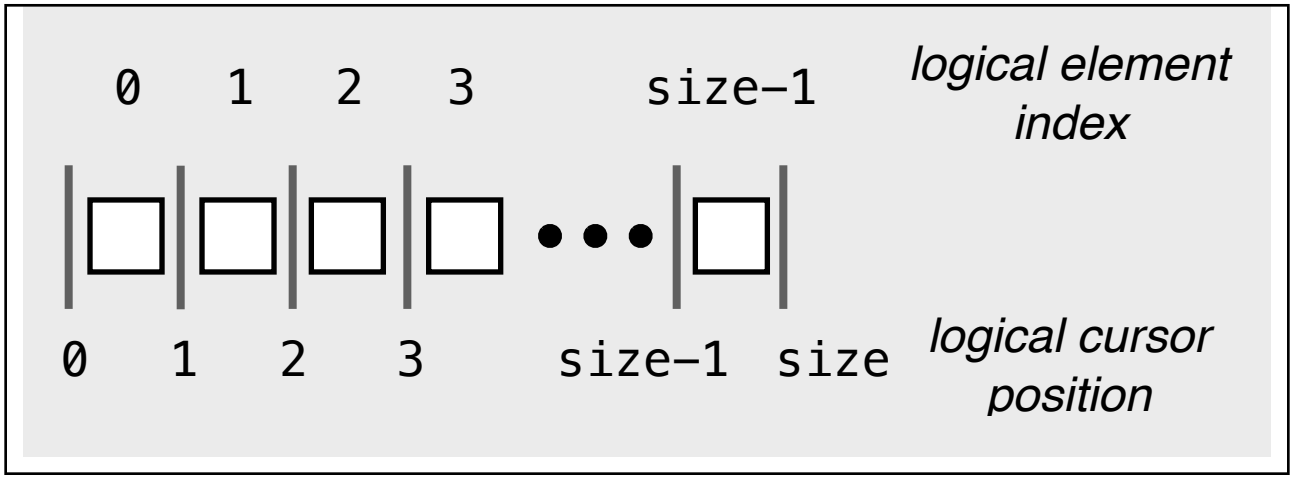
CS 228: Introduction to Data Structures
Lecture 20

The `ListIterator` Interface

The `ListIterator` interface extends the `Iterator` interface, taking advantage of the fact that elements are linearly ordered.

The internal state of a `ListIterator` includes a ***logical cursor position***. It is helpful to think of it as a kind of text cursor – a vertical line occupying the "gap" between elements.



We instantiate iterators using one of two methods.

`ListIterator<E> listIterator()`: Returns an iterator with logical cursor position `0`.

`ListIterator<E> listIterator(int pos)`: Returns an iterator with logical cursor position `pos`.

We can move a `ListIterator`, it backwards and forwards through a `List` object. That is, in addition to the `hasNext()` and `next()` methods of the `Iterator` class, a `ListIterator` also possesses `hasPrevious()` and `previous()` methods. We will study the semantics of these and the other key `ListIterator` methods — `add()`, `set()`, and `remove()` — next time.

The `ListIterator` Methods

We can query a `ListIterator` for its current position using:

`int nextIndex()`: Returns the index of the element after the cursor; returns `size()` when at the end.

and

`int previousIndex()`: Returns the index of the element that would be returned by a subsequent call to `previous()`. I.e., it returns the index of the element before the cursor; and returns −1 when at beginning.

As usual, we have the following methods:

`boolean hasNext()`: `true` when `nextIndex() < size()`.

`E next()`: Returns the element after the cursor and moves cursor forward.

There are also methods for iterating in reverse:

`boolean hasPrevious()`: `true` when `previousIndex() > 0`.

`E previous()`: Returns the element before the cursor, moves cursor backward.

`next()` and `previous()` may throw `NoSuchElementException`.

Notation: Assume we have a `list` of `Strings` “A", "B", "C", "D", "E" and an iterator `iter` created by `listIterator()`. We'll use a vertical bar `|` to denote cursor position. Thus,

```
Initially:          |ABCDE
iter.next();        A|BCDE
iter.next();        AB|CDE
```

The remaining three methods of `ListIterator` are

```
void remove()
void add (E item)
void set (E item)
```

These methods potentially modify the list, based on the cursor position.

`remove()`

The `remove()` method deletes the element behind the cursor or the element ahead of the cursor, depending on whether `next()` or `previous()` was called. Formally:

> `void remove()`: Removes from the list the last element that was returned by `next()` or `previous()`. This call can only be made once per call to `next()` or `previous()`. It can be made only if `ListIterator.add` has not been called after the last call to `next()` or `previous()`.

Example 1.

```
Initially:          |ABCDE
iter.next();        A|BCDE
iter.next();        AB|CDE (*)
iter.remove();      A|CDE
```

Note that by definition of `remove()`, after the last line of Example 1, we have `iter.nextIndex() == 1`.

Example 2.

```
Initially:            |ABCDE
iter.next();          A|BCDE
iter.next();          AB|CDE
iter.next();          ABC|DE
iter.previous();      AB|CDE (**)
iter.remove()         AB|DE
```

Although line (*) of Example 1 ***looks*** identical to line (**) of Example 2, the results of doing `remove()` immediately after are different. The reason is that, by definition, `remove()` deletes the last element that was returned by `next` or `previous`, so

- calling `remove()` after `next()` deletes the element ***behind*** the cursor and ***moves*** the cursor ***back***, and

- calling `remove()` after `previous()` deletes the element ***in front of*** the cursor and ***does not*** move the cursor.

This means that the iterator must not only keep track of its position, but also of which element is pending removal.

`add()`

The `add()` method puts the new element behind the cursor and advances the cursor. Formally:

`void add(E item)`: Inserts `item` into the list.

- The element is inserted immediately before the next element that would be returned by `next()`, if any, and after the next element that would be returned by `previous()`, if any.
- If the list contains no elements, the new element becomes the sole element on the list.
- The new element is inserted ***before*** the implicit cursor: a subsequent call to `next()` would be unaffected, and a subsequent call to `previous()` would return the new element.
- Increases by one the value returned by a call to `nextIndex()` or `previousIndex()`.

Example 3.

```
Initially:              |ABCDE
iter.add("X")           X|ABCDE
iter.add("Y")           XY|ABCDE
```

As next two examples show, `add()` always inserts ***behind*** the cursor, regardless of whether `previous` or `next` was called.

Example 4.

```
Initially:              |ABCDE
iter.next();            A|BCDE
iter.add("X")           AX|BCDE
```

Example 5.

```
Initially:              |ABCDE
iter.next();            A|BCDE
iter.next();            AB|CDE
iter.previous();        A|BCDE
iter.add("X")           AX|BCDE
```

Example 6. To add an item at the end of the list, we simply have to place a new iterator there using `listIterator(list.size())`, and then `add` the item.

```
Initially:          ABCDE|
iter.add("X");      ABCDEX|
iter.add("Y");      ABCDEXY|
```

`set()`

The `set()` method behaves like `remove()`, modifying the element behind the cursor or the element ahead of the cursor, depending on whether `next()` or `previous()` was called.

`void set(E item)`: Replaces the last element returned by `next()` or `previous()` with `item`. This call can be made only if neither `remove()` nor `add()` have been called after the last call to `next()` or `previous()`.

Example 7.

```
Initially:            |ABCDE
iter.next();          A|BCDE
iter.next();          AB|CDE (*)
iter.set("X");        AX|CDE
```

Example 8.

```
Initially:            |ABCDE
iter.next();          A|BCDE
iter.next();          AB|CDE
iter.next();          ABC|DE
iter.previous();      AB|CDE (**)
iter.set("X")         AB|XDE
```

Notice that, although lines (*) and (**) of Examples 7 and 8, respectively, are identical, the results of doing `set()` immediately after are different.

Like `remove()`, we cannot call `set()` without a prior to call to `previous()` or `next()`. However, there is nothing wrong with calling `set()` twice in a row: It just updates the same element.

Example 9.

```
Initially:          |ABCDE
iter.next();        A|BCDE
iter.set(“X”)       X|BCDE
iter.set(“Y”)       Y|BCDE
```