CSCI 136:
Data Structures
and
Advanced Programming

Lecture 11

Linked lists

Instructor: Dan Barowy

Williams

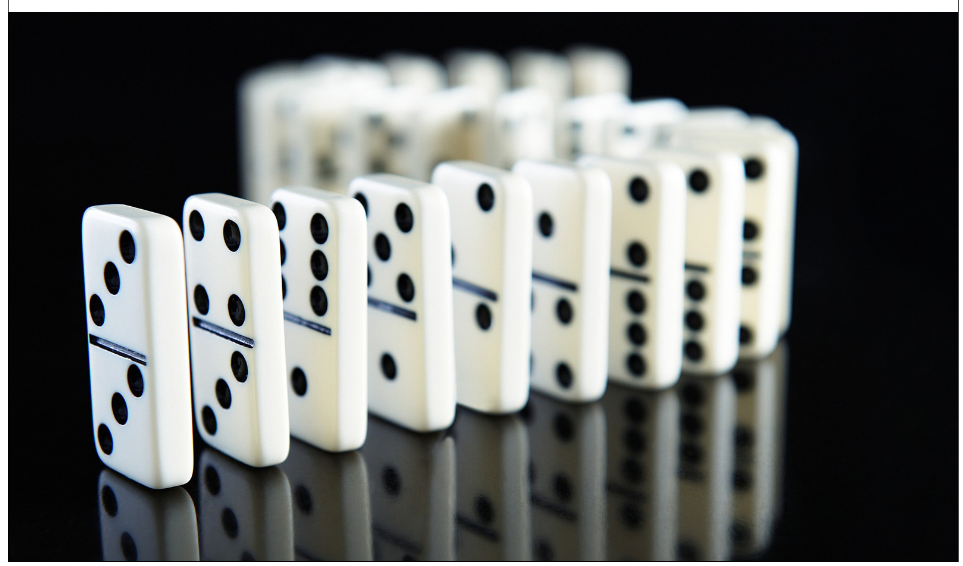
Topics

- Mathematical induction
  Vectors—why `add` is "always" O(1)
- Linked lists

Your to-dos

1. Read **before Fri**: Bailey, Ch 3.4–3.5.
2. Lab 4 (partner lab), **due Tuesday 3/9 by 10pm**.

Mathematical Induction



Principle of Mathematical Induction

Let **P(n)** be a **predicate** that is defined for **integers n**, and let **a** be a **fixed integer**.

If the following two statements are **true**:

1. **P(a)** is **true**.
2. For all integers **k ≥ a**, **if P(k)** is **true then P(k + 1)** is **true**.

then the statement

for all integers **n ≥ a**, **P(n)** is **true**

is **also true**.

To be clear:

If you want to prove that **P(n)** is **true** for all integers **n ≥ a,**

1. You must first prove that **P(a)** is **true**.

2. Then **suppose P(k)** is **true** and prove that **P(k+1)** is **true**.

Names for things and “form”

Hypothesis: **P(n)** is **true** for all integers **n ≥ a,**

1. Base case: **P(a)** is **true**.

2. Inductive step:

For all integers **k ≥ a**, **if P(k)** is **true then P(k+1)** is **true**.

Like recursion, there is an analogy

Example

Prove that the sum of the first n integers is:

$$\frac{n(n+1)}{2}$$

P(n) : $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$

Example: step 1

Step 1: Prove **P(a)**

$$P(a) : 1 = \frac{1(1+1)}{2}$$

Is this statement true? **Yes.**

Proof: $\frac{1(1+1)}{2} = \frac{2}{2} = 1$

Example: step 2

Step 2: Prove **P(k) ⇒ P(k+1)**

Assume the following is true:

P(k) : $1 + 2 + 3 + \dots + k = \frac{k(k+1)}{2}$

Prove:

P(k+1) : $1 + 2 + 3 + \dots + (k + 1) = \frac{(k+1)((k+1)+1)}{2}$

Example: step 2, left side

Step 2: Prove **P(k) ⇒ P(k+1)**

$$(1 + 2 + 3 + \dots + k) + (k + 1)$$

According to P(k), which is true,
it must be equal to:

$$(1 + 2 + 3 + \dots + k) + (k + 1) = \frac{k(k+1)}{2} + (k + 1)$$

Example: step 2, left side

Step 2: Prove **P(k) ⇒ P(k+1)**

Simplify
$$= \frac{k(k+1)}{2} + (k + 1)$$
$$= \frac{k(k+1)}{2} + \frac{2(k+1)}{2}$$
$$= \frac{k(k+1) + 2(k+1)}{2}$$

Let's stop here.
The left side is
$$= \frac{(k+1)(k+2)}{2}$$

Example: step 2, right side

Step 2: Prove **P(k) ⇒ P(k+1)**

P(k+1) : $1 + 2 + 3 + \ldots + (k + 1) = \frac{(k+1)((k+1)+1)}{2}$

Let's handle the right side now.

$$\frac{(k+1)((k+1)+1)}{2}$$

Simplify

$$\frac{(k+1)(k+2)}{2}$$

Let's stop here.

Example: step 2, conclusion

Step 2: Prove **P(k) ⇒ P(k+1)**

P(k+1) : $1 + 2 + 3 + \ldots + (k + 1) = \frac{(k+1)((k+1)+1)}{2}$

We just showed that the left side

$$\frac{(k+1)(k+2)}{2}$$

equals the right side

$$\frac{(k+1)(k+2)}{2}$$

Example: done

Step 1: Prove **P(a)** ✓

Step 2: Prove **P(k) ⇒ P(k+1)** ✓

Therefore,

P(n) : $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

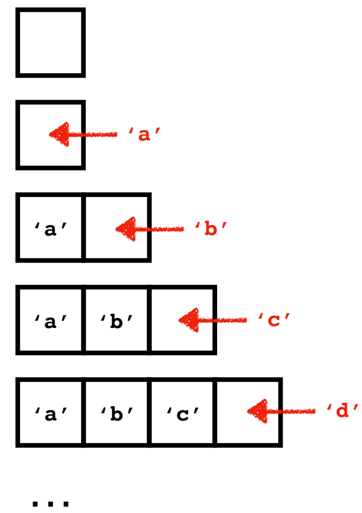
For all n ≥ 1.

Is **true**. ✓

Expanding vectors: why double?

Why is the **array doubling** strategy for Vector **better** than expanding the array **one element at a time**?

One-at-a-time expansion



Initial array.

Insert element.

New array; **copy** previous; **insert** element.

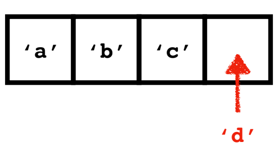
New array; **copy** previous; **insert** element.

New array; **copy** previous; **insert** element.

...

Insertion into an array

How much does **array insertion** cost?



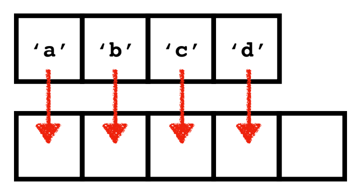
It costs **O(1)**.

In fact, lookup and insertion both cost **O(1)**.

Tradeoff: arrays are fixed size.

Copying an array

How much does an **array copy** cost?



It costs **O(1)** × **m**, where **m** is the size of the original array.

≈ **O(m)**

How many copies?

of copies for one-at-a-time expansion:

	1	**+**	**2**	**+**	**3**	**+**	**…**	**+(n-1)**
`add()`	2nd elem.		3rd elem.		4th elem.			nth elem.

Recall theorem: $1 + 2 + 3 + \ldots + k = k(k+1)/2$

Sub n-1 for k: $(n-1)((n-1)+1)/2 = n(n-1)/2$

$= (n^2-n)/2$

One-at-a-time expansion costs ≈ **$O(n^2)$**

How many copies?

of copies for doubling expansion:

	1	**+**	**2**	**+**	**4**	**+**	**…**	**+(n/2)**
`add()`	up to 2nd elem.		up to 4th elem.		up to 8th elem.			up to nth elem.

Neat theorem: $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k-1$

Suppose $n = 2^k$.

Then $1 + \ldots + n/2 = 1 + \ldots + 2^k/2$

$= 1 + \ldots + 2^{k-1} = 2^k-1 = n-1$

Doubling expansion costs ≈ **$O(n)$**

Which is faster?

💩 One-at-a-time expansion costs ≈ **$O(n^2)$** 💩

😎 Doubling expansion costs ≈ **$O(n)$** 😎

Doubling is Vin Diesel-approved.

A good practice induction problem

Prove: n cents can be obtained by using only 3-cent and 8-cent coins, for all n ≥ 15.

Linked Lists

Linked List

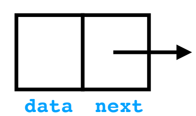
A **linked list** is a recursive data structure. A linked list is composed of simple pieces called **list nodes**. A list node contains **data** (of generic type **T**) and a **reference** (a "link") to either **another list node** or **null**.

Linked List

$\varnothing$

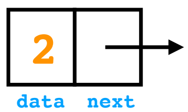
The empty list is defined as **null**.

Linked List



Every other list has at least one list node.

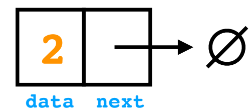
Linked List



A list node stores data of type `T`.

Here, `T` is `Integer`.

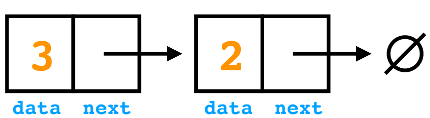
Linked List



The `next` field stores a reference (“link”) to the next node.

If the node is the last node, the next node is **null**.

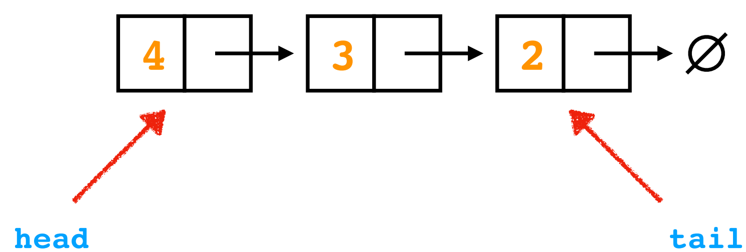
Linked List



If the next node is not **null**, it is, recursively, a list node.

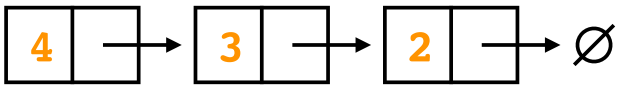
The last node in the list must always point to **null**.

Linked List



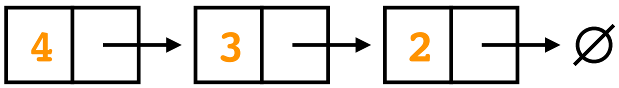
A list has parts.

Linked List



When we add data to a list, we always `append` to the `head`.

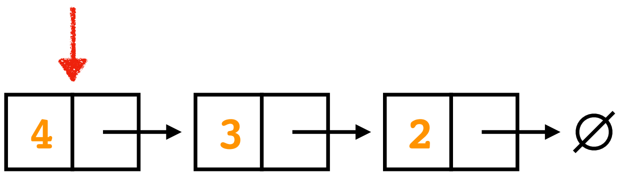
Linked List



To find a value, we must always traverse the list starting from the `head`.

E.g., looking for **2**…

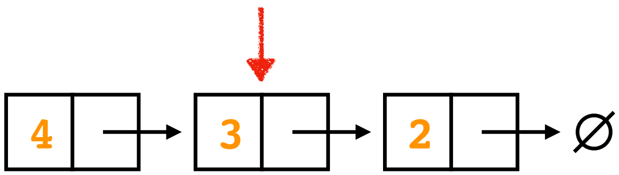
Linked List



To find a value, we must always traverse the list starting from the `head`.

E.g., looking for **2**…

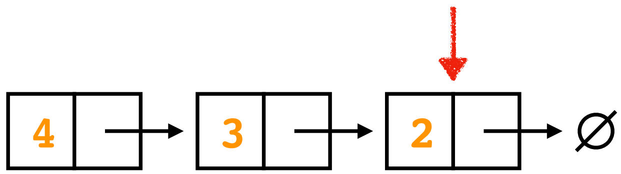
Linked List



To find a value, we must always traverse the list starting from the `head`.

E.g., looking for **2**…

Linked List



To find a value, we must always traverse the list starting from the `head`.

E.g., looking for **2**…

Example code

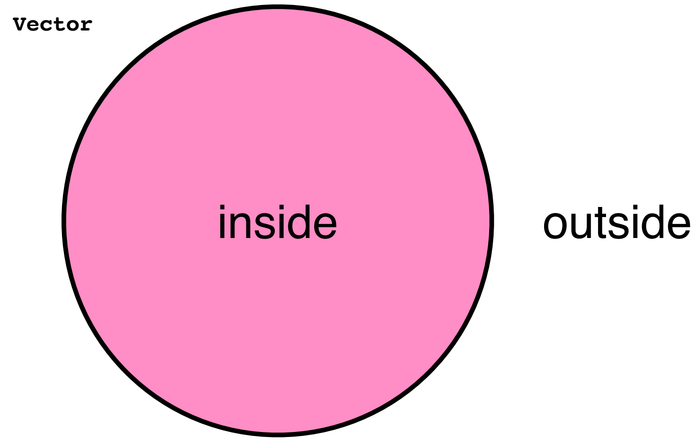
The purpose of a class:

To “abstract away” implementation details.

Abstraction

Abstraction is the process of **removing irrelevant information** so that a program is easier to understand.

Think of a class as having two sides.



Design so user never needs to "look inside".

Think of a class as having two sides.

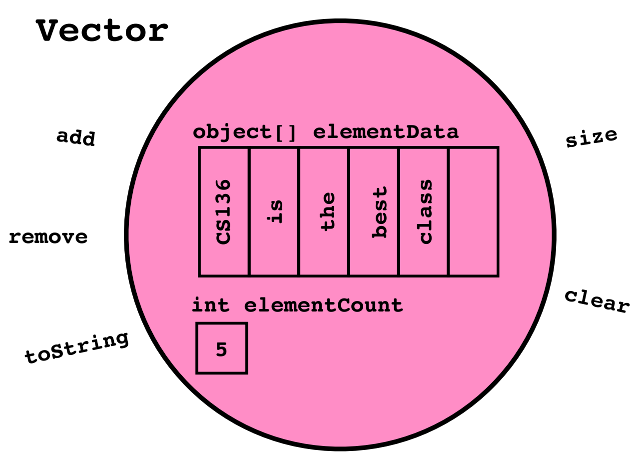
The outside: A class should represent **one idea**, and the class's methods should support working with that one idea.

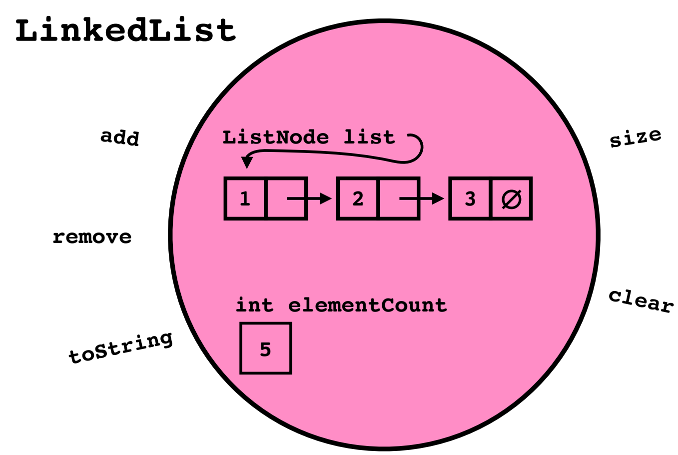
E.g., `Vector`: Represents an arbitrarily long sequence of elements. Ideally, it also has the same asymptotic properties as an array.

You can:

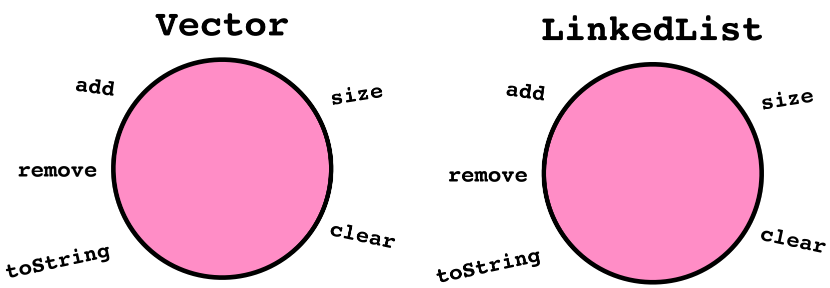
- `add` to it
- `remove` from it
- ask it for its `size`…
- convert it `toString`
- etc.

The **user** of a class **should not need to know how** a class works.





Do you see any similarities?



The two classes share the same **interface**.

Interface

An **interface** defines boundary between two systems across which they share information. An interface is a **contract**: calling a method defined in an interface returns the data as promised.

Because an interface **contains no implementation**, programmers who use them **cannot rely on implementation details**.

E.g., the **`List`** interface states that there must be an **`add`** method but does not say how it should be implemented.

`structure5` List implementations

In structure5, the following classes are all a kind of List:

```
Vector
SinglyLinkedList
DoublyLinkedList
CircularList
```

So what is a `List` exactly?

Recap & Next Class

Today:

- Why Vector should double
- Lists

Next class:

- ADTs
- More lists