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

1

2

3

linked lists

record lecture

4

(p)review: list interface

5

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();
```

6

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);

  - ...
```

7

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

the *array list*

8

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

this will be called a **linked list**

9

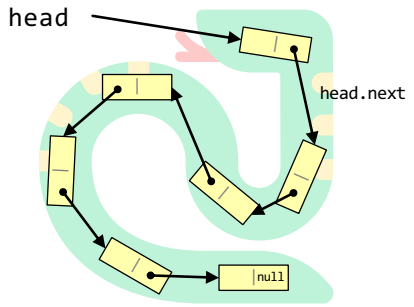
note

10

today we will be discussing the
simplest possible linked list

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

11



12

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

13

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

14

why are we doing this?

15

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

16

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

17

C: linked lists are incredibly
FUNdaMENTAL 🥳

(for us, as fundamental as arrays)

18

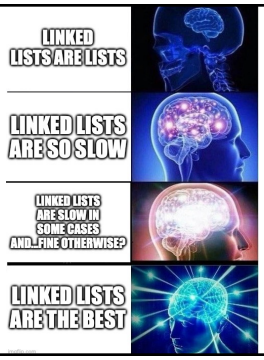
D: linked lists are actually big O better (than array lists) in very specific cases

19

E: linked lists are actually really, really important

(especially in the C programming language)

20

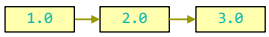


21

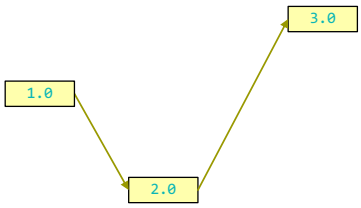
linked list

22

23



24



25

linked list

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

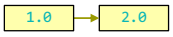
```
class Node {
  Value value;
  Node next;

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

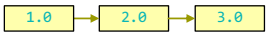
```
LinkedList list = new LinkedList();
```

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


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




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



26

linked list

```
class LinkedList {
  Node head;

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

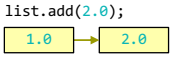
```
class Node {
  Value value;
  Node next;

  Node(Value value) {
```

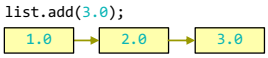
```
LinkedList list = new LinkedList();
```

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


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




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

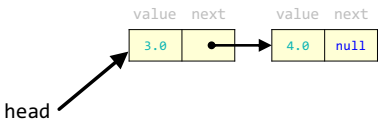


27

example: `addFront(value)`

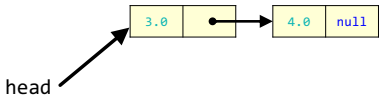
28

`addFront(2.0)`

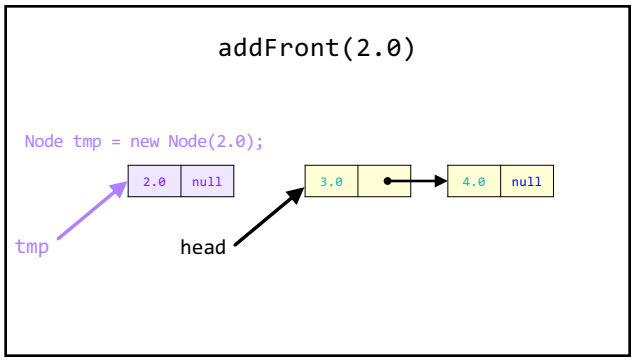


29

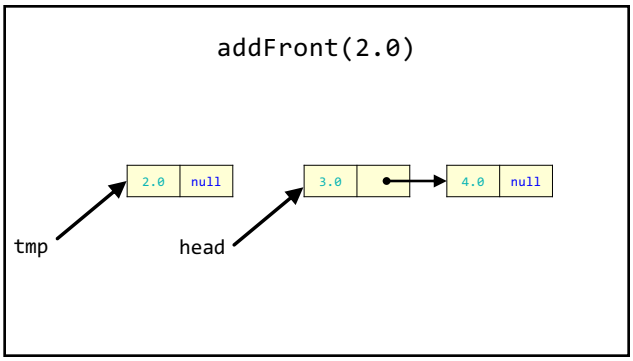
`addFront(2.0)`



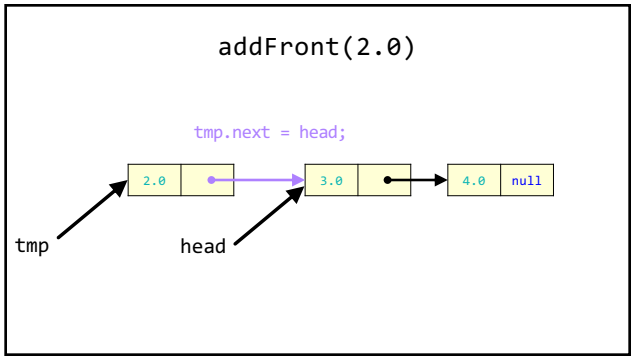
30



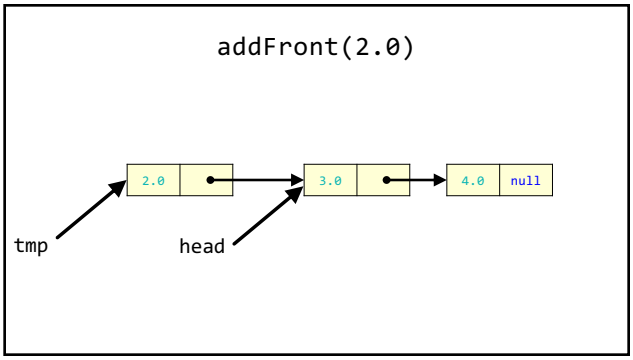
31



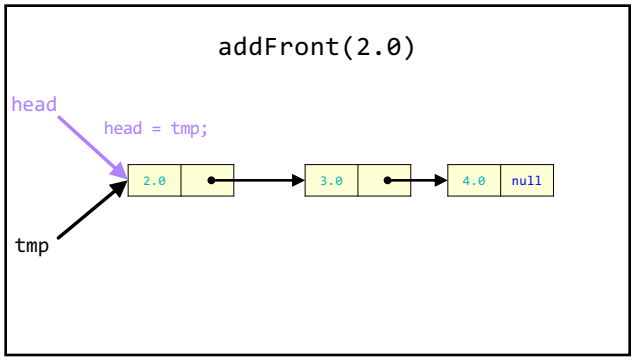
32



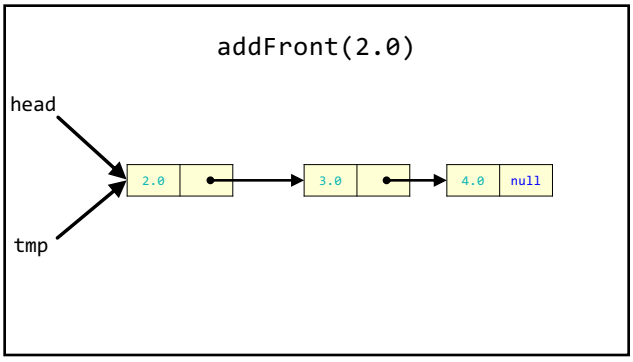
33



34

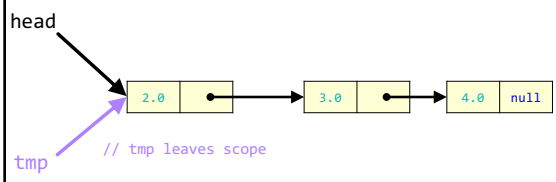


35



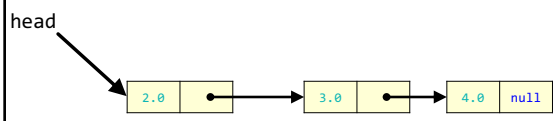
36

addFront(2.0)



37

addFront(2.0)

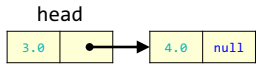


38

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

39

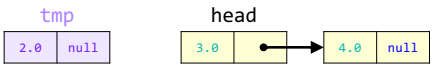
addFront(2.0)



40

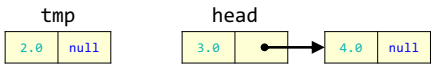
addFront(2.0)

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



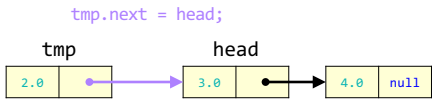
41

addFront(2.0)



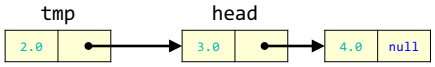
42

addFront(2.0)



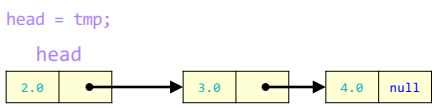
43

addFront(2.0)



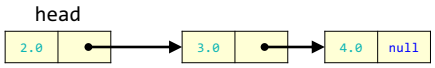
44

addFront(2.0)

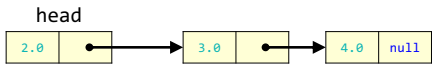


45

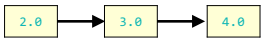
addFront(2.0)



46



47



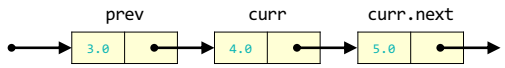
(even) more abstract diagram

48

example: remove

49

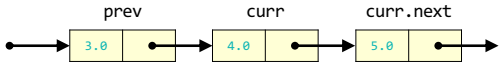
remove




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

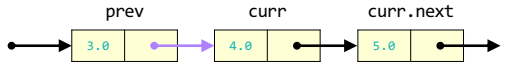
50

remove



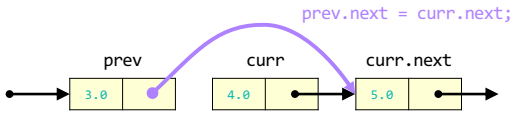
51

remove



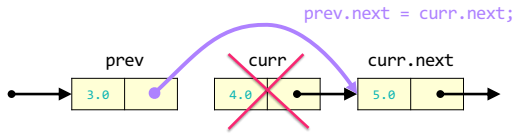
52

remove



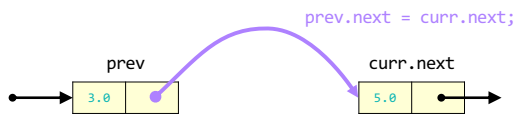
53

remove



54

remove

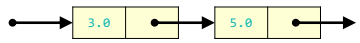


55

remove

56

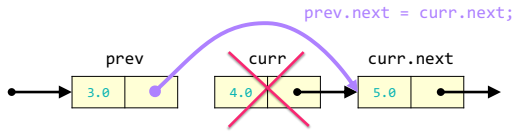
remove



57

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? 😵‍💫



58

🚚 it has been garbage collected

59

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

60

big O runtimes

61

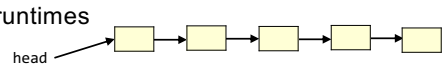
what is the big O runtime of size()?
[pointing activity]

62

O(n) 😥

63

singly-linked list
worst case runtimes



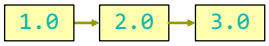
- list.add(index, value)
  - // O(n)
- list.removeByIndex(index)
  - // O(n)
- list.size()
  - // O(n)

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

64

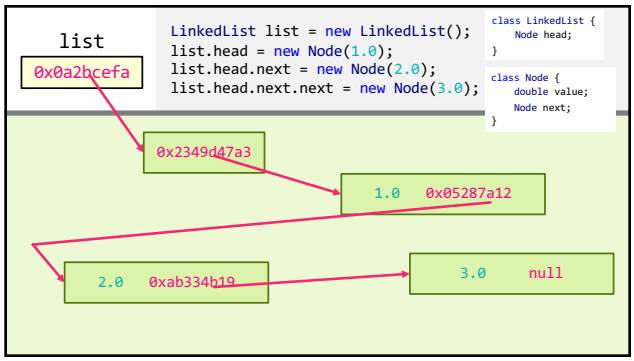
beyond big O runtime

65

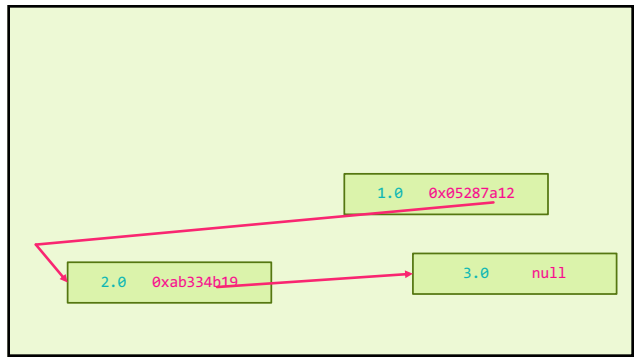


what does this actually look like in memory?

66



67



68

what does this *mean*?

cons? ☹️

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

69

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

70

72

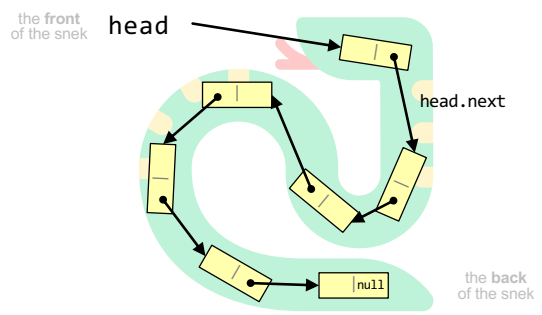
ANNOUNCEMENTS
friday's colloquium will be cool!
my friend Pascal makes ✨**underwater robotic spheres**✨!
midterms will be returned Fri

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

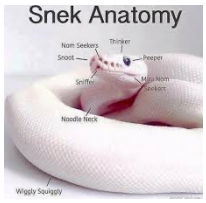
73



74

linked lisssssst

75



76

[record lecture]

77

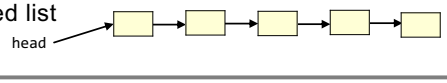
LinkedList review

78

runtimes

79

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)`

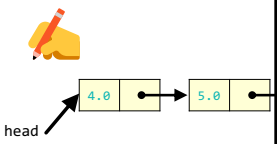
80

additional warmup: prepending a list

81

example

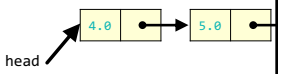
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



82

example

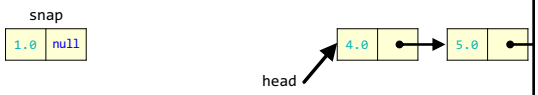
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



83

example

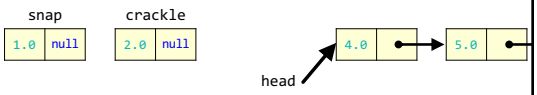
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



84

example

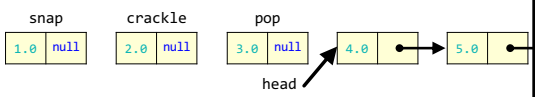
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



85

example

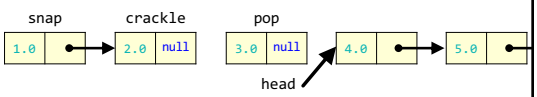
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



86

example

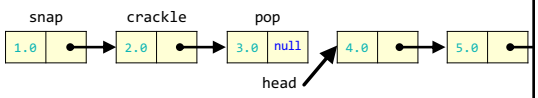
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



87

example

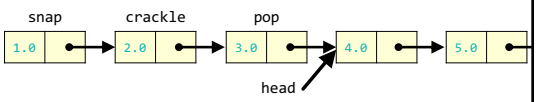
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



88

example

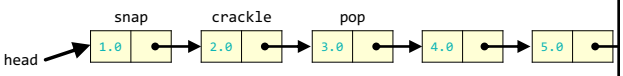
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



89

example

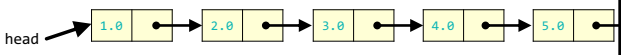
```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



90

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;
```



91

```
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;
```

vs.

```
addFront(3.0);
addFront(2.0);
addFront(1.0);
```

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

🧠 same big O or different big O?

(if we were to call it many times)

92

🧠 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)

93

[implement LinkedList]

94

[implement LinkedList]

95

`size()`

96

[implement `size()` poorly]

97

[implement `size()` poorly]

98

joyful implementation of size

99

```
static class LinkedList {
    Node head;

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

100

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) 😄

101

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 code that depends on size will be very weirdly broken)

102

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

103

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
😳 👍

104

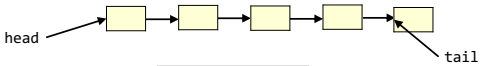
(it's this thing again)

105

tail reference

106

singly-linked list with reference to tail

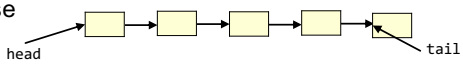



```
class LinkedList {
    Node head;
    Node tail;
}
```

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

107

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) 😡

108

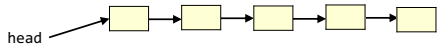
doubly-linked list

109

singly- vs. doubly-linked

110

singly-linked list

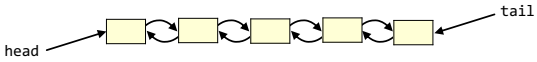



```
class LinkedList {
    Node head;
}
```

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

111

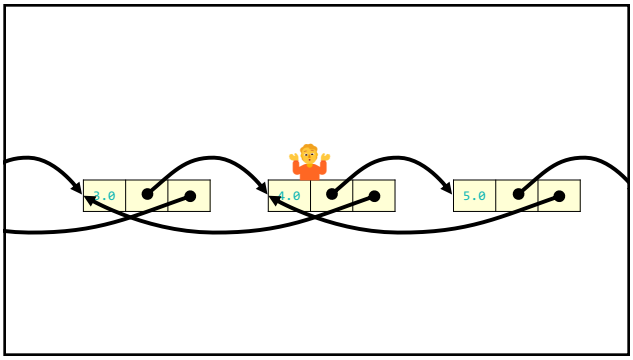
doubly-linked list



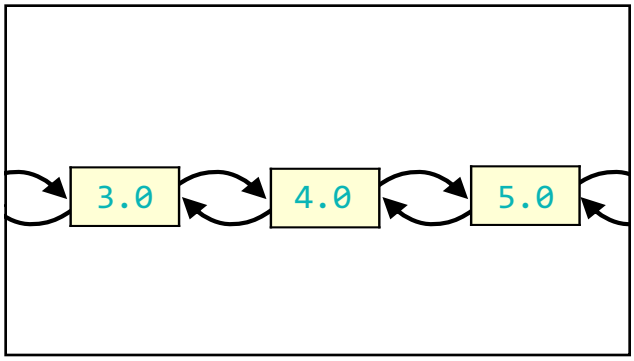

```
class LinkedList2 {
    Node2 head;
    Node2 tail;
}
```

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

112



113

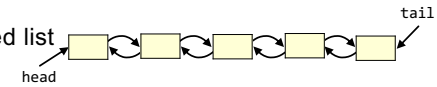


114

runtimes

115

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)`** 😀

116

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)

117

something fun?

118