Linked list visualized

[1, 8, 4, 7]

head
  data: 1
  next: data: 8
  next: data: 4
  next: data: 7
  next: null

Adding to the end

[1, 8, 4, 7]

head
  data: 1
  next: data: 8
  next: data: 4
  next: data: 7
  next: null

How can we add a value to the end of the list?

Adding to the end: add 9 to the end

[1, 8, 4, 7]

head
  data: 1
  next: data: 8
  next: data: 4
  next: data: 7
  next: null

find the end of the list

How do we know we're at the end?

Adding to the end: add 9 to the end

[1, 8, 4, 7]

head
  data: 1
  next: data: 8
  next: data: 4
  next: data: 7
  next: null

find the end of the list

finger.next() == null
Adding to the end: add 9 to the end

[1, 8, 4, 7]

create a new node and add it to the end

5

Adding to the end: add 9 to the end

[1, 8, 4, 7]

create a new node and add it to the end

6

Any special cases we need to handle?

7

7

Adding to the end: add 9 to the end

[1, 8, 4, 7]

Node finger = head;
    while (finger.next() != null) {
        finger = finger.next();
    }
    finger.setNext(new Node(9));

8

Adding to the end

[1, 8, 4, 7]

Node finger = head;
    while (finger.next() != null) {
        finger = finger.next();
    }
    finger.setNext(new Node(9));

8
Adding to the end

```
[1, 8, 4, 7]

head: null
```

What would happen here?

```
Node finger = head;
while (finger.next() != null) {
    finger = finger.next();
}
finger.setNext(new Node(value));
```

Removing a value

```
[1, 8, 4, 7]
```

```
public void addLast(E value) {
    if (head == null) {
        head = new Node(value);
    } else {
        Node finger = head;
        while (finger.next() != null) {
            finger = finger.next();
        }
        finger.setNext(new Node(value));
    }
}
```

How can we remove a value from the list?

```
[1, 8, 4, 7]
```

Removing a value: remove 4

```
[1, 8, 7]
```

```
head
   data: 1
next: ——

   data: 8
next: ——

   data: 7
next: null
```

Find it
Removing a value: remove 4
[1, 8, 4, 7]

Need the node before it too!

splice it out

Removing a value: remove 4
[1, 8, 4, 7]

splice it out

Removing a value: remove 4
[1, 8, 4, 7]

splice it out

Removing a value: remove 4
[1, 8, 4, 7]

splice it out
Removing a value

```java
Node finger = head.next;
Node prev = head;
while (finger != null && !finger.value.equals(value)) {
    prev = finger;
    finger = finger.next;
}
if (finger != null) {
    prev.setNext(finger.next);
}
```
Removing a value

Node finger = head.next;
Node prev = head;
while (finger != null && !finger.value.equals(value)) {
    prev = finger;
    finger = finger.next;
}
if (finger != null) {
    prev.setNext(finger.next);
} else {
    When would finger be null?
    }

Node finger = head.next;
Node prev = head;
while (finger != null && !finger.value.equals(value)) {
    prev = finger;
    finger = finger.next;
}
if (finger != null) {
    prev.setNext(finger.next);
}
Removing a value

```
head: null

What would happen here?
```

Removing a value

```
Node finger = head.next;
Node prev = head;
while (finger != null && finger.value.equals(value)) {
  prev = finger;
  finger = finger.next;
}
if (finger == null) {
  prev.setNext(finger.next);
}
```

Any special cases we need to handle?

Removing a value

```
Node finger = head.next;
Node prev = head;
while (finger != null && finger.value.equals(value)) {
  prev = finger;
  finger = finger.next;
}
if (finger == null) {
  prev.setNext(finger.next);
}
```

Any other special cases we need to handle?

Delete 1. Does it work?
Removing a value

```java
public void remove(E value){
    if (head == null) {
        return;
    }
    if (head.value.equals(value)) {
        head = head.next;
        return;
    }
    Node finger = head.next;
    Node prev = head;
    while (finger != null && !finger.value.equals(value)) {
        prev = finger;
        finger = finger.next;
    }
    if (finger == null) {
        prev.setNext(finger.next);
    }
    return;
}
```
Linked lists: fast or slow?

- add to the end: slow
- add to the front: fast
- contains: slow
- get: slow
- insert at an index: slow
- remove an element: slow
- remove from the front: fast
- set the value of an existing element: slow
- size: fast

Can we make any of these faster?

Linked list visualized

```
[1, 8, 4, 7]
```

head

```
data: 1
next: --
data: 8
next: --
data: 4
next: --
data: 7
next: null
```

How does this help us?

- We don't have to search for the end of the list!

```javascript
head

```
data: 1
next: 
data: 8
next: 
data: 4
next: 
data: 7
next: null
```

tail

tail = tail.next();
```
Linked list visualized

```
public void add(int value) {
    if (head == null) {
        head = new Node(value);
        tail = head;
    } else {
        tail.setNext(new Node(value));
        tail = tail.next;
    }
}
```

Any downsides?

Circularly linked list visualized

```
tail.next() == head
```

Where is head?
Linked lists: fast or slow?

- add to the end: fast
- add to the front: fast
- contains: slow
- get: slow
- insert at an index: slow
- remove an element: slow
- remove from the front: fast
- set the value of an existing element: slow
- size: fast

Linked list visualized

\[1, 8, 4, 7\]

Is removing from the end fast?

Linked list visualized

\[1, 8, 4, 7\]

No! Need access to previous

Doubly linked list visualized

Keep track of both the next and previous node in the list
Yes removing from the end fast?
Doubly linked list visualized

Could I remove a node if I had a reference to it?

Doubly linked list visualized

Could I remove a node if I had a reference to it?
Doubly linked list visualized

How does this help us? Still have to find the node?

If we’re iterating over the data already
Other scenarios (see upcoming assignment G)
## List performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>Singly linked list</th>
<th>Doubly linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>add to end</td>
<td>fast</td>
<td>fast</td>
<td>fast</td>
</tr>
<tr>
<td>add to front</td>
<td>slow</td>
<td>fast</td>
<td>fast</td>
</tr>
<tr>
<td>insert at index</td>
<td>slow</td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td>contains</td>
<td>slow</td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td>get</td>
<td>fast</td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td>set</td>
<td>fast</td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td>remove</td>
<td>slow</td>
<td>slow</td>
<td>slow</td>
</tr>
<tr>
<td>remove from end</td>
<td>fast</td>
<td>slow</td>
<td>fast</td>
</tr>
<tr>
<td>remove from front</td>
<td>slow</td>
<td>fast</td>
<td>fast</td>
</tr>
<tr>
<td>size</td>
<td>fast</td>
<td>fast</td>
<td>fast</td>
</tr>
</tbody>
</table>