CS 228: Introduction to Data Structures
Lecture 19
Monday, March 2, 2015

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 1.

```
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 2.

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

Example 3.

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

Example 4. 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 1.

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

Example 2.

```
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, as for `remove()`, although lines (*) and (**) are identical in Examples 1 and 2, the results of doing `set()` immediately after are different.

Like `remove()`, we can't 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.

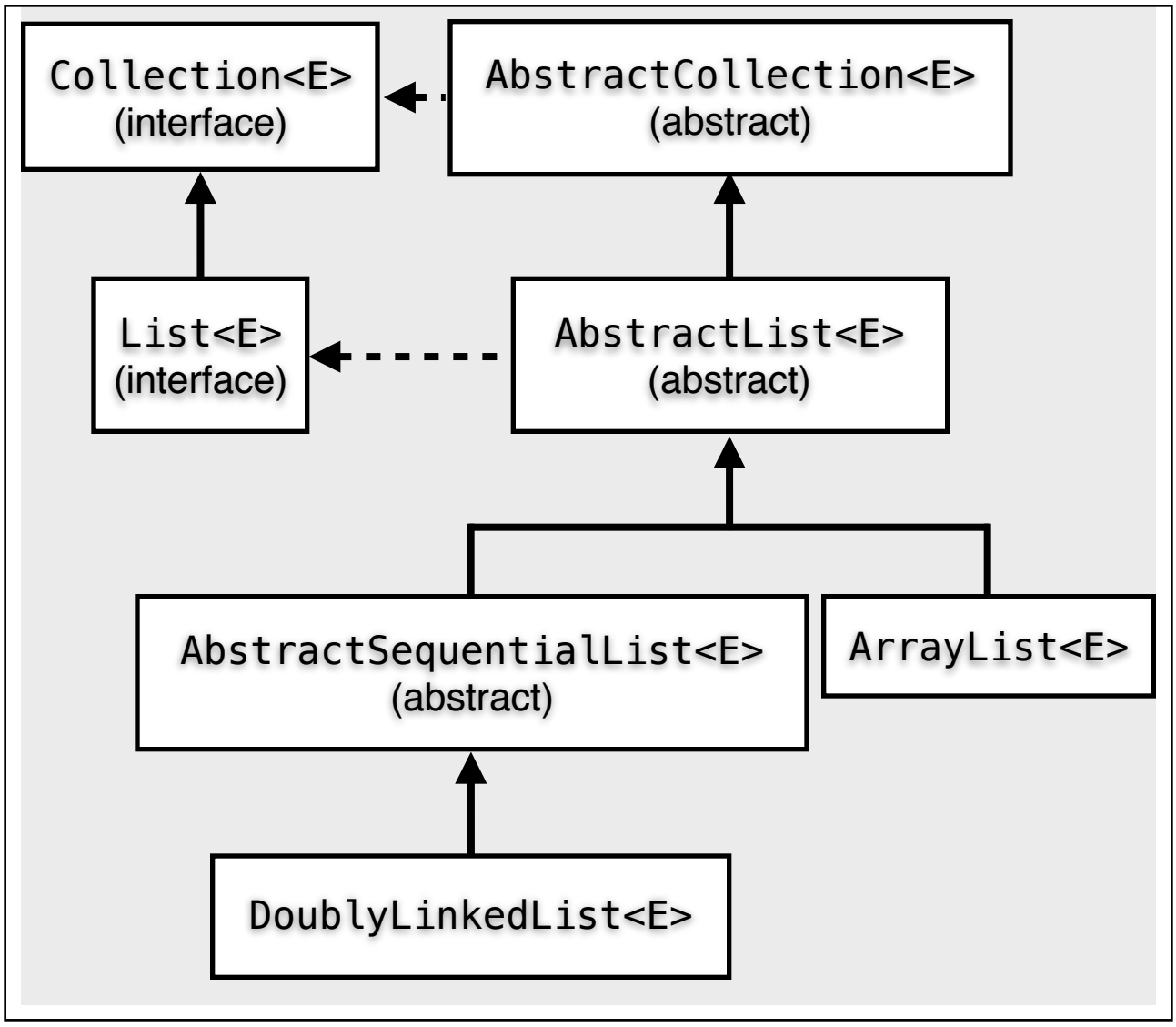
Example 3.

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

The `AbstractSequentialList` Class

Implementing the full `List` interface would be overwhelming. Instead, we build upon the existing abstract class `AbstractSequentialList`, where all methods other than `size()` and `listIterator(pos)` are optional. The class hierarchy is on the next page.

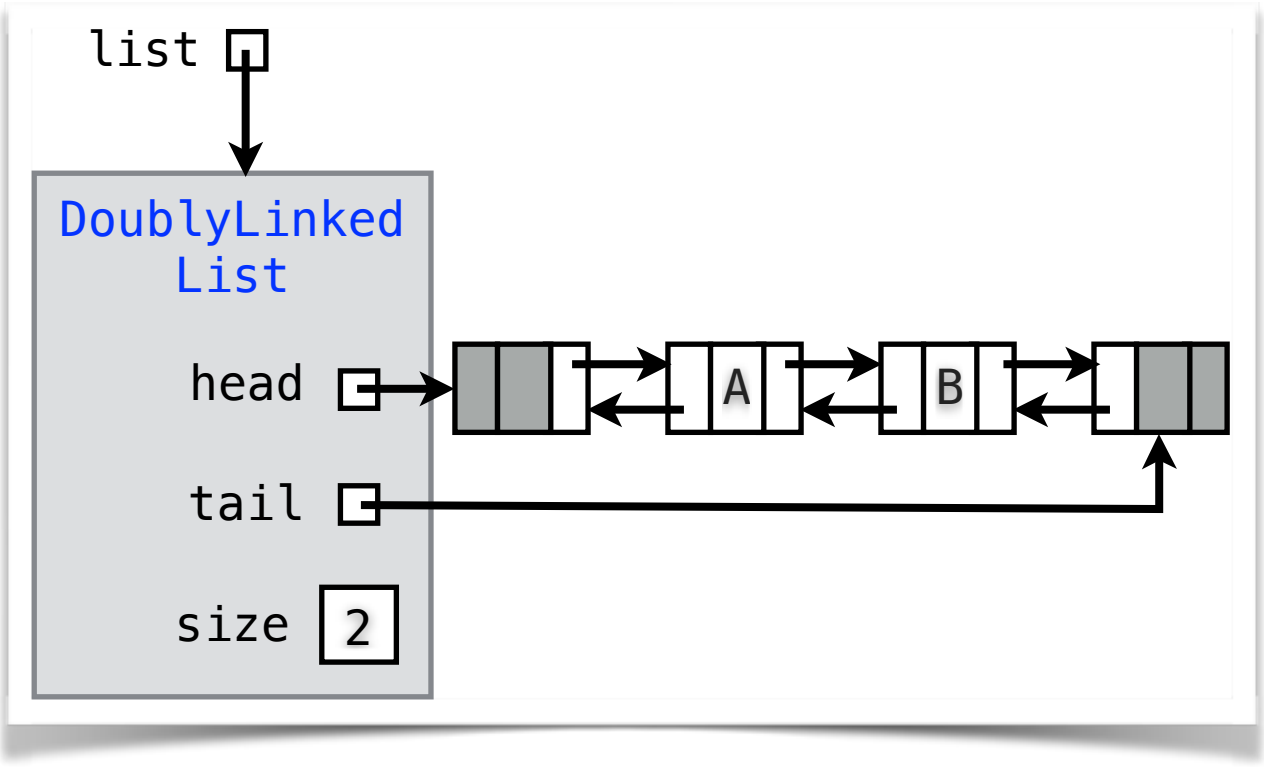
Note that `AbstractSequentialList` assumes that the the list is represented via a sequential access data structure. Contrast this with `ArrayList`, which assumes an array as the backing store.


Collection<E>
(interface)
AbstractCollection<E>
(abstract)
List<E>
(interface)
AbstractList<E>
(abstract)
AbstractSequentialList<E>
(abstract)
ArrayList<E>
DoublyLinkedList<E>

A Doubly-Linked `List` Implementation

Singly-linked lists are a poor choice for implementing the `List` interface, because it isn't easy to iterate backwards on them. Therefore, we focus instead on a doubly-linked list implementation of the `List` interface; it is called `DoublyLinkedList`.

Our lists will have ***dummy nodes*** at `head` and `tail`. Here is a `DoublyLinkedList` object, called `list`, that stores two strings.



The class definition begins like this:

```
public class DoublyLinkedList<E>
extends AbstractSequentialList<E> {

   private Node head;

   private Node tail;

   private int size;
```

We use the same `Node` class as in `DoublyLinkedCollection`. The constructor is:

```
  public DoublyLinkedList()
  {
    head = new Node(null);
    tail = new Node(null);
    head.next = tail;
    tail.previous = head;
    size = 0;
  }
```

We will need a few helper methods. The first of them splices a given node at a specified position in a list.

`void link(Node current, Node newNode)`:
Inserts newNode into this list after `current` without updating `size`.

Precondition: `current != null`, `newNode != null`

```
private void
link(Node current, Node newNode)
{
  newNode.previous = current;
  newNode.next = current.next;
  current.next.previous = newNode;
  current.next = newNode;
}
```