CS 228: Introduction to Data Structures
Lecture 19
Monday, October 10, 2016

`Collection` : A Doubly-Linked Implementation

The class `DoublyLinkedCollection`, posted on Blackboard, implements the `Collection` interface using a doubly-linked list as the backing store. As in `FirstCollection`, is `AbstractCollection`.

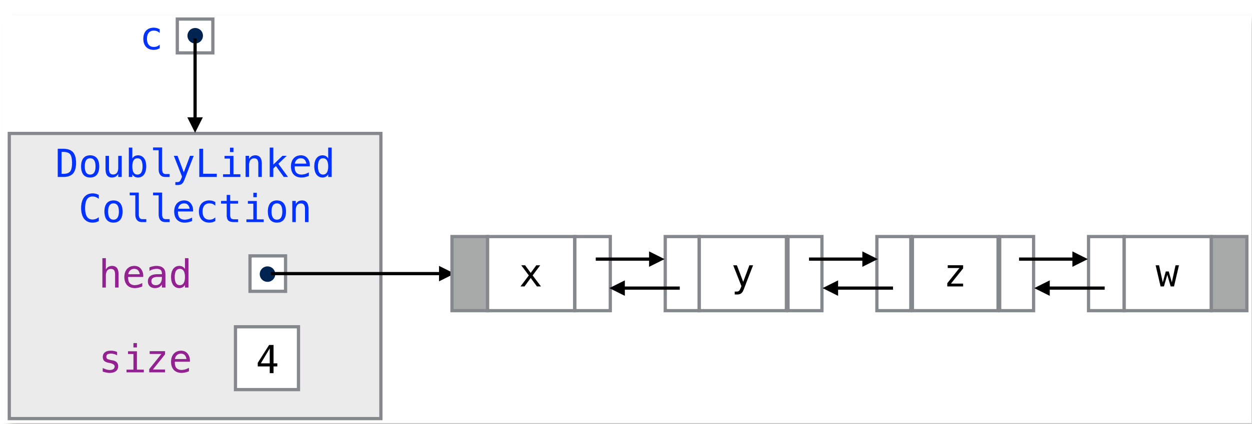
```
public class DoublyLinkedCollection<E>
  extends AbstractCollection<E>
{
  private Node head = null;
  private int size = 0;
```

Nodes are implemented by an inner class `Node` defined within `DoublyLinkedCollection`. Thus, `Node` has access to the type parameter E.

```
private class Node
{
  public E data;
  public Node next;
  public Node previous;

  public Node(E data,
              Node next, Node previous)
  {
    this.data = data;
    this.next = next;
    this.previous = previous;
  }
}
```

Note that all fields of `Node` are public, so they are visible within the definition of `DoublyLinkedCollection`, although they are not visible outside it. A collection now looks like this:



The size() method

This method is trivial:

```
@Override
public int size()
{
  return size;
}
```

The add() method

Since a collection object is not required to maintain the elements in any particular order, we will just put each new element at the beginning of the chain. This saves us from having to locate or keep track of the end of the chain.

```
@Override
 public boolean add(E item)
 {
   // add at beginning
   Node temp = new Node(item, head, null);

   // special case for empty or nonempty
   // list
   if (head != null)
   {
     head.previous = temp;
   }

   head = temp;
   ++size;
   return true;
 }
```

Iterators for `DoublyLinkedCollection`

We implement iterators through an inner class called `LinkedIterator`. This analogous to what we did for array-based collections. The `iterator()` method is then implemented as follows.

```
@Override
 public Iterator<E> iterator()
 {
   return new LinkedIterator();
 }
```

The code for the inner class `LinkedIterator` begins like this:

```
 private class LinkedIterator
 implements Iterator<E>{

   private Node cursor = head;
   private Node pending = null;

   @Override
   public boolean hasNext()
   {
     return cursor != null;
   }
```

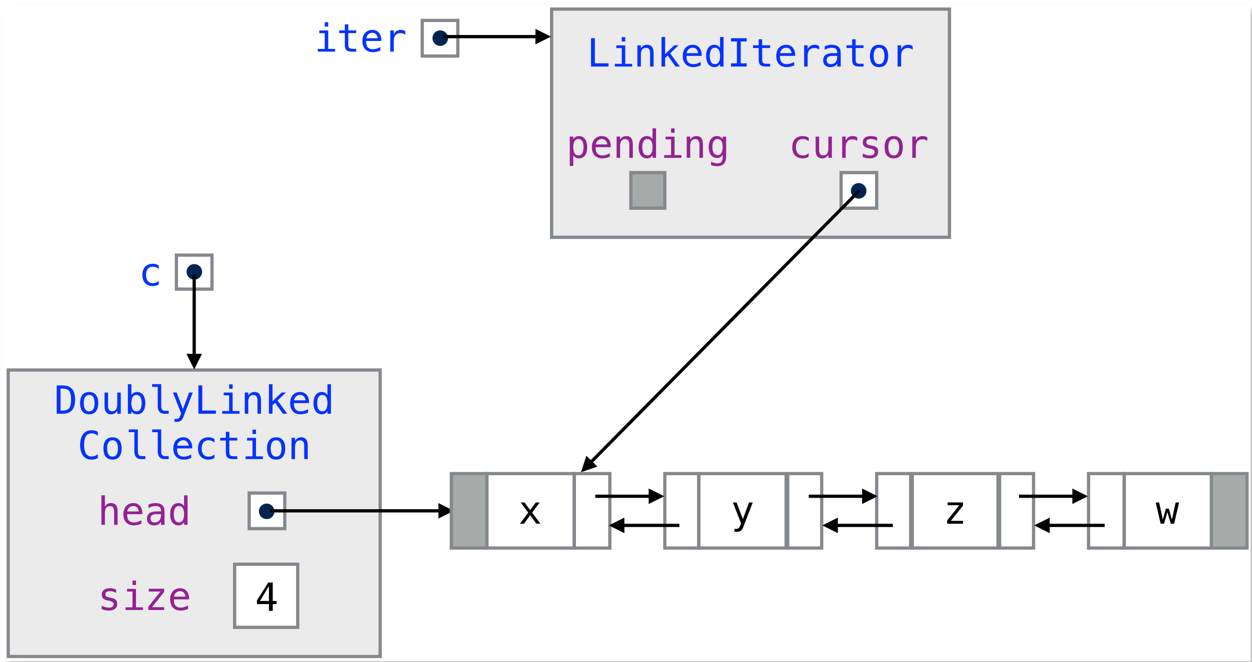
Note that, since `LinkedIterator` is defined within `DoublyLinkedCollection`, the former has access to the latter's instance variables, head and `size`. `LinkedIterator` has two Node fields. The first is `cursor`, which references the next node to be examined. `cursor` is initialized to head (so that it references the first item in the list). `hasNext()` is `true` if and only if `cursor != null`.

The other Node field is `pending`, which is initialized to `null`, and is used as follows:

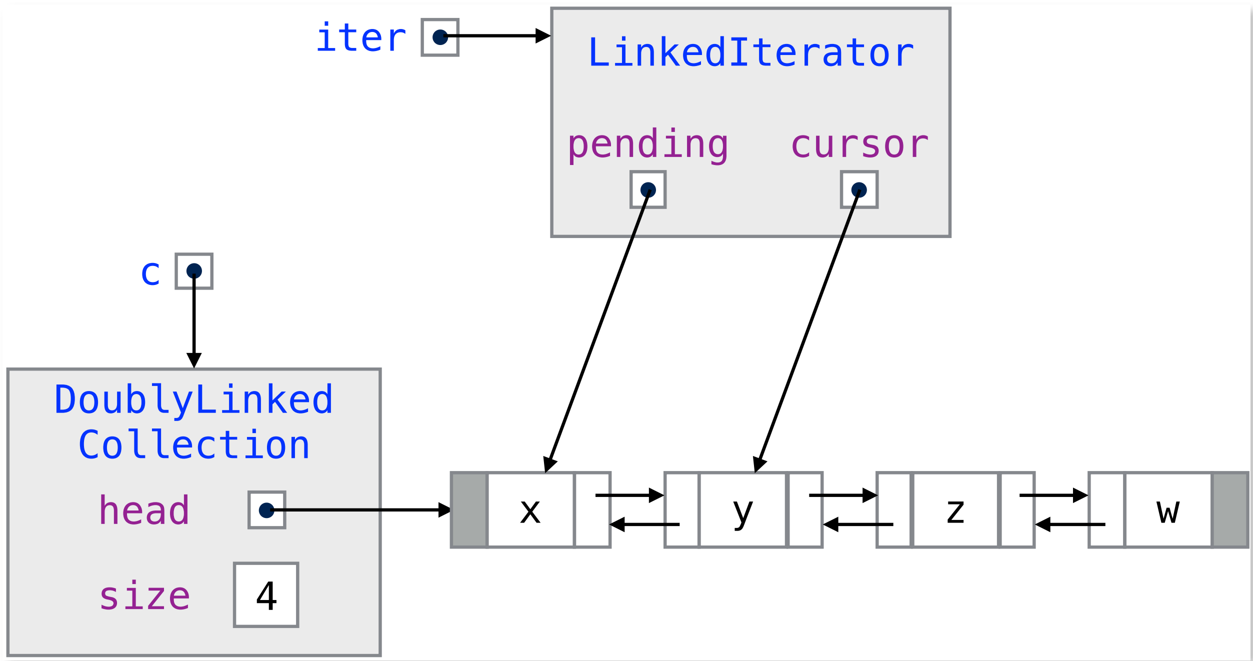
- If `pending == null`, then this indicates that there was no prior `next()`, so `remove()` is not allowed.
- If `pending != null`, then `pending` references the node whose removal is “pending” after the previous `next()`.

The next figure shows what happens after we execute.

```
Iterator<String> iter = c.iterator();
```



Every time `next()` is called, `pending` is updated to reference the *predecessor* of the node that `cursor` references. The figure on the next page shows what happens after we invoke `iter.next()`.



Now, `pending` references the node that will be deleted by the next `remove()`. After a `remove()` operation, `pending` is reset to `null`, indicating that a `remove()` is not allowed (until another `next()` is performed).

The code for the `next()` method is as follows.

```
public E next()
{
  if (!hasNext())
    throw new NoSuchElementException();

  pending = cursor;
  cursor = cursor.next;
  return pending.data;
}
```

Note that, since `LinkedIterator` is defined within `DoublyLinkedCollection`, we can refer to the type variable E. We used the same idea in `FirstCollection`.

Implementing the Iterator `remove()` Method

Let us summarize what we require of `LinkedIterator`'s two instance variables, `cursor` and `pending`.

Class invariants:

1. `cursor` points to the next element to be returned by next(). `cursor` is `null` if the list is empty or there are no more elements.

2. `pending` points to the element just returned by `next()`; a `null` value indicates that `remove()` may not be called

We have already seen how to implement `hasNext()` and `next()`. Here is the implementation of `remove()`.

```java
@Override
public void remove()
{
  if (pending == null)
    throw new IllegalStateException();

  // unlink pending node
  if (pending.previous != null)
  {
    pending.previous.next
      = pending.next;
  }
  if (pending.next != null)
  {
    pending.next.previous
      = pending.previous;
  }

  // if we're deleting the head, update
  // head reference
  if (pending == head)
  {
    head = pending.next;
  }

  --size;
  pending = null;
}
```

Note. The approach we just saw is not the only way to implement iterators. We could have instead used a boolean `canRemove` flag — as we did in `FirstCollection` — to indicate whether we are in a state where it is legal to call `remove()`, and a *single* pointer into the list (rather than two). A single pointer suffices for deletion, since nodes in a doubly-linked list have references to their successors and predecessors. To make this idea work, though, we have to be careful about the boundary cases (e.g., what if the iterator is at the beginning or end of the list?). We leave the details as an exercise.

Note 2. An example of how to use a `DoublyLinkedCollection` is posted on Blackboard.

Singly-Linked Lists

The supplement to these notes describes an implementation of the `Collection` interface based on null-terminated, singly-linked lists without dummy nodes. This singly-linked implementation is in some ways more complex than the doubly-linked one, because to remove a node in a linked list, we need a reference to the ***predecessor*** of that node. Singly-linked structures are better suited for collections where the access policy is restricted, such as stacks and queues.

The `List` Interface

A ***list*** is a linearly ordered collection with random access to the elements. The `List` interface extends the `Collection` interface, to which it adds:

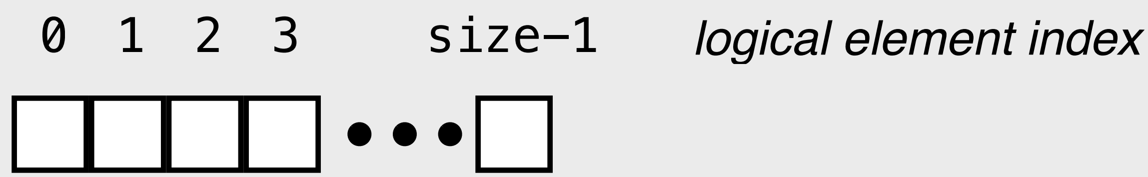
- Methods to access elements by index, including
  - `void add(int k, E item)`: Add `item` at index k.
  - `E get(int k)`: Return the item at index k.
  - `E set(int k, E element)`: Set item k to `element`.
  - `E remove(int k)`: Remove the item at index k.

  Note. As with arrays, the index is `0`-based.

- A `listIterator()` method that returns a `ListIterator` object, which can start iteration at any point in the list and move backward or forward.

The semantics of the `ListIterator` methods is complex, so we leave them for later, and focus instead on the index-based methods.

It helps to visualize a `List` object as a sequence of boxes (each of which references an element of the list), numbered with a ***logical index*** or position:



When you add an element at a given position `i`, the new item will have logical index `i`, and the logical index is increased by 1 for all items to the right.

Examples.

```
Initially:              A B C D E
add(2, X)   ==>         A B X C D E

Initially:              A B C D E
set(4, Y)   ==>         A B C D Y (returns E)

Initially:              A B C D E
remove(1)   ==>         A C D E   (returns B)
```

```
Initially:                 A B C D E
remove(size() - 1) ==> A B C D (returns E)

Initially:                 A B C D E
add(5, Z)  ==>             A B C D E Z
```

For most methods, using an index outside the range `0` to `size()–1` (inclusive) results in an `IndexOutOfBoundsException`. However, `add()` is different, because it makes sense to add an element or a collection of elements at the end of the list, which is position `size()`.