Node-Based Structures
More on Linked Lists

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, October 25, 2024

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Data Handling & Sequences

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

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.
- Things we need to be able to do:
  - 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.

Review
Allocation & Efficiency [1/2]

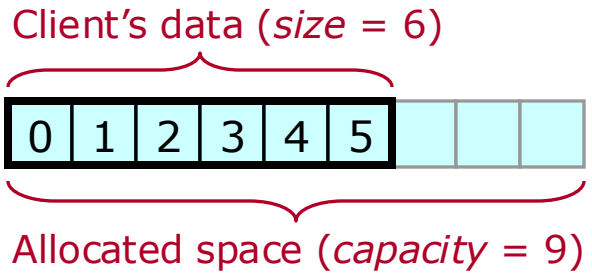
Originally, our resizable array class had two data members:

- 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.



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).

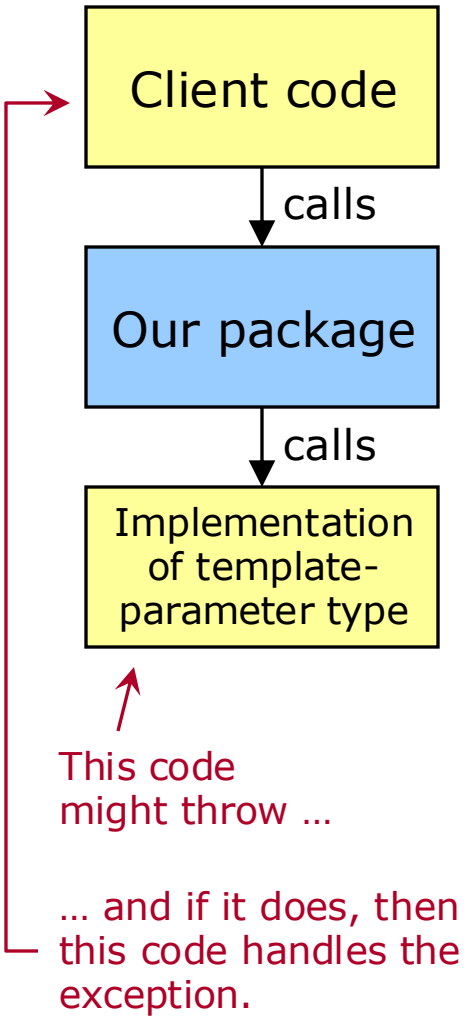
Insert-at-end is still linear time!

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.



Node-Based Structures

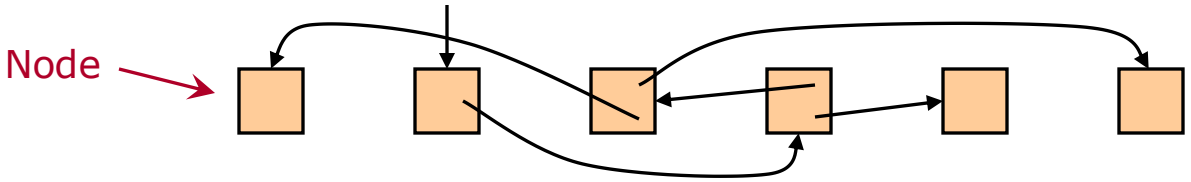
Node-Based Structures
Introduction [1/3]

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.



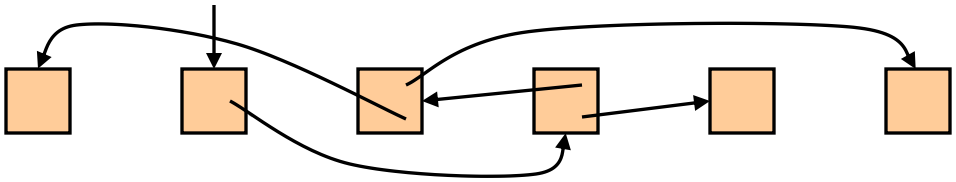
Most of the data structures we look at from now on will be node-based.

- 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]

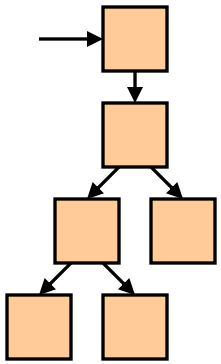
When we draw pictures of node-based data structures, the positions of nodes in the picture usually have nothing to do with their positions in memory.

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]

Internal pointers in a node-based structure are often owning pointers.

Think of nodes as resources to be owned & managed.

Therefore, it can be convenient to implement node-based structures in C++ using *smart pointers*, which handle clean-up automatically.

See the following slides.

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.

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. Then *never* do new or delete.

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

Arguments for `Foo` constructor.

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
```

To get an ordinary (non-smart) pointer to the object a smart pointer points to, call member function `get`. This does not affect ownership issues at all. The pointer returned is non-owning.

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

Node-Based Structures
Smart Pointers — Null

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

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

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

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

Unlike regular pointers, this works on a default-constructed `unique_ptr`/`shared_ptr` that has not been set to a value.

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

```
unp.reset();  // Relinquish ownership;
              //  unp is now null
```

I almost never use `reset`.

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);
}

auto unp = makeAFoo();
```

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

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.

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.

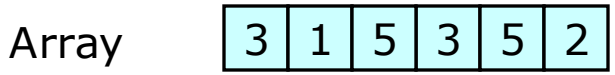
Suggestions

- When you want a smart pointer, start by using `unique_ptr`.
- To transfer ownership, pass/return by value.
- Otherwise, pass by reference or reference-to-const, OR call `get` and pass a regular pointer.
- 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`

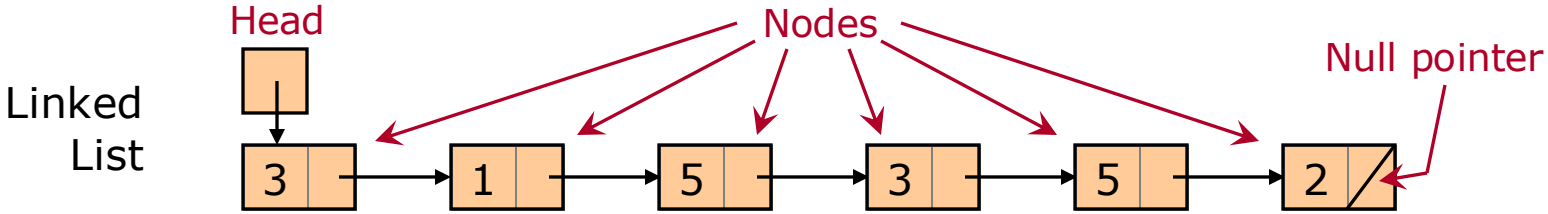
More on Linked Lists

More on Linked Lists
Refresher [1/2]

Earlier, we looked at a node-based structure: the (**Singly**) **Linked List**. Like an array, a Linked List stores a sequence of items.



Each Linked List node has a single data item and a pointer to the next node, or a null pointer at the end of the list. We keep track of a Linked List using its **head** pointer.



A Linked List is a forward-only sequential-access structure. To find items, we follow pointers though the list. We cannot quickly find the 100,000th item. Nor can we quickly find the previous item.

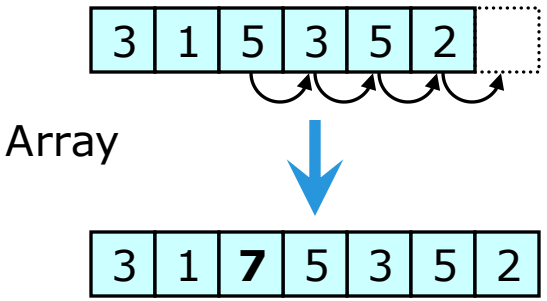
More on Linked Lists
Refresher [2/2]

Why not always use (smart) arrays?

One reason: Linked Lists support fast insertion.

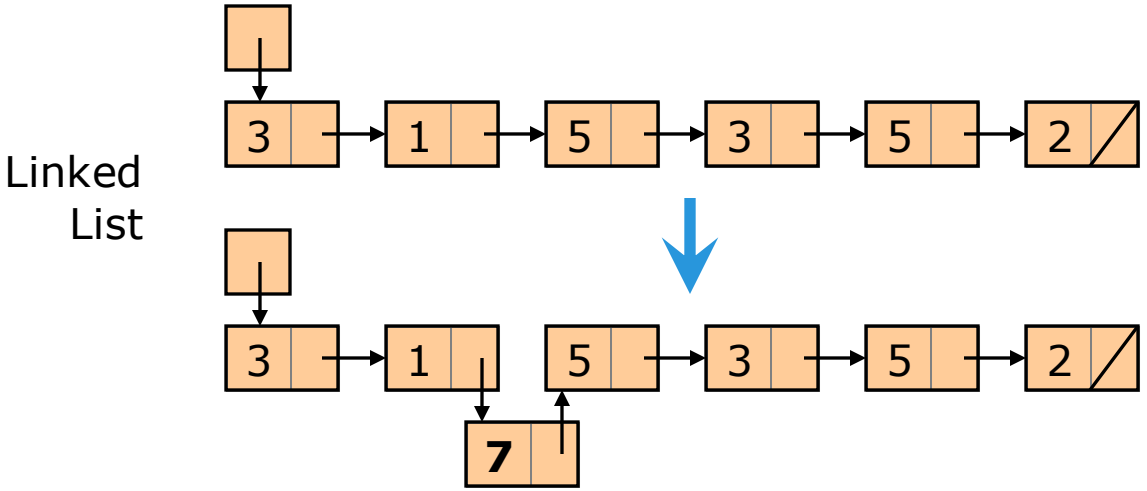
Suppose we have a sequence 3, 1, 5, 3, 5, 2.
We wish to insert a 7 before the first 5.

With an array, we move all later items up.
For a large array this can be very slow.



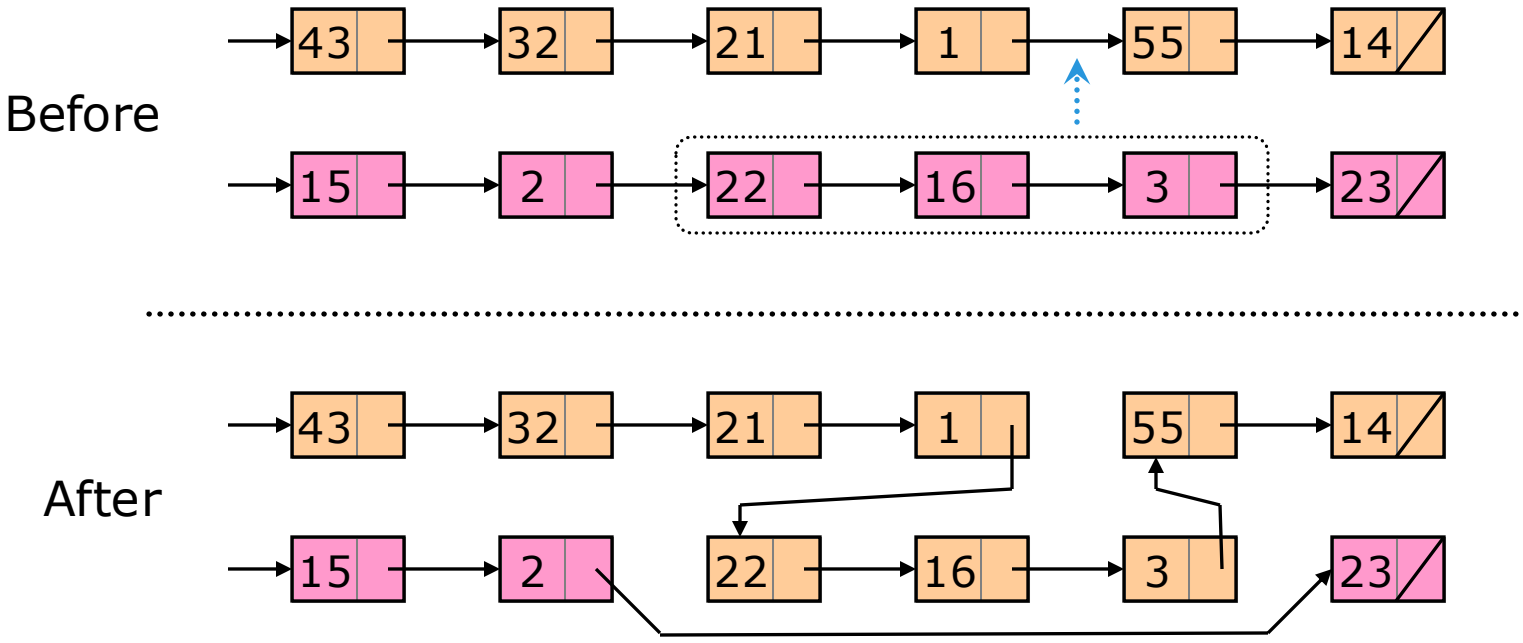
With a Linked List, *if we know the location*, insertion is fast.

For large datasets, the speed-up can be huge.



More on Linked Lists
More Advantages [1/2]

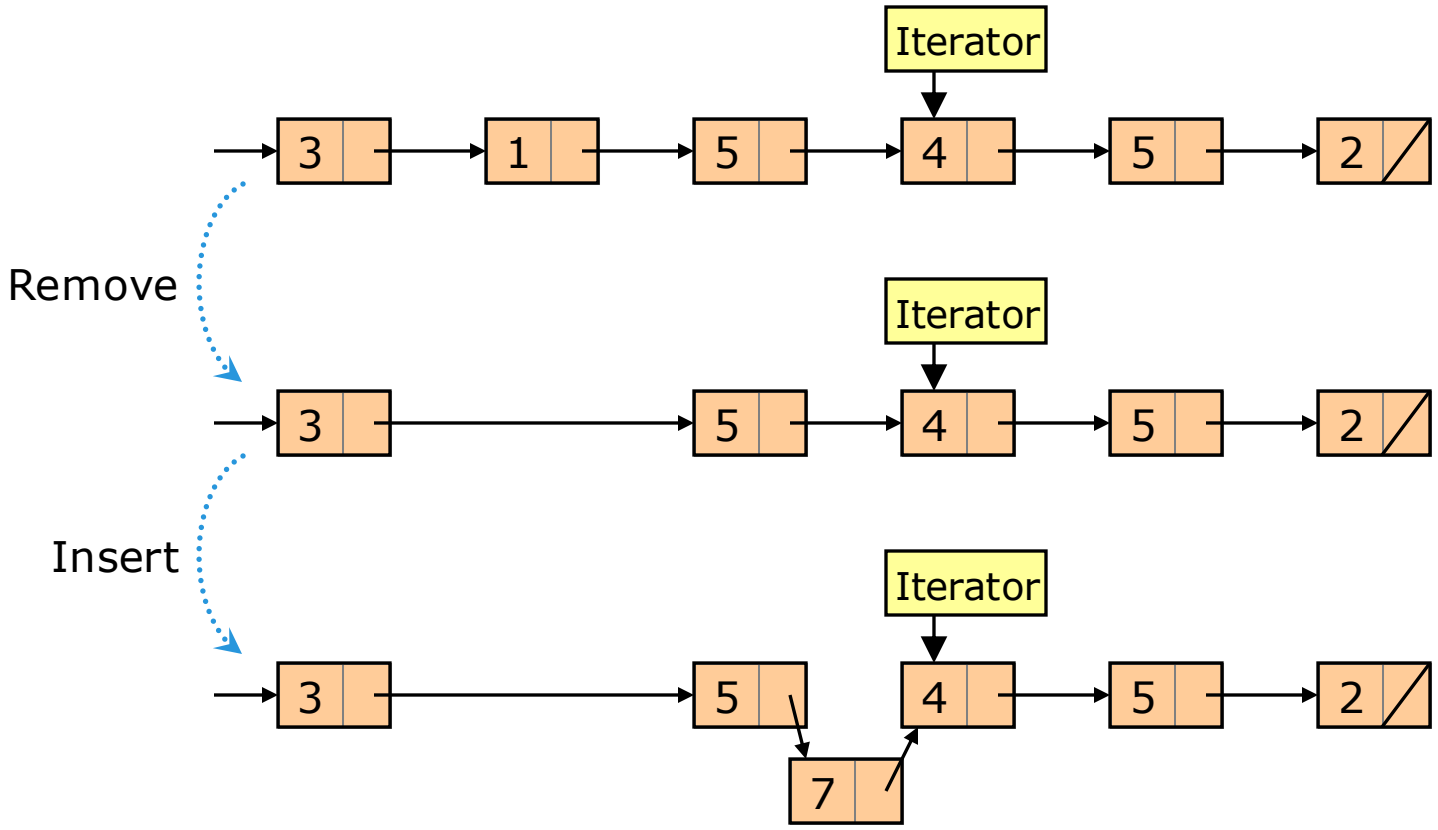
With Linked Lists, we can also do a fast **splice**:



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 [1/3]

Find the order of each of the following.
Assume good choices are made.

	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	$O(n)$	$O(n)$

*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.

What other issues arise when comparing the two data structures?

We often **find** faster with an array.

We often **rearrange** faster with a Linked List.

More on Linked Lists
Comparison with Arrays [2/3]

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 [3/3]

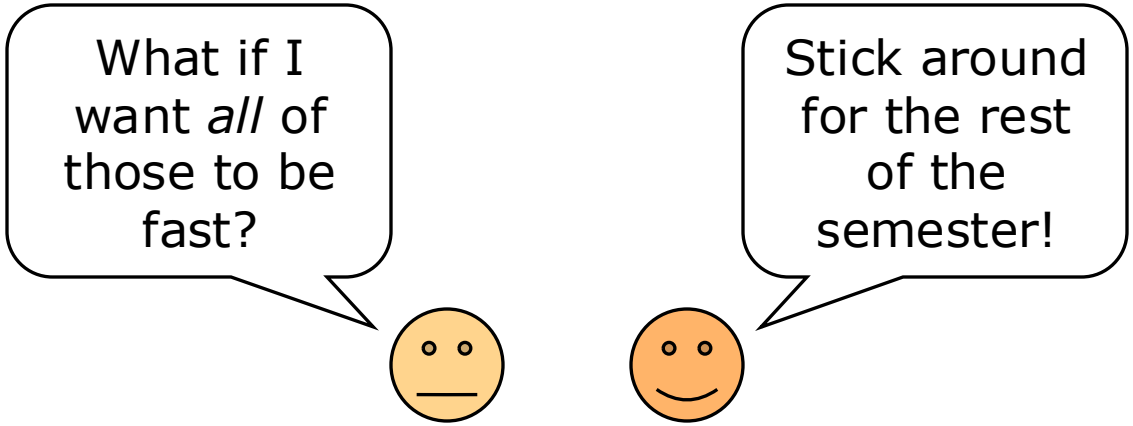
A Moral for Our Story

- 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.

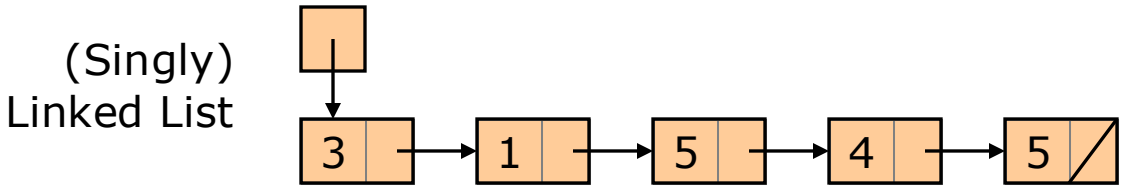
Very rough guidelines:

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

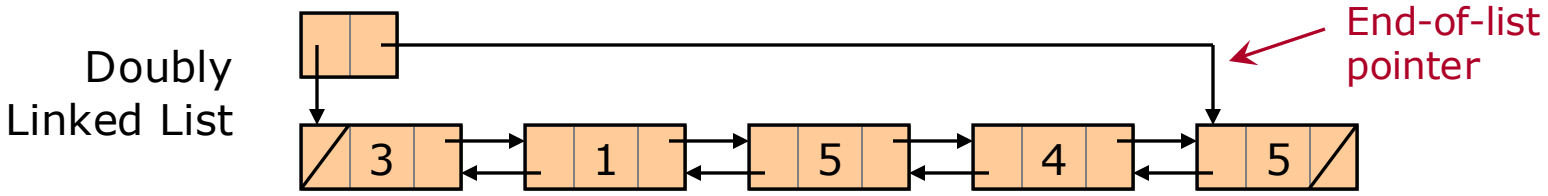


More on Linked Lists
Doubly Linked Lists [1/3]

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).



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]

	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.

We often **find** faster with an array.

We often **rearrange** faster with a Linked List.

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

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
Implementation — Two Approaches

Two Approaches to Implementing a Linked List

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

You will probably never have occasion to write the former—except perhaps as practice. But you may need to write the latter.

Q. How would these be different? In particular, what classes might we define in each case?

A1. First situation: several classes—container, node, iterator, const_iterator, *maybe others?*

A2. Second situation: a node class—and probably nothing else.

More on Linked Lists
Implementation — CODE

We have already written an internal-use-style Singly Linked List, in *Arrays & Linked Lists*.

See `llnode.hpp`.

Let's rewrite this Linked List using smart pointers.

TO DO

- Following the suggestions in *Node-Based Structures*: *Smart Pointers*, update our Linked List to include the following.
  - An owning smart pointer.
  - An insert-at-beginning operation.
  - A remove-at-beginning operation.
  - Exception-safety information.

Done. See `llnode2.hpp`. *See* `use_list2.cpp` *for a program that uses this Linked List.*

More on Linked Lists
TO BE CONTINUED …

More on Linked Lists will be continued next time.