Implementing a Linked List

Sixtyish slides of Linked List bliss
add(int idx, String s)
Two Cases to Handle

Add at the head of the list

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

Add anywhere else

- Start at the head of the list and traverse the nodes until you’re at node at position $index - 1$, call this node $current$
- Create a new node with the specified data, call this node $new$
- Set $new.next$ to be $current.next$
- Set $current.next$ to be $new$
Adding at the head

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

l.insert(0, “NEW_HEAD”)
Adding at the head

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

```python
l.insert(0, "NEW_HEAD")
```
Adding at the head

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

```python
l.insert(0, "NEW_HEAD")
```
Adding at the head

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

```
l.insert(0, “NEW_HEAD”)```

```
HEAD

“NEW_HEAD”

“C”

“B”

“A”
```
Adding at the head

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

l.insert(0, “NEW_HEAD”)

Rearranging for style, no change actually made to the list!
Adding within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you’re at node at position index - 1, call this node current
- Create a new node with the specified data, call this node new
- Set new.next to be current.next
- Set current.next to be new

l.insert(2, “NEW_NODE”)
Adding within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you’re at node at position `index - 1`, call this node `current`
- Create a new node with the specified data, call this node `new`
- Set `new.next` to be `current.next`
- Set `current.next` to be `new`

```
l.insert(2, "NEW_NODE")
```
Adding within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you’re at node at position $index - 1$, call this node $current$
- Create a new node with the specified data, call this node $new$
- Set $new$.next to be $current$.next
- Set $current$.next to be $new$

```python
l.insert(2, "NEW_NODE")
```
Adding within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you’re at node at position index - 1, call this node current
- Create a new node with the specified data, call this node new
- Set new.next to be current.next
- Set current.next to be new

```python
l.insert(2, "NEW_NODE")
```
Adding within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you’re at node at position $index - 1$, call this node current
- Create a new node with the specified data, call this node new
- Set $new.next$ to be $current.next$
- Set $current.next$ to be new

```java
l.insert(2, "NEW_NODE")
```
Adding within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you’re at node at position $index - 1$, call this node $current$
- Create a new node with the specified data, call this node $new$
- Set $new.next$ to be $current.next$
- Set $current.next$ to be $new$

```python
l.insert(2, "NEW_NODE")
```

Rearranging for style, no change actually made to the list!
Let’s do it!
add(String s)

Adding a String to the end of the list.
Two Cases to Handle

Adding at the end of an empty list

Append at the end of a non-empty list
Two Cases to Handle

Adding at the end of an empty list

Adding to a non-empty list

Technically true, but why repeat ourselves?
We already wrote add to insert at any index, so we can just add at the end of the list!
public void add(String s) {
    add(size(), s);
}
Get
Two cases to handle:

1. List is empty
2. List non-empty
Two cases to handle:

1. List is empty, or requested index is too big/small
   a. Throw new IllegalArgumentException
2. List non-empty
Two cases to handle:

1. Invalid index
2. List non-empty
   a. Start at head

```java
l.get(2);
```
Two cases to handle:

1. List is empty
2. List non-empty
   a. Start at head
   b. Follow next pointers \textit{index} times.

\texttt{l.get(2);}
Two cases to handle:

1. List is empty
2. List non-empty
   a. Start at head
   b. Follow next pointers $i$ times.

```java
l.get(2);
```
Two cases to handle:

1. List is empty
2. List non-empty
   a. Start at head
   b. Follow next pointers index times.
   c. Return current.data

```java
l.get(2);
```
Writing `get`. What goes here?

```java
if (index < 0 || index >= size) {
    // code block
}
```
Writing `get`. What goes here?

```java
if (index == 0) {
    // code goes here
}
```
LinkedIn lst has 4 elements. What's the index of lst.head.next.next?

0
1
2
3

NullPointerException
If head is not null and I set head.next = head; which of the following is true?

- The LinkedList consists of exactly one node
- The LinkedList is empty
- This results in a NullPointerException
- The program will loop forever
Organize the lines on the right to the correct location for the blanks on the left. This code should handle the case in `get` where we’re getting from a valid index that’s not the head.

```cpp
// move to the node at index

A. ___________

while (i < index) {

  C. ___________

D. ___________

E. ___________

1 curr = curr.next;
2 i++;;
3 Node curr = head;
4 return curr.data;
5 int i = 0;
```
// move to the node at index

A. ___________

B. ___________

while (i < index) {
  
  C. ___________

  D. ___________

}

E. ___________

curr = curr.next;
i++;
Node curr = head;
return curr.data;
int i = 0;
public String get(int index) {
    if (index < 0 || index >= size) {
        throw new IllegalArgumentException();
    }
    if (index == 0) {
        return head.data;
    }
    // move to the node at index
    int i = 0;
    Node curr = head;
    while (i < index) {
        curr = curr.next;
        i++;
    }
    return curr.data;
}
Remove
Two Cases to Handle (after checking for valid input)

Remove at the head of the list

- Copy the data in head
- If head is the only node, remove it and decrement the size of the list
- If head has a next, then set head to be head.next and decrement the size of the list
- Return the data

Remove anywhere else

- Start at the head of the list and traverse the nodes until you're at node at position index - 1, call this node current
- Copy the data in current.next
- Set current.next to current.next.next
- Decrement the size of the list and return the saved data.
Removing at the head

- Copy the data in head
- If head is the only node, remove it and decrement the size of the list
- If head has a next, then set head to be head.next and decrement the size of the list
- Return the data

```java
l.remove(0)
```
Removing at the head

- Copy the data in `head`
- If `head` is the only node, remove it and decrement the size of the list
- If `head` has a `next`, then set `head` to be `head.next` and decrement the size of the list
- Return the data

```java
l.remove(0)
```
Removing at the head

- Copy the data in `head`
- If `head` is the only node, remove it and decrement the size of the list
- If `head` has a `next`, then set `head` to be `head.next` and decrement the size of the list
- Return the data

```java
l.remove(0)
```
Removing at the head, take 2

- Copy the data in `head`
- If `head` is the only node, remove it and decrement the size of the list
- If `head` has a `next`, then set `head` to be `head.next` and decrement the size of the list
- Return the data

```
l.remove(0)
```
Removing at the head, take 2

- Copy the data in `head`
- If `head` is the only node, remove it and decrement the size of the list
- If `head` has a `next`, then set `head` to be `head.next` and decrement the size of the list
- Return the data

l.remove(0)
Removing at the head, take 2

- Copy the data in `head`
- If `head` is the only node, remove it and decrement the size of the list
- If `head` has a `next`, then set `head` to be `head.next` and decrement the size of the list
- Return the data

```java
l.remove(0)
```
Removing within the List

- Start at the head of the list and traverse the nodes until you’re at node at position $index - 1$, call this node $current$
- Copy the data in $current.next$
- Set $current.next$ to $current.next.next$
- Decrement the size of the list and return the saved data.

```java
l.remove(2);
```
Removing within the List

- Start at the head of the list and traverse the nodes until you’re at node at position \( index - 1 \), call this node \( current \).
- Copy the data in \( current.next \).
- Set \( current.next \) to \( current.next.next \).
- Decrement the size of the list and return the saved data.

```java
l.remove(2);
```
Removing within the List

- Start at the head of the list and traverse the nodes until you're at node at position $index - 1$, call this node $current$
- Copy the data in $current.next$
- Set $current.next$ to $current.next.next$
- Decrement the size of the list and return the saved data.

```java
l.remove(2);
```
Removing within the List

- Start at the head of the list and traverse the nodes until you're at node at position $index - 1$, call this node $current$
- Copy the data in $current.next$
- Set $current.next$ to $current.next.next$
- Decrement the size of the list and return the saved data.

```
l.remove(2);
```
Removing within the List

- Start at the head of the list and traverse the nodes until you're at node at position $index - 1$, call this node $current$
- Copy the data in $current.next$
- Set $current.next$ to $current.next.next$
- Decrement the size of the list and return the saved data.

```java
l.remove(2);
```
public String remove(int index) {
    if (index < 0 || index >= size) {
        throw new IllegalArgumentException();
    }
}
Unscramble the lines on the right to finish remove. (error checking already done)

```java
String value = "";
if (A. ________) { // remove at head
    B. __________
    C. __________
} else {
    // move to the correct position
    D. __________
    E. __________;
    while (F. ________) {
        G. __________
        H. __________
    }
    I. __________ // Remember value
    J. __________ // remove node
}
K. __________
L. __________
```

```java
int i = 0;
return value;
Node curr = head;
size--;
index == 0
value = curr.next.data;
i++;
curr = curr.next;
value = head.data;
i <= index - 1
head = head.next;
```
String value = "";
if (A. ______) { // remove at head
    B. _______
    C. _______
} else {
    // move to the correct position
    D. _______
    E. _______
    while (F. ______) {
        G. _______
        H. _______
    }
    I. _______ // Remember value
    J. _______ // remove node
}
K. _______
L. _______
public String remove(int index) {
    if (index < 0 || index >= size) {
        throw new IllegalArgumentException();
    }
    String value = "";
    if (index == 0) { // remove at head
        value = head.data;
        head = head.next;
    } else {
        // move to the correct position
        int i = 0;
        Node curr = head;
        while (i < index - 1) {
            curr = curr.next;
            i++;
        }
        value = curr.next.data; // Remember value
        curr.next = curr.next.next; // remove node
    }
    size--;
    return value;
}
Contains
Just one case! (Whew...)

Searching the list for a target value

- Start at the head of the list
- Check each Node’s data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.

```
1.contains("B");
```
Just one case! (Whew...)

Searching the list for a target value

- Start at the head of the list
- Check each Node’s data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.

```
l.contains("B");
```

"B".equals("A") → Nope!
Just one case! (Whew...)

Searching the list for a target value

- Start at the head of the list
- Check each Node's data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.

```java
l.contains("B");
```

```
"B".equals("B") → return true;
```
What if it's not there?

Searching the list for a target value

- Start at the head of the list
- Check each Node’s data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.

`l.contains("D");`
What if it’s not there?

Searching the list for a target value

- Start at the head of the list
- Check each Node’s data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.

```java
l.contains("D");
```

"D.equals("A") → Nope!"
What if it's not there?

Searching the list for a target value

- Start at the head of the list
- Check each Node's data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.

```java
l.contains("D");
```

"D".equals("B") → Nope!
What if it’s not there?

Searching the list for a target value

- Start at the head of the list
- Check each Node’s data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.

```
l.contains("D");
```

"D".equals("C") → Nope!
What if it's not there?

Searching the list for a target value

- Start at the head of the list
- Check each Node's data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.

```java
l.contains("D");
```
The rest
size()

Returns the current number of elements in the list.

```java
// Returns number of elements in this list
public int size() {
    return size;
}
```
isEmpty()

Returns true when the list is empty.

```java
public boolean isEmpty() {
    return (head == null);
}
```