Intro to Linked Lists

Prequisites

Students should be familiar with the following:

- Java syntax (data types, references, conditionals, loops, classes, scope, exceptions)
- Basic object-oriented programming
- Building and running Java files

Learning Objectives

By the end of this lesson, students should be able to:

- Visualize the **linked list** data structure.
- Identify a linked list **node** and its components.
- **Retrieve**, **update**, **insert**, and **delete** elements in a linked list via traversal and recognize the edge cases of such operations.
- Implement a simple **singly-linked list (SLL)** in Java.
- *Implement structural optimizations to the SLL, such as tracking size and last element, and adding a Sentinel node.

This lesson should prepare students for:

- Implementing a **doubly-linked** and **circular linked list**.
- Analyzing the **runtime complexity** of linked list operations.
- Recognizing common linked list use-cases

Defining a List

(a) How do you define a list?

(b) What are a list's essential properties?

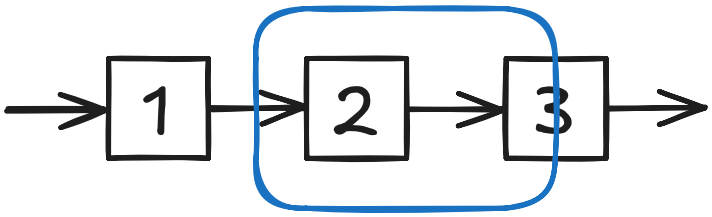
(c) What should we be able to do with a list?

Boxes and Arrows

Let's write out a list of numbers from left to right and identify some of the properties we discussed.

Draw a box around each value and represent the ordering (left-to-right) with an arrow between adjacent boxes.

Now let's go about writing a class to represent this data. Let's zoom in on a single box. **What information does this box need to be aware of?**



Fill out the Java class below to represent the elements of the box.

```
public class Box {



    public Box(                        ) {



    }
}
```

With this `Box` class, we can now build up a "list" by chaining these boxes together one at a time.

This is a very rudimentary **singly-linked list**: a chain of elements, or **nodes**, that are linked together using references from one node to the next.

Managing Boxes

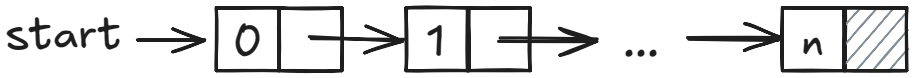
Let's now think about how we might make our chain of boxes useful.

If we are to use our chain of boxes as a list, we'll need to be able to (1) calculate the *size* of our list, (2) *get* (*retrieve*) values from it, (3) *set* (*update*) values in it, (4) *insert* items into it, and (5) *remove* (*delete*) items from it.

We'll go through these conceptually one by one.

Size

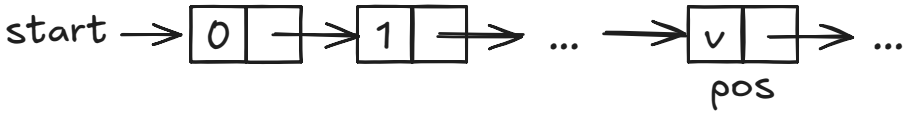
Goal: Determine how many boxes are in the chain.



Complete `TODO 1` in the `Box.java` exercises.

Get

Goal: Get the value stored in the box at a position `pos`.

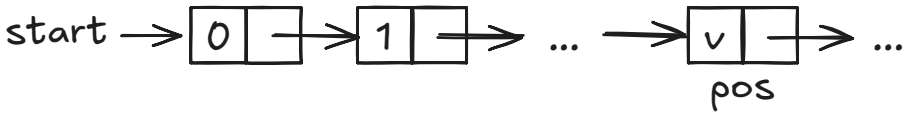


When is `pos` valid?

Complete `TODO 2` in the `Box.java` exercises.

Set

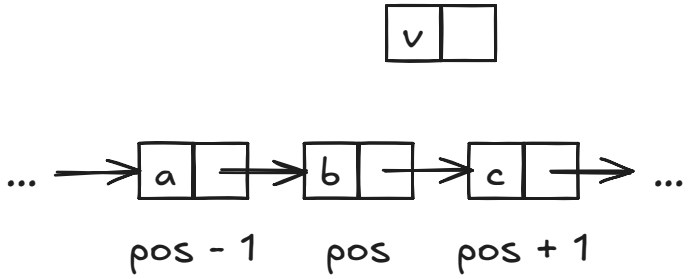
Goal: Set the value stored in the box at position `pos` to a new value `val`.



Complete `TODO 3` in the `Box.java` exercises.

Insert

Goal: Insert a new box at position `pos` with a value `val`.

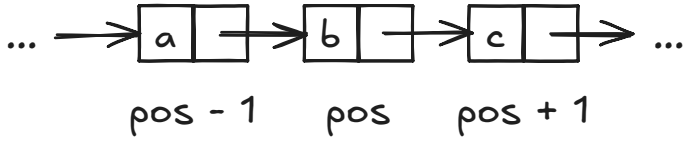


Would this process be different if we are inserting the box at the front of the list?

Complete `TODO 4` and `TODO 5` in the `Box.java` exercises.

Remove

Goal: Remove the box at position `pos`.



Is this process different if we are removing from the beginning or end of the list?

Complete `TODO 6` and `TODO 7` in the `Box.java` exercises.

A Better Interface

It's quite awkward to work directly with our `Box` class. Let's clean this up by creating an `IntLinkedList` class that has `size` `get`, `set`, `insert`, and `remove` methods implemented for it.

Complete the `IntLinkedList` class defined below by implementing the unfinished class methods.

This is also available in the `IntLinkedList.java` exercises file.

```
public class IntLinkedList {

    // a private nested class for nodes
    private class IntNode {
        int value;
        IntNode next;

        IntNode(int value, IntNode next) {
            this.value = value;
            this.next = next;
        }
    }

    /****** CLASS FIELDS ******/

    // a reference to the first node in the list
    private IntNode first;


    /****** CLASS METHODS ******/

    public IntLinkedList() {
        first = null;
    }

    public int size() {                    // YOUR CODE HERE









    }

    public int get(int pos) throws IndexOutOfBoundsException {
        // YOUR CODE HERE
```

```
    }

    public void set(int pos, int val) throws IndexOutOfBoundsException {
        // YOUR CODE HERE









    }

    public void insert(int pos, int val) throws IndexOutOfBoundsException {
        // YOUR CODE HERE









    }

    public void remove(int pos) throws IndexOutOfBoundsException {
        // YOUR CODE HERE









    }
}
```

More Improvements

Now that we've implemented a minimally functional linked list, let's now think about what we can do to improve it a bit.

Of course, there are many, many things we could do to improve upon our linked list, but I want to call attention to 3 things in particular:

1. the `size()` function
2. inserting at the end of the list
3. edge case awkwardness

A `length` Field

(a) **How long does it take to find get the size of our list?**

(b) Let us instead create a `length` field to track the list's length that we can return instantly instead of computing it every time the `size` method is called. **How do we need to modify our `IntLinkedList` class to accommodate this?**

Make the above modifications to `IntLinkedList.java` exercises.

An `addLast` Method

(a) **How long does it take to insert a value at the end of the list?**

(b) Let's create a `IntNode last` field that we want to always refer to the last node in the list. We'll now define an `addLast` method that can quickly add a new node to the end of the list. **Use the new `last` field to complete the `addLast` method below.**

```
// adds the value to the end of the list
public void addLast(int val) {
    // YOUR CODE HERE




}
```

Make the above modifications to `IntLinkedList.java` exercises.

Pesky Edge Cases

(a) **What were our edge cases for insertion and deletion?**

(b) Assuming we have implemented out `length` field, **how can we modify our functions to check for an out-of-bounds index *before* traversing the list?**

(c) Let's now add a dummy node at the front of our linked list. The dummy node should always be there, and never be counted toward the size of the list. **How would we modify the `insert` and `remove` functions to account for this dummy node?**

Make the above modifications to `IntLinkedList.java` exercises.

The dummy node that we have added to our linked list implementation is called a **sentinel node**, and is often used in many implementations of linked lists to reduce edge cases, and sometimes as a marker during traversal.

Summary

Definitions:

Linked List A list implementation represented as a reference chain of nodes.

Linked List Node A container abstraction that holds a value to be stored, as well as a reference to an adjacent node.

We saw how to implement the following operations for a simple **singly-linked list** by traversing and modifying references:

- **size** (or *length*)
- **retrieval** (`get`)
- **update** (`set`)
- **insertion** (`add`/`insert`)
- **deletion** (`remove`/`delete`)

We also made improvements upon our linked list implementation by:

- adding a `length` field that is kept up-to-date during other operations.
- adding an `addLast` method that allows for fast insertion at the end of the list by tracking the `last` node in the list.
- adding a **sentinel node** that simplifies our edge cases during insertion and deletion.