[In Class] Lesson 7 (4/21)

Extending LinkedIntList: add(index, value)

Activity

Write a new method inside the `LinkedIntList` class `add(int index, int value)` that adds a new `ListNode` to the list at the given index. For example, consider "myList" before and after a call to `add(1, 4)` If passed an invalid index throw an `IllegalArgumentException`.

```
myList = front > 1 > 2 > 3
myList.add(1, 4);
myList = front > 1 > 4 > 2 > 3
```

Meet the LinkedIntList class

In previous lessons we have been directly manipulating objects of the `ListNode` type. Now we are going to construct a new class that holds a reference to the first `ListNode` in a collection of unknown number of `ListNodes` and includes various behavior that manages our "Linked List" functionality.

Consider the `LinkedIntList` code shown here. You can see it has a field "front" of type `ListNode`. This is a reference to the first `ListNode`, which in turn may link to another `ListNode`, which may link to another `ListNode` and so on. We need to add methods to this class to manipulate a list like there where our only entry point is a reference to the first node.

We have previously been using the `ListNode` class as a separate file, but now we are going to include it as a "static inner class" to the `LinkedIntList` class. This is a way to make an object definition available to a given class without having to use "import" statements. We typically use this style of syntax for small classes like `ListNode` that are only going to be used by one class like `LinkedIntList`.

Activity:

We have the same `printList` method from Wednesday's lesson, but have removed the parameter. This implementation is currently broken, can you fix it to make sure our list remains intact?

Extending LinkedIntList: remove(value)

In the pre-class work and section you explored numerous ways to manipulate a collection of ListNodes linked from a single "front" pointer. Some key things to remember:

- When looping over a LinkedList, create a temporary pointer that you will use to examine your list "safely" - i.e. without destroying the list. We will call this pointer `curr` for "current", you can think of `curr` as similar to `i` in a `for loop` representing the current iteration of the loop or your "current" place in the list.
- To loop over a list we typically use `while loops` where we test if the loop pointer or "`curr`"

  ```
  while (curr != null) // loop to end of list
  ```

  ```
  while (curr.next != null) // stop one early to make edits
  ```

- LinkedLists can be tricky because the code to manipulate the first, middle and last nodes can be different, so always be sure to check if your code handles an empty list (`front == null`), a list with just one node (`front.next == null`), manipulating the first node, a node in the middle and the very last node.

Activity

Write a new method inside the `LinkedIntList` class `remove(int value)` that removes the `ListNode` at the given index. If the value given doesn't exist the list should be unchanged. For example, consider the below list `myList` before and after a call to `remove(-3)` .

Hint: don't forget to update the `size` field.

```
myList = front -> 16 -> -3 -> 27
myList.remove(-3);
myList = front -> 16 -> 27
```