ANNOUNCEMENTS
today is **No Laptop Monday**!

Week08a

WARMUP
"a chain is only as strong as its weakest link"
- what does this expression mean?
- what is a chain?
- what is a link?
- is this true for real-world metal chains? why or why not?

TODAY linked lists

linked lists

record lecture

(p)review: list interface

list interface

```
// Get the element with this index.
ElementType get(int index);

// Add (append) an element to the back of the list.
void add();

// Add (insert) an element into the list so it has this index
void add(int index, ElementType element);

// Remove (delete) the element in the list at this index.
void remove(int index);

// Get the number of elements currently in the list.
int size();
```

list interface (cont.)

```
// NOTE: Many other functions could be included
//       in this interface.
void sort(); // Sort the list.
void reverse(); // Reverse the list.

List<ElementType> sorted(); // Get sorted copy of the list.
List<ElementType> reversed(); // Get reversed copy of list.

// Get index of first element with this value.
int find(ElementType element);

...
```

a few weeks ago,
we implemented the list interface
using an array

the *array list*

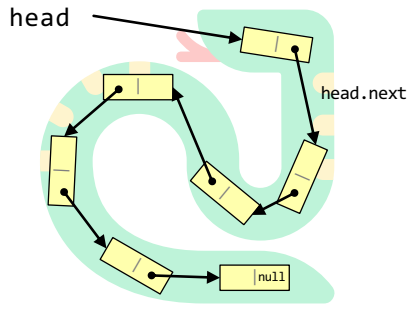
this week, we will implement
the list interface using nodes with
"links" (references) to other nodes

this will be called a **linked list**

note

today we will be discussing the
simplest possible linked list

(LinkedList literally just has a
reference to Node head.)



some other implementations are possible.
some will be faster than this one.

for linked lists, do NOT memorize
big O runtimes out of context

why are we doing this?

A: it will be cool to see two very different implementations of the same interface 🧦

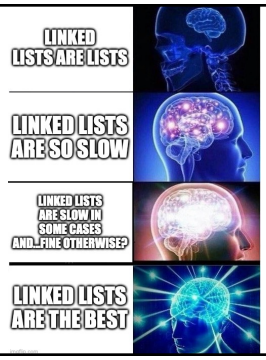
B: linked lists will prepare us for trees and graphs 🌳

C: linked lists are incredibly FUNdaMENTAL 🥳

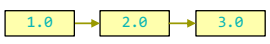
(for us, as fundamental as arrays)

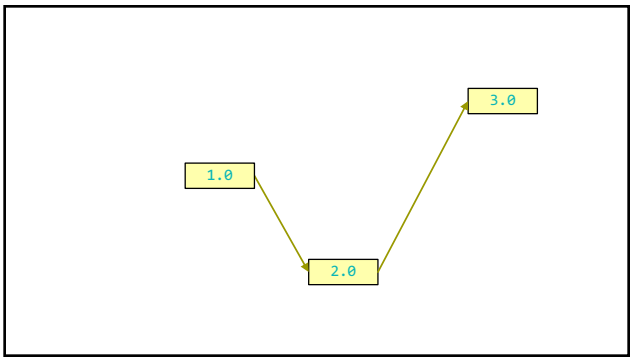
D: linked lists are actually big O better (than array lists) in very specific cases

E: linked lists are actually really, really important
(especially in the C programming language)



linked list





linked list

```
class LinkedList {
  Node head;
}
```

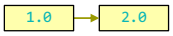
```
class Node {
  Value value;
  Node next;

  Node(Value value) {
    this.value = value;
  }
}
```

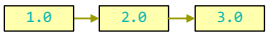
```
LinkedList list = new LinkedList();
```

```
list.head = new Node(1.0);
```


```
list.head.next = new Node(2.0);
```




```
list.head.next.next = new Node(3.0);
```



linked list

```
class LinkedList {
  Node head;

  void add(Value value) {
    ...
  }
}
```

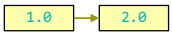
```
class Node {
  Value value;
  Node next;

  Node(Value value) {
```

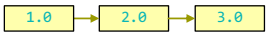
```
LinkedList list = new LinkedList();
```

```
list.add(1.0);
```


```
list.add(2.0);
```

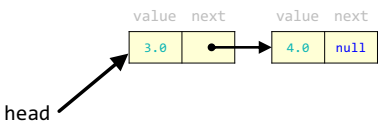



```
list.add(3.0);
```

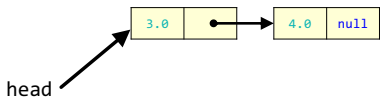


example: `addFront(value)`

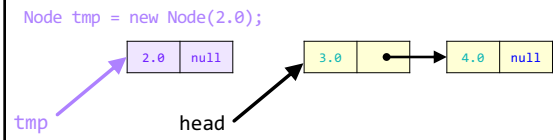
addFront(2.0)



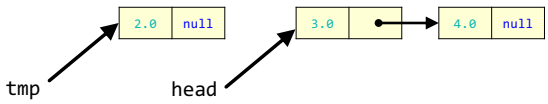
addFront(2.0)



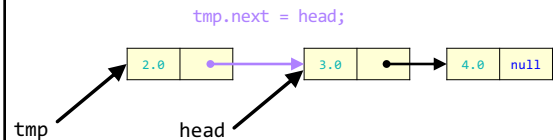
addFront(2.0)



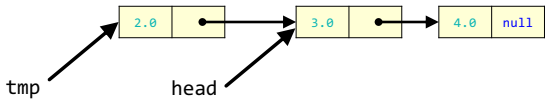
addFront(2.0)



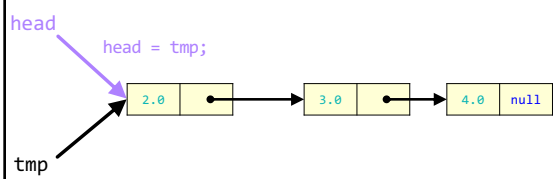
addFront(2.0)



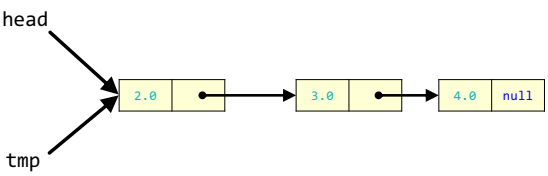
addFront(2.0)



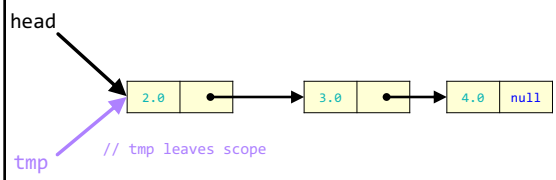
addFront(2.0)



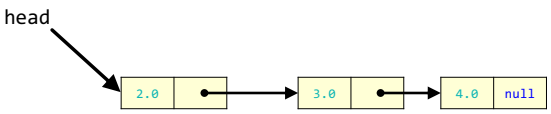
addFront(2.0)



addFront(2.0)

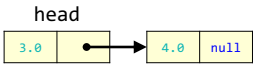


addFront(2.0)



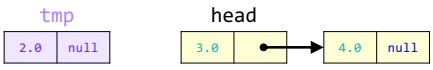
same thing but with labels instead of arrows for `head` and `tmp`

addFront(2.0)

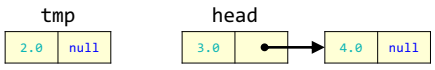


addFront(2.0)

```
Node tmp = new Node(2.0);
```

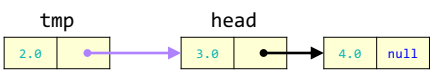


addFront(2.0)

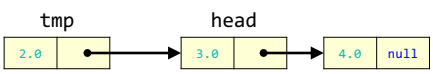


addFront(2.0)

tmp.next = head;

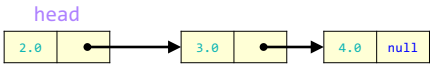


addFront(2.0)

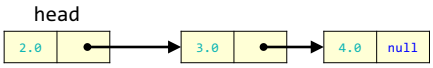


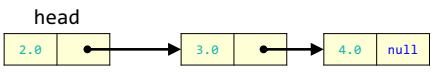
addFront(2.0)

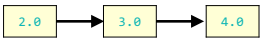
head = tmp;



addFront(2.0)



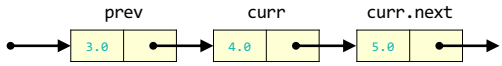




(even) more abstract diagram

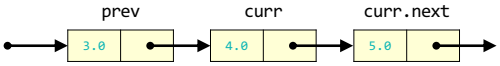
example: remove

remove

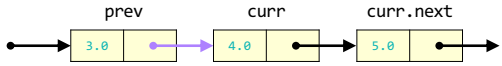



```
Node prev = null; // the previous node
Node curr = head; // the current node
while (...) {
    ...
    prev = curr;
    curr = curr.next;
}
```

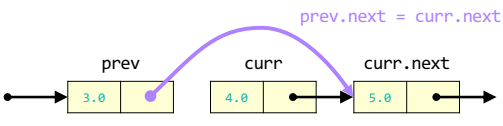
remove



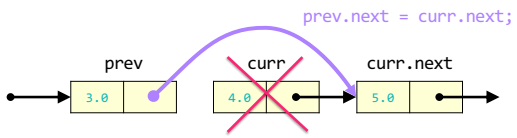
remove



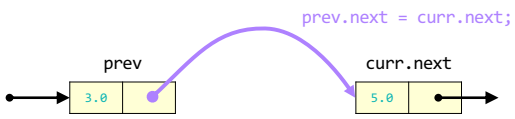
remove



remove

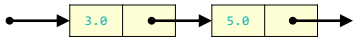


remove



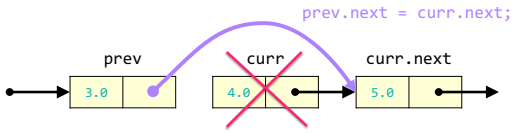
remove

remove



where did the middle node go?

- way back when...we created it using `new`
- then we just like..."stopped referring to it"
- is it gone now? 😵



🚚 it has been garbage collected

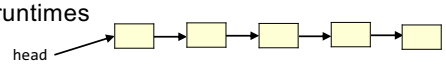
[board discussion of
"no directed path from stack to the node"]

big O runtimes

what is the big O runtime of size()?
[pointing activity]

O(n) 😢

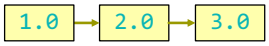
singly-linked list
worst case runtimes



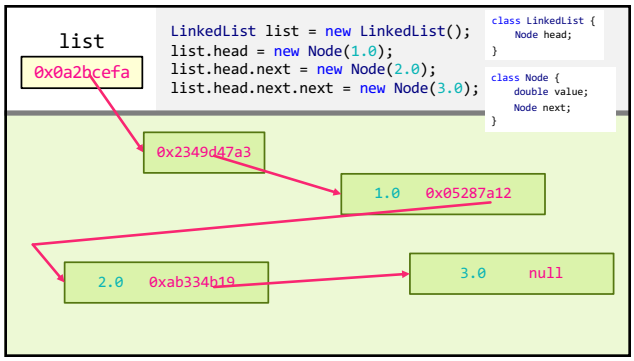
- `list.add(index, value)`
  - `// O(n)`
- `list.removeByIndex(index)`
  - `// O(n)`
- `list.size()`
  - `// O(n)`

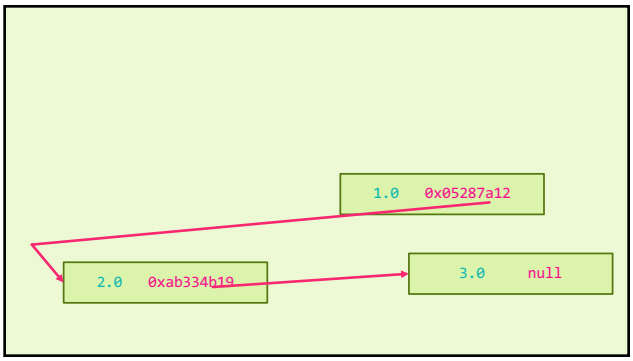
- `list.addFront() // list.add(0, value)`
  - `// O(1)`
- `list.removeFront() // list.removeByIndex(0)`
  - `// O(1)`
- `list.addBack()`
  - `// O(n)`
- `list.removeBack()`
  - `// O(n)`

beyond big O runtime



what does this actually look like in memory?





what does this *mean*?

cons? 😕

pros? 😳
(how is this very different than an array list?)

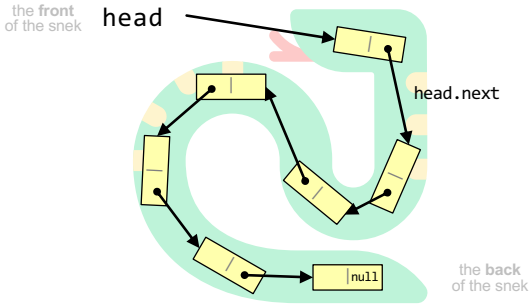
https://x.com/_kzr/status/1672497446705037312

ANNOUNCEMENTS
friday's colloquium will be cool!
my friend Pascal makes ✨**underwater robotic spheres**✨!
midterms will be returned Fri

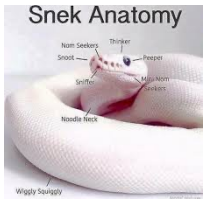
Week08b

WARMUP
review: for a linked list, `size()` is O(n) runtime
- how could you make `size()` (or perhaps... *`size`*) O(1)?
-- is this at all...*spooky*? 👻

TODAY `size()` and riffs on linked lists



linked lisssssst

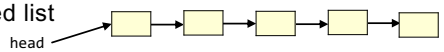


[record lecture]

LinkedList review

runtimes

worst case singly-linked list runtimes

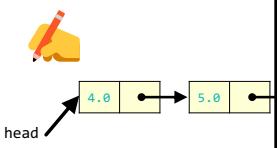


- `list.add(index, value)`
  - `// O(n)`
- `list.removeByIndex(index)`
  - `// O(n)`
- `list.size()`
  - `// O(n)`
- `list.addFront()`
  - `// O(1)`
- `list.removeFront()`
  - `// O(1)`
- `list.addBack()`
  - `// O(n)`
- `list.removeBack()`
  - `// O(n)`

additional warmup: prepending a list

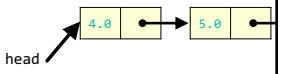
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



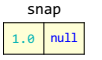
example

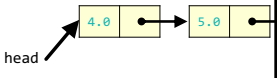
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



example

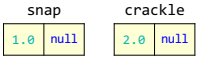
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```

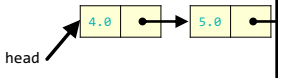




example

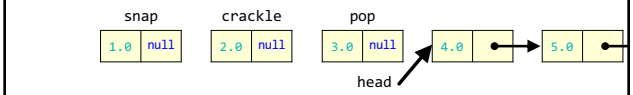
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```





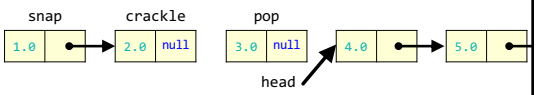
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



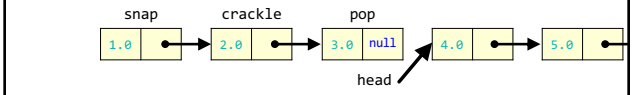
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



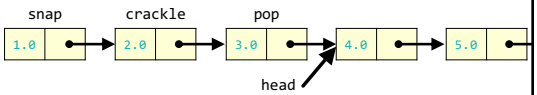
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



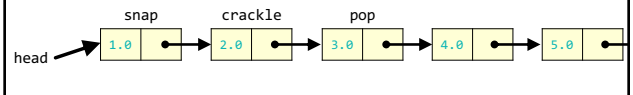
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



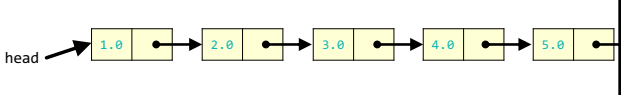
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



🧠 was that faster than calling `addFront()` n times?

🧠 same big O or different big O?
(if we were to call it many times)

🧠 was that faster than calling `addFront()` n times?
yes
(only updated head once)

🧠 same big O or different big O?
same
(still have to "hook up" O(n) references)

[implement LinkedList]

[implement LinkedList]

`size()`

[implement `size()`]

```
static class LinkedList {
    Node head;

    int size() {
        int result = 0;
        Node curr = head;
        while (curr != null) {
            ++result;
            curr = curr.next;
        }
        return result;
    }
}
```

size()

- what is the big O runtime of this method?
  - O(n) 😩
- this seems like a pretty steep cost to pay just to know the list's size... what would be a more efficient approach?
  - store size as an instance variable
    - update it every time you change the number of elements in the list (inside of `add`, `remove`, etc.)
  - what is the runtime of this approach?
    - O(1) 😀

while way more efficient, this approach is perhaps a bit spooky 👻

multiple functions are now also responsible for carefully modifying an instance variable
(mess up, and any that depends on size will be very weirdly broken)

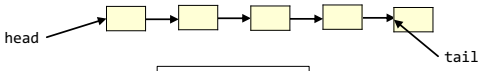
note: the A homework doesn't use size at all

but if you *were* going to implement/use the list's size...
i would start with size() as a function,
get everything working perfectly,
and only then carefully turn it into an instance variable
😳👍

(it's this thing again)

tail reference

singly-linked list with reference to tail

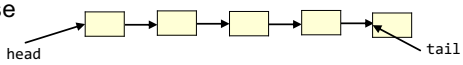



```
class LinkedList {
    Node head;
    Node tail;
}
```

```
class Node {
    double value;
    Node next;
}
```

singly-linked list with reference to tail

worst case runtimes



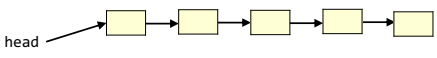
- `list.add(index, value)`
  - `// O(n)`
- `list.removeByIndex(index)`
  - `// O(n)`
- `list.size()`
  - `// O(n)`

- `list.addFront()`
  - `// O(1)`
- `list.removeFront()`
  - `// O(1)`
- `list.addBack()`
  - **`// O(1)`** 😀
- `list.removeBack()`
  - `// O(n)` 😡

doubly-linked list

singly- vs. doubly-linked

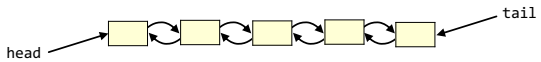
singly-linked list




```
class LinkedList {
    Node head;
}
```

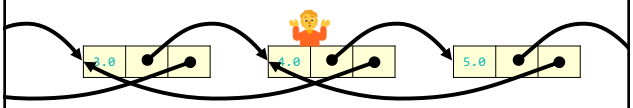
```
class Node {
    double value;
    Node next;
}
```

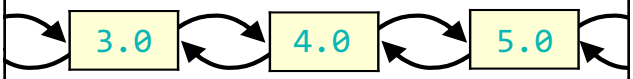
doubly-linked list




```
class LinkedList2 {
    Node2 head;
    Node2 tail;
}
```

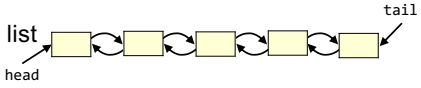
```
class Node2 {
    double value;
    Node2 next;
    Node2 prev;
}
```





runtimes

worst case doubly-linked list runtimes



- `list.add(index, value)`
  - `// O(n)`
- `list.removeByIndex(index)`
  - `// O(n)`
- `list.size()`
  - `// O(n)`
- `list.addFront()`
  - `// O(1)`
- `list.removeFront()`
  - `// O(1)`
- `list.addBack()`
  - **`// O(1)`** 😀
- `list.removeBack()`
  - **`// O(1)`** 😀

a doubly-linked list is a great way to implement a **deque** (**double-ended queue**)

- `O(1) addFront()`
  `O(1) removeFront()`
  `O(1) addBack()`
  `O(1) removeBack()`

- 🧠 could you pull this off with an **array list**?
  - no.
    - addFront() is O(n)

- 🧠 could you pull this off with an array?
  - sort of!—the **array deque** (amortized O(1) add)

something
fun?