CS 228: Introduction to Data Structures
Lecture 18
Friday, October 7, 2016

An iterator for `FirstCollection`

Recall that the `Iterator<E>` interface has three methods:

```
boolean hasNext()
E next()
void remove()
```

We have already explained the definition of `next()` and `hasNext()` in the `Collection` interface. The specification of `remove()` is as follows.

> **`void remove()`**
> Removes the element returned by the last call to `next()`. Once an element has been removed, `remove()` cannot be called again until another call to `next()` has been made.

If `remove()` is invoked at an illegal or inappropriate time — i.e., before another call to `next()` — then an `IllegalStateException` should be thrown: We are violating the class contract by invoking the method when the object is not in the right state.

We are now ready to describe the concrete implementation of iterators for `FirstCollection` (refer to the code in Blackboard). The details of the iterator implementation are hidden from the clients in a private ***inner class*** within `FirstCollection` called `MyIterator`:

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

Placing `MyIterator` within `FirstCollection` gives it access to the `data` array and the `size` field. `MyIterator` centers around a `cursor` variable, which marks the current position (state) of the iterator.

- `cursor` is initialized to 0.

- `next()` returns the item in position `cursor` of `data` and *then* increments `cursor`. It also sets `canRemove` to `true` to indicate that `remove()` is now allowed.

- `hasNext()` is `true` if `cursor < size`.
- `remove()` must remove the element just before the cursor, because that's the one that was returned by the previous call to `next()`. To ensure that `remove()` is not called before `next()`, `FirstCollection` maintains a state variable **`canRemove`**, which is only true if `next()` has been invoked. Then, `remove()`
  - shifts elements beyond `cursor` down by one and decrements `size`,
  - decrements `cursor`, so that the subsequent call to `next()` is handled correctly, and
  - sets `canRemove` to `false` to disallow another deletion until `next()` is invoked again.

The details follow.

```java
private class MyIterator implements
  Iterator<E>
{
  // index of the next element to be
  // returned by next()
  private int cursor = 0;
  private boolean canRemove = false;

  @Override
  public boolean hasNext()
  {
    return cursor < size;
  }

  @Override
  public E next()
  {
    if (cursor >= size)
      throw new NoSuchElementException();
    canRemove = true;
    return data[cursor++];
  }
```

```
    @Override
    public void remove()
    {
      if (!canRemove)
      {
        throw new IllegalStateException();
      }

      // delete element before cursor.
      // Note that must have cursor >= 1
      for (int i = cursor; i < size; ++i)
      {
        data[i - 1] = data[i];
      }

      // null out the vacated cell to avoid
      // memory leak
      data[size - 1] = null;

      --size;
      --cursor;
      canRemove = false;
    }
  }
}
```

Linked Lists

While the array implementation of `Collection` is simple and adequate for many purposes, it also has some disadvantages.

- A certain amount of memory is required for the array, even if the collection contains very few elements.
- Removing an element requires all elements to the right to be shifted, making it an O(n) operation.

Linked lists avoid these issues. A linked list consists of ***linked nodes***. Each ***node*** is a simple container, holding some piece of data, with references (links) to one or more other nodes. Linked lists play a central role in data structures, and they come in many varieties. You can have just forward links, backward and forward links, “dummy” nodes, multiple successors (which gives you a ***tree***), circular links, etc.

Singly-Linked Lists

In a ***singly-linked list***, each node has a reference to the next node in the list. For now, let us ignore the issue of generic types. Then, we can define the type `Node` as:

```
public class NodeDemo
{
  private class Node
  {
    public Object data;
    public Node next;

    public Node(Object data)
    {
      this.data = data;
    }
  }
. . . }
```

Since this will be a private inner class within some collection, we will follow common practice and not bother with accessor methods. Observe that even though the fields are public, they will not be accessible outside the enclosing class.

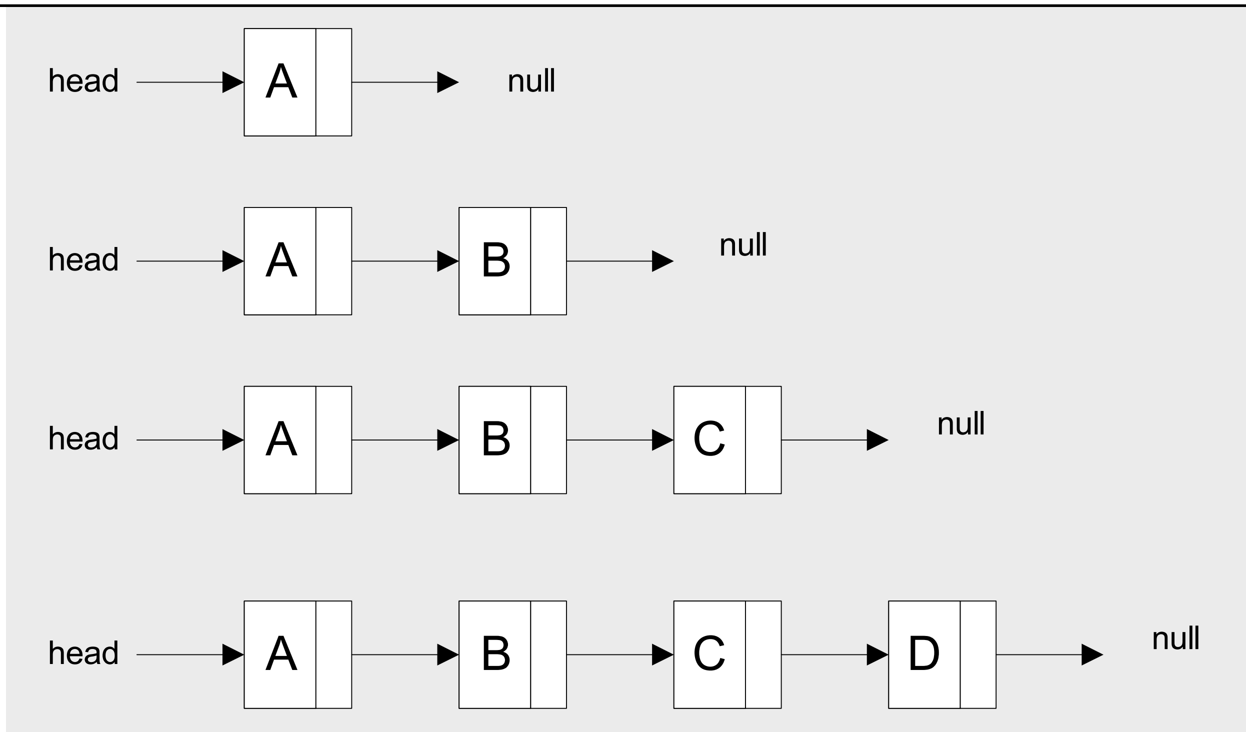
We can build a list like this[1]:

```
Node head = new Node("A");
head.next = new Node("B");
head.next.next = new Node("C");
```

[1] Aside from this example, we'll ***never*** use this coding style to build linked lists. In fact, if we're going to add frequently to the end of the list, it is more efficient and cleaner to maintain a reference to the last node (the "tail") of the list. We'll see examples of this later.

```
head.next.next.next = new Node("D");
```

Here is the result.



This data structure is called a ***null-terminated singly-linked list***.

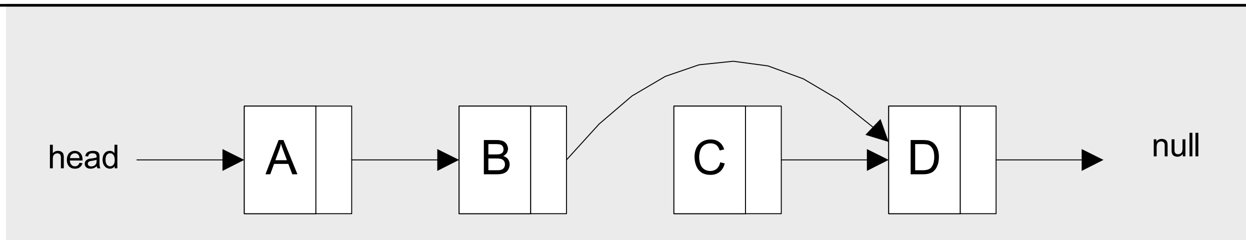
We can access any element by starting at head:

```
System.out.println(head.data);
System.out.println(head.next.data);
System.out.println(head.next.next.data);
System.out.println
  (head.next.next.next.data);
```

Now, suppose we do this:

```
head.next.next = head.next.next.next;
```

The result is:



This effectively removes the node containing “C” from the list. Strictly speaking, what happens is that, since C is no longer referenced, it becomes “garbage,” which is eventually reclaimed by Java’s garbage collector.

Exercise: What happens if we do `head = null`?

We can loop through the list using a temporary variable:

```
Node current = head;
while (current != null)
{
  System.out.println(current.data);
  current = current.next;
}
```

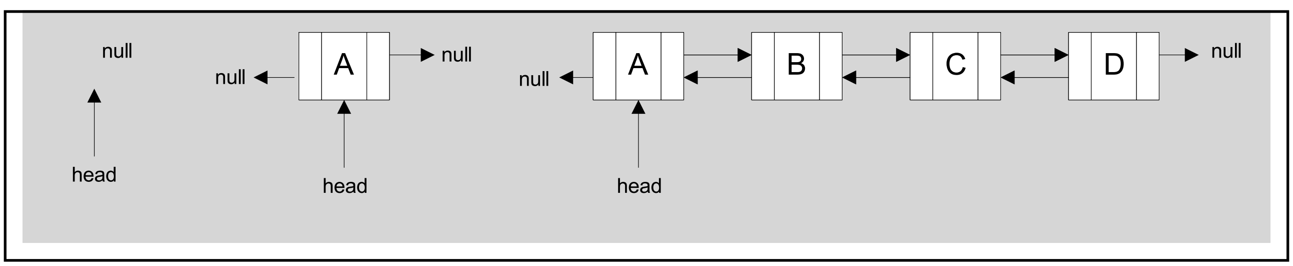
A major limitation of singly-linked lists is that we cannot quickly access the ***predecessor*** of the current element, making it difficult to delete this element. It also means that we can only iterate in one direction. Bi-directional iteration can be quite useful; indeed, it is essential for implementing the `List` and `ListIterator` APIs that we will study soon.

Implementing `Collection` with Doubly-Linked Lists

In ***doubly-linked lists***, nodes have backward links as well as forward links, making bi-directional iteration possible. Doubly linked lists are actually easier to manipulate than singly-linked ones, at the cost of a small amount of memory. We'll see how to use doubly-linked lists to implement the `Collection` class. The sample code — `DoublyLinkedCollection.java` — is posted on Blackboard. Note that in class we will only focus on the main methods. You are responsible for carefully reading *all* the code.

Basic Structure

Nodes in a doubly-linked list have references to ***two*** other nodes. Here's what an empty, a one element, and a four-element doubly-linked list look like. As before, the list is accessed through a head pointer.



We define nodes using an inner class within the collection. Thus, it has access to the type parameter E.

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

  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 head is an instance variable. A list has size zero if head `== null`.