists are a *little* slower.

- Constant-time operations remain $O(1)$, but the constant is larger.

The Bottom Line

- Doubly Linked Lists are a good basis for a general-purpose generic container type.
- Singly Linked Lists are more special-purpose (all those asterisks).

More on Linked Lists
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	$O(n)$* amortized const	$O(1)$
Remove @ end	$O(1)$	$O(1)$
Traverse	$O(n)$	$O(n)$

With Doubly Linked Lists, we can eliminate asterisks.

*$O(1)$ if no reallocate-and-copy.

- Pre-allocate to ensure this.

Find faster
with an array

Rearrange faster
with a Linked List

More on Linked Lists
TO BE CONTINUED …

More on Linked Lists will be continued next time.