[Pre-Class Work] 7: Linked Lists

From ListNodes to LinkedLists - Add and toString [Background Reading]

🤔 Motivation

Now that we've worked with the concept of ListNodes, it's time to begin building our LinkedList class!

Recall that since last time, we worked with manually iterating through a series of ListNodes to perform functions such as `get`, `add`, and `remove`. While we can theoretically just use ListNode objects to perform all the same functionality as a List, having to write loops and move pointers each time we want to use List methods would become a pretty redundant and tedious task.

Instead, we can *abstract* away all those neat little ListNode operations into a LinkedList class, thus making methods like `get`, `add`, and `remove` convenient and useable even by clients who have never seen a ListNode before (much like many of us while we were learning about the List ADT for the first time)!

🔨 Writing the Class

Below is the ListNode class that we've been working with:

```
// Class that represents a single node containing an
// integer value.
public class ListNode {
    public final int data;
    public ListNode next;

    // Constructs a ListNode with the given data
    public ListNode(int data) {
        // Sets the next field to null, meaning there
        // is no next linked node.
        this(data, null);
    }

    // Constructs a ListNode with the given data
    // and given next node.
    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
```

```
}
```

Let's begin to create a LinkedList using ListNode objects. Last time in lecture, we practiced creating methods using loop traversal strategies to manipulate elements in a List.

Below, we have a preliminary **LinkedIntList** class that contains an `add` and `toString` method:

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    // the head of the LinkedList
    private ListNode front;


    // post: Adds the given value to the end of this list
    public void add(int value) {
        //case that the List contains no elements
        if (front == null) {
            front = new ListNode(value);
        } else {
            ListNode current = front;
            while (current.next != null) {
                current = current.next;
            }
            current.next = new ListNode(value);
        }
    }

    // post: returns a String representation of the contents of the list in one line,
    //         separated by spaces
    public String toString() {
        String listString = "";
        ListNode current = front;
        while (current != null) {
            listString += current.data + " ";
            current = current.next;
        }
        return listString;
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
```

```
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

> Note that we have moved our ListNode class into our LinkedIntList class! Doing this allows us to create and use ListNode objects in LinkedIntList without any extra imports. We can additionally choose to make this inner class `private`, a good implementation design decision that we can discuss more at length in section (although `public` works just fine for now).

Now that we've abstracted them away into our LinkedIntList class, we can call these methods without needing to rewrite our while loops and temp pointers. Neat!

Main Points

- To avoid having to write loops and move pointers each time we want to use List methods, we *abstracted* away all those neat little ListNode operations into a LinkedList class
- Methods like `get`, `add`, and `remove` are now convenient and useable even by clients who have never seen a ListNode before!
- We started work on our **LinkedIntList** class that contains an `add` and `toString` method!

Constructors [Background Reading]

Notice that our LinkedIntList class is missing a few crucial pieces. We've practiced implementing some List methods already in lecture so far -- let's add them into our class one by one!

Constructors

When writing the constructor, it's important to consider the *default state* of our object. In the case of our LinkedIntList, it would make sense that creating a new instance of this object would just initialize an empty list with no values. Since the data of our list is stored by our `ListNode front` field, we can set it to be empty (null):

```
// post: constructs an empty LinkedIntList
public LinkedIntList() {
    this.front = null;
}
```

Note that since the default state of any object (including `ListNode` objects) is null upon declaration, so re-initializing the field here is not technically required!

We can also define more constructors for our LinkedIntList depending on our desired functionality. For example, it may be useful to create a LinkedIntList object directly from an array of integers:

```
// post: constructs a LinkedIntList from the given int[] nums
//       constructs an empty LinkedIntList if nums is null or has a size of 0
public LinkedIntList(int[] nums) {
    if (nums == null || nums.length == 0) {
        this.front = null;
    } else {
        for (int i = 0; i < nums.length; i++) {
            this.add(nums[i]);
        }
    }
}
```

Notice how we don't need to handle creating the first node in the LinkedIntList in this implementation. This is because we are leveraging the existing `add` method we wrote, which handles that edge case for us!

Now that we've defined two constructors with the same name but different parameters (overloading), let's simplify our code using the `this()` constructor keyword:

```
// post: constructs an empty LinkedIntList
public LinkedIntList() {
```

```
    this(null);
}

// post: constructs a LinkedIntList from the given int[] nums
//       constructs an empty LinkedIntList if nums is null or has a size of 0
public LinkedIntList(int[] nums) {
    if (nums == null || nums.length == 0) {
        this.front = null;
    } else {
        for (int i = 0; i < nums.length; i++) {
            this.add(nums[i]);
        }
    }
}
```

Be careful when overloading a method with `null` inputs! Since any object can be null, the compiler won't know which method to call when we pass in the null input if we have another constructor that also takes in an object parameter, throwing an error due to the ambiguous reference.

Main Points

- This time, we implemented the constructor of our LinkedLintList class.
    - We considered the *default state* of our LinkedIntList by creating a constructor that initialized an empty list no values.
    - We created another constructor to create a LinkedIntList object directly from an array of integers.
- These two constructors use **overloading** since they have the same name but different parameters.
- We didn't need to handle creating the first node in the LinkedIntList in this implementation because we are leveraging the existing `add` method we wrote, which handles that edge case for us.
- We should be careful when overloading a method with `null` inputs since any object can be `null` (the compiler won't know what method to call when we pass in `null` input).

Size [Background Reading]

Size

Next, let's implement a size method. Recall that when writing our TenStack class in the first week of section, we used a `int size` field to store and iteratively update size as we added/removed elements from our collection. We can apply the same principle below here:

```
// post: returns the number of elements in the list
public int size() {
    return this.size;
}
```

Now we can update the rest of our class accordingly (note the changes made to the constructors and the `add` method):

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    // post: constructs an empty LinkedIntList
    public LinkedIntList() {
        this(null);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
    public LinkedIntList(int[] nums) {
        this.size = 0;
        if (nums == null || nums.length == 0) {
            this.front = null;
        } else {
            for (int i = 0; i < nums.length; i++) {
                this.add(nums[i]);
            }
        }
    }

    // post: Adds the given value to the end of this list
    public void add(int value) {
        // case that the List contains no elements
        if (front == null) {
            front = new ListNode(value);
        } else {
            ListNode current = front;
```

```
            while (current.next != null) {
                current = current.next;
            }
            current.next = new ListNode(value);
        }
        this.size++;
    }

    // post: returns a String representation of the contents of the list in one line,
    //         separated by spaces
    public String toString() {
        String listString = new String();
        ListNode current = front;
        while (current != null) {
            listString += current.data + " ";
            current = current.next;
        }
        return listString;
    }

    // post: returns the number of elements in the list
    public int size() {
        return this.size;
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Same as with the constructor methods from the last slide, notice how we do not need to explicitly updating our size field in our second constructor. This is because we changed the add method to iteratively increment size directly, which we call from the constructor. Since we begin with a size of 0, we should have the correct size after we terminate our loop!

Main Points

- We added an `int size` field to `LinkedIntList` to store and iteratively update size as we added/removed elements from our collection.
- We also added a `size()` method to return the size of our `LinkedIntList` using our field!

```
// post: returns the number of elements in the list
public int size() {
    return this.size;
}
```

Get [Background Reading]

Get

Now, let's implement a get method for our LinkedIntList so that we can retrieve items by index:

```
// post: returns the value of the element at the given int index of the list
public int get(int index) {
    ListNode temp = this.front;

    // moves the temp pointer to the ListNode at the given index
    for (int i = 0; i < index; i++) {
        temp = temp.next;
    }
    return temp.data;
}
```

It's important to think about edge cases too. What happens if the index is out of bounds? What happens if the list is empty? Why don't we add a check for valid input, updating our method documentation accordingly:

```
// post: returns the value of the element at the given int index of the list
//       throws an IllegalArgumentException if the index is out of bounds
public int get(int index) {
    if (index < 0 || index > this.size - 1) {
        throw new IllegalArgumentException("not a valid index!");
    }
    ListNode temp = this.front;
    for (int i = 0; i < index; i++) {
        temp = temp.next;
    }
    return temp.data;
}
```

This is a great start! Next, we can move on to a more challenging method.

Main Points

- We implemented a `get` method for our LinkedIntList so that we can retrieve items by index.
- After considering edge cases, we modified our `get` method in this way:
    - We threw an `IllegalArgumentException` when the index was out of bounds (e.g. negative or greater than the size of the list)
    - This also works if the list is empty!

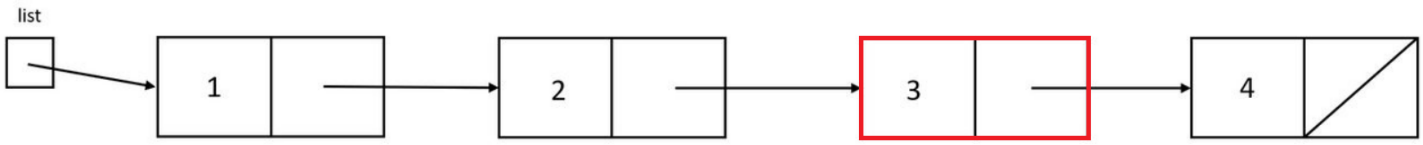
Remove [Background Reading]

Remove

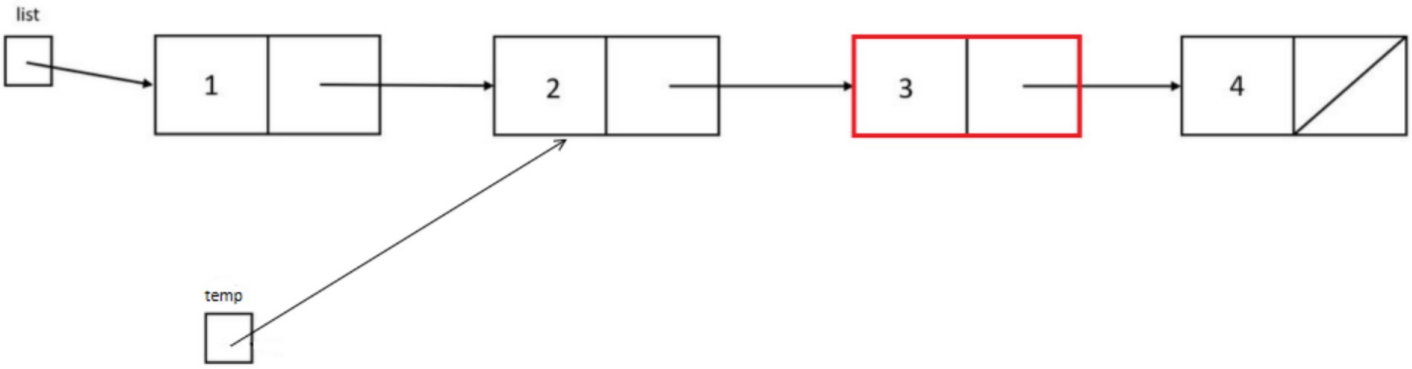
Now, we can write a remove method that can delete an element at the given index from the list. Like with our `get` method, our method only needs to handle *valid* index inputs. However, this method will be a little trickier to implement because we now need to deal with manipulating ListNode pointers.

Recall that the only real way to delete a single ListNode from a LinkedList is by **dereferencing** that particular node. In most situations, we do this by updating our `.next` field of the ListNode **before** the one we want to remove, since it holds the reference to our target ListNode.

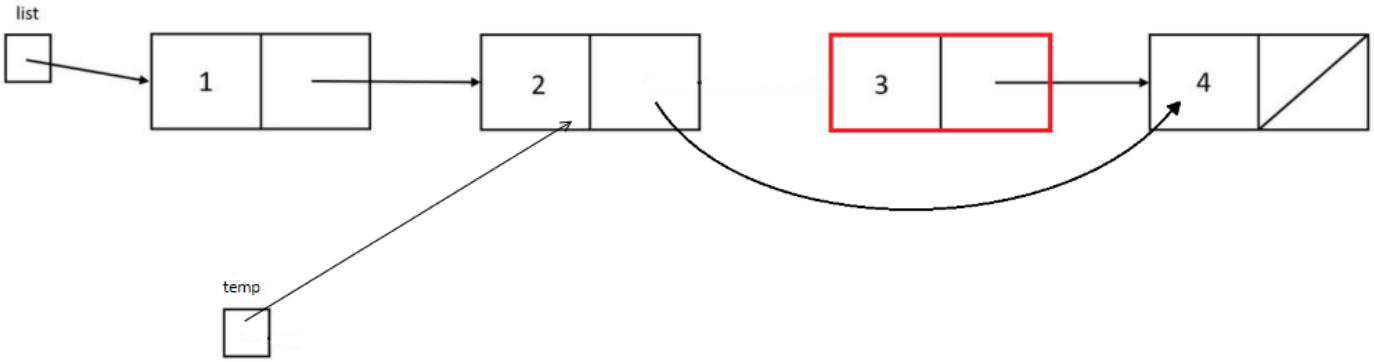
For example, if we wanted to remove the third node from the following LinkedList:



We could start by creating a **temporary pointer** and iterate it to the preceding (second) node:



And then set the second node's next field to skip over the third node using `temp.next`:



Since there is no pointer holding on to the third node anymore, it gets dropped (effectively removed)!

Here's the code for these steps below:

```
// post: removes and returns the value of the element at the given int index of the list
//       throws an IllegalArgumentException if the index is out of bounds
public int remove(int targetIndex) {
    if (targetIndex < 0 || targetIndex > this.size - 1) {
        throw new IllegalArgumentException("not a valid index!");
    }
    ListNode temp = this.front;

    // index - 1 gets us to the ListNode one before our target for removal
    for (int i = 0; i < targetIndex - 1; i++) {
        temp = temp.next;
    }
    int data = temp.next.data;

    // dereferencing the target ListNode by skipping over it to the next ListNode in the list
    temp.next = temp.next.next;

    this.size--;
    return data;
}
```

This method is a little more complex than the others we've made up to this point, so let's give it a test!

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    public static void main(String[] args) {

        // Creating our test LinkedIntList
        int[] numbers1 = new int[] {1, 2, 3};
        int[] numbers2 = new int[] {1, 2, 3, 4, 5, 6};
        LinkedIntList removeTest1 = new LinkedIntList(numbers1);
```

```
        LinkedIntList removeTest2 = new LinkedIntList(numbers2);

        System.out.println(removeTest1);
        removeTest1.remove(1);
        System.out.println(removeTest1);
        System.out.println();

        System.out.println(removeTest2);
        removeTest2.remove(4);
        System.out.println(removeTest2);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
    public LinkedIntList(int[] nums) {
        this.size = 0;
        if (nums == null || nums.length == 0) {
            this.front = null;
        } else {
            for (int i = 0; i < nums.length; i++) {
                this.add(nums[i]);
            }
        }
    }

    // post: Adds the given value to the end of this list
    public void add(int value) {

        //case that the List contains no elements
        if (front == null) {
            front = new ListNode(value);
        } else {
            ListNode current = front;
            while (current.next != null) {
                current = current.next;
            }
            current.next = new ListNode(value);
        }
        this.size++;
    }

    // post: returns a String representation of the contents of the list in one line,
    //         separated by spaces
    public String toString() {
        String listString = "";
        ListNode current = front;
        while (current != null) {
            listString += current.data + " ";
            current = current.next;
        }
        return listString;
    }

    //post: returns the number of elements in the list
```

```
    public int size() {
        return this.size;
    }

    // post: removes and returns the value of the element at the given int index of the list
    //       throws an IllegalArgumentException if the index is out of bounds
    public int remove(int targetIndex) {
        if (targetIndex < 0 || targetIndex > this.size - 1) {
            throw new IllegalArgumentException("not a valid index!");
        }
        ListNode temp = this.front;

        // index - 1 gets us to the ListNode one before our target for removal
        for (int i = 0; i < targetIndex - 1; i++) {
            temp = temp.next;
        }
        int data = temp.next.data;

        // dereferencing the target ListNode by skipping over it to the next ListNode in the list
        temp.next = temp.next.next;

        this.size--;
        return data;
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

So far, it seems like we are properly handling remove calls from the middle of our list. However, like with `get`, it is important to consider different edge cases. We already properly handle invalid index inputs with our exception, but let's consider a few more possible states of our list for the `remove` method call in addition to the most basic case of removing from the middle:

- If we remove at the very end
- If we have an empty list
- If we remove at the very beginning
- If we remove from a list with only 1 element

Let's update our test cases to cover these situations:

Removing the very last element

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    public static void main(String[] args) {
        // Testing remove on very last element
        int[] numbers = new int[] {1, 2, 3};
        LinkedIntList removeFromEnd = new LinkedIntList(numbers);

        System.out.println(removeFromEnd);
        System.out.println("Removing the " + removeFromEnd.remove(2));
        System.out.println(removeFromEnd);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
    public LinkedIntList(int[] nums) {
        this.size = 0;
        if (nums == null || nums.length == 0) {
            this.front = null;
        } else {
            for (int i = 0; i < nums.length; i++) {
                this.add(nums[i]);
            }
        }
    }

    // post: Adds the given value to the end of this list
    public void add(int value) {

        // case that the List contains no elements
        if (front == null) {
            front = new ListNode(value);
        } else {
            ListNode current = front;
            while (current.next != null) {
                current = current.next;
            }
```

```
            current.next = new ListNode(value);
        }
        this.size++;
    }

    // post: returns a String representation of the contents of the list in one line,
    //         separated by spaces
    public String toString() {
        String listString = "";
        ListNode current = front;
        while (current != null) {
            listString += current.data + " ";
            current = current.next;
        }
        return listString;
    }

    // post: returns the number of elements in the list
    public int size() {
        return this.size;
    }

    // post: removes and returns the value of the element at the given int index of the list
    //       throws an IllegalArgumentException if the index is out of bounds
    public int remove(int targetIndex) {
        if (targetIndex < 0 || targetIndex > this.size - 1) {
            throw new IllegalArgumentException("not a valid index!");
        }
        ListNode temp = this.front;

        // index - 1 gets us to the ListNode one before our target for removal
        for (int i = 0; i < targetIndex - 1; i++) {
            temp = temp.next;
        }
        int data = temp.next.data;

        // dereferencing the target ListNode by skipping over it to the next ListNode in the list
        temp.next = temp.next.next;

        this.size--;
        return data;
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
```

```
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Everything looks good!

Removing from an empty list

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    public static void main(String[] args) {
        // Testing empty list removal
        LinkedIntList removeEmpty = new LinkedIntList();
        System.out.println("Removing the " + removeEmpty.remove(0));
        System.out.println(removeEmpty);
    }

    // post: constructs an empty LinkedIntList
    public LinkedIntList() {
        this(null);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
    public LinkedIntList(int[] nums) {
        this.size = 0;
        if (nums == null || nums.length == 0) {
            this.front = null;
        } else {
            for (int i = 0; i < nums.length; i++) {
                this.add(nums[i]);
            }
        }
    }

    // post: Adds the given value to the end of this list
    public void add(int value) {
```

```
        // case that the List contains no elements
        if (front == null) {
            front = new ListNode(value);
        } else {
            ListNode current = front;
            while (current.next != null) {
                current = current.next;
            }
            current.next = new ListNode(value);
        }
        this.size++;
    }

    // post: returns a String representation of the contents of the list in one line,
    //         separated by spaces
    public String toString() {
        String listString = "";
        ListNode current = front;
        while (current != null) {
            listString += current.data + " ";
            current = current.next;
        }
        return listString;
    }

    // post: returns the number of elements in the list
    public int size() {
        return this.size;
    }

    // post: removes and returns the value of the element at the given int index of the list
    //       throws an IllegalArgumentException if the index is out of bounds
    public int remove(int targetIndex) {
        if (targetIndex < 0 || targetIndex > this.size - 1) {
            throw new IllegalArgumentException("not a valid index!");
        }
        ListNode temp = this.front;

        // index - 1 gets us to the ListNode one before our target for removal
        for (int i = 0; i < targetIndex - 1; i++) {
            temp = temp.next;
        }
        int data = temp.next.data;

        // dereferencing the target ListNode by skipping over it to the next ListNode in the list
        temp.next = temp.next.next;

        this.size--;
        return data;
    }


    // Inner class that represents a single node containing an
    // integer value.
```

```
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Same here! Since the list is empty, any index argument we pass to the remove method should trigger the `IllegalArgumentException`.

Removing the very first element

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    public static void main(String[] args) {
        // Testing remove on very first element
        int[] numbers = new int[] {1, 2, 3};
        LinkedIntList removeFromFront = new LinkedIntList(numbers);

        System.out.println(removeFromFront);
        System.out.println("Removing the " + removeFromFront.remove(0));
        System.out.println(removeFromFront);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
    public LinkedIntList(int[] nums) {
        this.size = 0;
        if (nums == null || nums.length == 0) {
            this.front = null;
        } else {
            for (int i = 0; i < nums.length; i++) {
                this.add(nums[i]);
```

```
        }
    }
}

// post: Adds the given value to the end of this list
public void add(int value) {

    // case that the List contains no elements
    if (front == null) {
        front = new ListNode(value);
    } else {
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
    this.size++;
}

// post: returns a String representation of the contents of the list in one line,
//         separated by spaces
public String toString() {
    String listString = "";
    ListNode current = front;
    while (current != null) {
        listString += current.data + " ";
        current = current.next;
    }
    return listString;
}

// post: returns the number of elements in the list
public int size() {
    return this.size;
}

// post: removes and returns the value of the element at the given int index of the list
//      throws an IllegalArgumentException if the index is out of bounds
public int remove(int targetIndex) {
    if (targetIndex < 0 || targetIndex > this.size - 1) {
        throw new IllegalArgumentException("not a valid index!");
    }
    ListNode temp = this.front;

    // index - 1 gets us to the ListNode one before our target for removal
    for (int i = 0; i < targetIndex - 1; i++) {
        temp = temp.next;
    }
    int data = temp.next.data;

    // dereferencing the target ListNode by skipping over it to the next ListNode in the list
    temp.next = temp.next.next;
```

```
        this.size--;
        return data;
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Hmm... something's not quite right here. We wanted to remove the element at index 0 (the 1), however it looks like we removed the element at index 1 instead (the 2). Let's keep testing.

Removing the only element from a single-element list

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    public static void main(String[] args) {
        // Testing single element list removal
        int[] number = new int[] {0};
        LinkedIntList removeSingle = new LinkedIntList(number);

        System.out.println(removeSingle);
        System.out.println("Removing the " + removeSingle.remove(0));
        System.out.println(removeSingle);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
```

```
public LinkedIntList(int[] nums) {
    this.size = 0;
    if (nums == null || nums.length == 0) {
        this.front = null;
    } else {
        for (int i = 0; i < nums.length; i++) {
            this.add(nums[i]);
        }
    }
}

// post: Adds the given value to the end of this list
public void add(int value) {

    // case that the List contains no elements
    if (front == null) {
        front = new ListNode(value);
    } else {
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
    this.size++;
}

// post: returns a String representation of the contents of the list in one line,
//         separated by spaces
public String toString() {
    String listString = "";
    ListNode current = front;
    while (current != null) {
        listString += current.data + " ";
        current = current.next;
    }
    return listString;
}

// post: returns the number of elements in the list
public int size() {
    return this.size;
}

// post: removes and returns the value of the element at the given int index of the list
//       throws an IllegalArgumentException if the index is out of bounds
public int remove(int targetIndex) {
    if (targetIndex < 0 || targetIndex > this.size - 1) {
        throw new IllegalArgumentException("not a valid index!");
    }
    ListNode temp = this.front;

    // index - 1 gets us to the ListNode one before our target for removal
    for (int i = 0; i < targetIndex - 1; i++) {
```

```
            temp = temp.next;
        }
        int data = temp.next.data;

        // dereferencing the target ListNode by skipping over it to the next ListNode in the list
        temp.next = temp.next.next;

        this.size--;
        return data;
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Uh oh! The dreaded `NullPointerException`!

From these edge-case tests, we can deduce that the problem lies when we want to remove the very first element of our list (since the single-element scenario is just a special version of removing the first element).

These issues occur because of the way we've configured our loop bounds. Since we are iterating our for loop to `targetIndex - 1`, notice how our cases for handling index 0 and index 1 are the same -- neither input would allow the loop to run since 0 is not less than either -1 or 0, resulting both of them passing through.

That works great in the case of removing at index 1 (as tested above), but not so great when removing at index 0. Especially in the case of a single-element list, this causes problems because our code assumes there are ListNodes after the `targetIndex` node by calling `temp.next.data` and `temp.next.next`. However, if `temp.next` happens to be `null`, we cause a `NullPointerException` since null objects don't have fields!

Thus, after we revise our code:

```
// post: removes and returns the value of the element at the given int index of the list
//       throws an IllegalArgumentException if the index is out of bounds
public int remove(int targetIndex) {
    if (targetIndex < 0 || targetIndex > this.size - 1) {
        throw new IllegalArgumentException("not a valid index!");
    }
    ListNode temp = this.front;
    int data = temp.data;

    // removing very first element
    if (targetIndex == 0) {

        // skipping front pointer to the next second element
        // (or null, if the list only contains one element)
        this.front = temp.next;
        this.size--;
        return data;
    }

    for (int i = 0; i < targetIndex - 1; i++) {
        temp = temp.next;
    }
    data = temp.next.data;
    temp.next = temp.next.next;
    this.size--;
    return data;
}
```

Run the tests!

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    public static void main(String[] args) {
        // Testing remove on very first element
        int[] numbers = new int[] {1, 2, 3};
        LinkedIntList removeFromFront = new LinkedIntList(numbers);

        System.out.println(removeFromFront);
        System.out.println("Removing the " + removeFromFront.remove(0));
        System.out.println(removeFromFront);

        // Testing single element list removal
        int[] number = new int[] {0};
        LinkedIntList removeSingle = new LinkedIntList(number);

        System.out.println(removeSingle);
```

```
        System.out.println("Removing the " + removeSingle.remove(0));
        System.out.println(removeSingle);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
    public LinkedIntList(int[] nums) {
        this.size = 0;
        if (nums == null || nums.length == 0) {
            this.front = null;
        } else {
            for (int i = 0; i < nums.length; i++) {
                this.add(nums[i]);
            }
        }
    }

    // post: Adds the given value to the end of this list
    public void add(int value) {

        // case that the List contains no elements
        if (front == null) {
            front = new ListNode(value);
        } else {
            ListNode current = front;
            while (current.next != null) {
                current = current.next;
            }
            current.next = new ListNode(value);
        }
        this.size++;
    }

    // post: returns a String representation of the contents of the list in one line,
    //         separated by spaces
    public String toString() {
        String listString = "";
        ListNode current = front;
        while (current != null) {
            listString += current.data + " ";
            current = current.next;
        }
        return listString;
    }

    // post: returns the number of elements in the list
    public int size() {
        return this.size;
    }

    // post: removes and returns the value of the element at the given int index of the list
    //       throws an IllegalArgumentException if the index is out of bounds
    public int remove(int targetIndex) {
        if (targetIndex < 0 || targetIndex > this.size - 1) {
```

```
            throw new IllegalArgumentException("not a valid index!");
        }
        ListNode temp = this.front;
        int data = temp.data;

        if (targetIndex == 0) {
            this.front = temp.next;
            this.size--;
            return data;
        }

        for (int i = 0; i < targetIndex - 1; i++) {
            temp = temp.next;
        }
        data = temp.next.data;
        temp.next = temp.next.next;
        this.size--;
        return data;
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Hooray! Everything's looking good 😎.

🔑 Main Points

- We created a remove method that could delete an element at the given index from the list.
    - This method was a little trickier to implement because we needed to manipulate ListNode pointers.

- Our method first checked whether the given index is valid and threw an `IllegalArgumentException` if the index was out of bounds.
- To delete a single node, we needed to update the `next` pointer of the node right before the one we wanted to remove.
    - In our method, we iterate to the node before the one we want to delete, store the data from the node we want to delete, and update the `next` pointer to skip over the node we want to delete.
- We may get a `NullPointerException` if we try to get the `next` node of a node that is `null`! We updated our code to avoid this issue.

Our LinkedIntList Class [Background Reading]

The LinkedIntList Class

Here's what our LinkedIntList class looks like when we put everything together. Feel free to create your own test case scenarios and mess around with it!

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    public static void main(String[] args) {
        int[] numbers = new int[] {1, 2, 3};
        LinkedIntList testList = new LinkedIntList(numbers);
    }

    // post: constructs an empty LinkedIntList
    public LinkedIntList() {
        this(null);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
    public LinkedIntList(int[] nums) {
        this.size = 0;
        if (nums == null || nums.length == 0) {
            this.front = null;
        } else {
            for (int i = 0; i < nums.length; i++) {
                this.add(nums[i]);
            }
        }
    }

    // post: Adds the given value to the end of this list
    public void add(int value) {
        if (front == null) {
            front = new ListNode(value);
        } else {
            ListNode current = front;
            while (current.next != null) {
                current = current.next;
            }
            current.next = new ListNode(value);
        }
        this.size++;
```

```
    }

    // post: returns a String representation of the contents of the list in one line,
    //       separated by spaces
    public String toString() {
        String listString = "";
        ListNode current = front;
        while (current != null) {
            listString += current.data + " ";
            current = current.next;
        }
        return listString;
    }

    // post: returns the number of elements in the list
    public int size() {
        return this.size;
    }

    // post: returns the value of the element at the given int index of the list
    //       throws an IllegalArgumentException if the index is out of bounds
    public int get(int index) {
        if (index < 0 || index > this.size - 1) {
            throw new IllegalArgumentException("not a valid index!");
        }
        ListNode temp = this.front;
        for (int i = 0; i < index; i++) {
            temp = temp.next;
        }
        return temp.data;
    }

    // post: removes and returns the value of the element at the given int index of the list
    //       throws an IllegalArgumentException if the index is out of bounds
    public int remove(int targetIndex) {
        if (targetIndex < 0 || targetIndex > this.size - 1) {
            throw new IllegalArgumentException("not a valid index!");
        }
        ListNode temp = this.front;
        int data = temp.data;

        if (targetIndex == 0) {
            this.front = temp.next;
            this.size--;
            return data;
        }

        for (int i = 0; i < targetIndex - 1; i++) {
            temp = temp.next;
        }
        data = temp.next.data;
        temp.next = temp.next.next;
        this.size--;
        return data;
```

```
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

```
// LinkedIntList.java
// Represents a list of integers.
public class LinkedIntList {

    private ListNode front;
    private int size;

    public static void main(String[] args) {
        int[] numbers = new int[] {1, 2, 3};
        LinkedIntList testList = new LinkedIntList(numbers);
        System.out.println(testList);
    }

    // post: constructs an empty LinkedIntList
    public LinkedIntList() {
        this(null);
    }

    // post: constructs a LinkedIntList from the given int[] nums
    //       constructs an empty LinkedIntList if nums is null or has a size of 0
    public LinkedIntList(int[] nums) {
        this.size = 0;
        if (nums == null || nums.length == 0) {
            this.front = null;
        } else {
            for (int i = 0; i < nums.length; i++) {
                this.add(nums[i]);
            }
        }
```

```
}

// post: Adds the given value to the end of this list
public void add(int value) {

    if (front == null) {
        front = new ListNode(value);
    } else {
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
    this.size++;
}

// post: returns a String representation of the contents of the list in one line,
//         separated by spaces
public String toString() {
    String listString = new String();
    ListNode current = front;
    while (current != null) {
        listString += current.data;
        current = current.next;
    }
    return listString;
}

// post: returns the number of elements in the list
public int size() {
    return this.size;
}

// post: returns the value of the element at the given int index of the list
//       throws an IllegalArgumentException if the index is out of bounds
public int get(int index) {
    if (index < 0 || index > this.size - 1) {
        throw new IllegalArgumentException("not a valid index!");
    }
    ListNode temp = this.front;
    for (int i = 0; i < index; i++) {
        temp = temp.next;
    }
    return temp.data;
}

// post: removes and returns the value of the element at the given int index of the list
//       throws an IllegalArgumentException if the index is out of bounds
public int remove(int targetIndex) {
    if (targetIndex < 0 || targetIndex > this.size - 1) {
        throw new IllegalArgumentException("not a valid index!");
    }
    ListNode temp = this.front;
```

```
        int data = temp.data;

        if (targetIndex == 0) {
            this.front = temp.next;
            this.size--;
            return data;
        }

        for (int i = 0; i < targetIndex - 1; i++) {
            temp = temp.next;
        }
        data = temp.next.data;
        temp.next = temp.next.next;
        this.size--;
        return data;
    }


    // Inner class that represents a single node containing an
    // integer value.
    public static class ListNode {
        public final int data;
        public ListNode next;

        // Constructs a ListNode with the given data
        public ListNode(int data) {
            // Sets the next field to null, meaning there
            // is no next linked node.
            this(data, null);
        }

        // Constructs a ListNode with the given data
        // and given next node.
        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

Our LinkedIntList class isn't fully complete and ready to implement the List interface quite yet, but it has enough basic functionality for now. Let's go to the next slide to see what else we can add!

Main Points

- Our new `LinkedIntList` is complete! We now have
  - Two different constructors
  - An add, toString, size, get, and remove methods
- Mess around with it as much as you'd like!

Implementing More Methods [Programming Question]

We've created a lot of basic functionality for our LinkedIntList class, but Java's List Interface is quite comprehensive and contains many more methods. Go ahead and try to implement a few extra methods!

Starter Methods:

- `clear()` -- removes every item from the LinkedIntList
- `contains(int element)` -- returns true or false depending on if the LinkedIntList contains the given element
- `indexOf(int element)` -- returns the index of the first occurence of the given element in the LinkedIntList, or -1 if the given element is not contained in the LinkedIntList

More Complex Methods:

- `add(int index, int element)` -- adds the given element at the specified index to the LinkedIntList
- `set(int index, int element)` -- replaces the existing element at the given index in the LinkedIntList with the new given element (be careful, note the `final` keyword for the `data` field in the ListNode class!)

Remember that in order for a class to implement an interface, it must contain **all** methods required by that interface!