CS 228: Introduction to Data Structures
Lecture 20
Wednesday, March 4, 2015

`List`: A Doubly-Linked Implementation (contd.)

Two more helper methods that will prove useful are the following.

`void unlink(Node current)`: Removes `current` from the list without updating `size`.

Precondition: `current != null`

`findNodeByIndex(int pos)`: Returns the `Node` whose index is `pos`, which will be `head` if `pos == −1` and `tail` if `pos == size`.

Precondition: `size >= pos >= −1`.

```
private void unlink(Node current)
{
  current.previous.next = current.next;
  current.next.previous
    = current.previous;
}

private Node findNodeByIndex(int pos)
{
  if (pos == -1) return head;
  if (pos == size) return tail;

  Node current = head.next;
  int count = 0;
  while (count < pos)
  {
    current = current.next;
    ++count;
  }
  return current;
}
```

Time complexities of the helper methods. It is not hard to see that `link()` and `unlink()` are O(1)-time operations, while `findNodeByIndex()` takes O(n) time in the worst case, where n is the length of the list.

`add()`. We have two options when adding an element. The first just adds a new item at the end of the list.

```
public boolean add(E item)
{
  Node temp = new Node(item);
  link(tail.previous, temp);
  ++size;
  return true;
}
```

The second adds the item at a specific position.

```
public void add(int pos, E item)
{
  if (pos < 0 || pos > size)
    throw new IndexOutOfBoundsException
              ("" + pos);

  Node temp = new Node(item);
  Node predecessor =
      findNodeByIndex(pos - 1);
  link (predecessor, temp);
  ++size;
}
```

Time complexities of the add() methods. The `add(item)` method takes O(1) time, since we have direct access to the end of the list. On the other hand, `add(pos,item)` takes O(n) time in the worst case — where, as usual n is the length of the list — since we have to traverse the list (using `findNodeByIndex`) to locate the insertion point.

Note. The code posted on Blackboard also has implementations of `get()` and `contains()` — study that code carefully.

List Iterators

As usual, we implement iterators with an inner class, here called `DoublyLinkedIterator`. We give users two options.

```
public ListIterator<E> listIterator()
{
  return new DoublyLinkedIterator();
}
```

```java
public ListIterator<E>
  listIterator(int pos)
{
  return new DoublyLinkedIterator(pos);
}
```

The class declaration begins like this:

```java
private class DoublyLinkedIterator
implements ListIterator<E>
{
  // direction for remove() and set()
  private static final int BEHIND = -1;
  private static final int AHEAD = 1;
  private static final int NONE = 0;

  private Node cursor;
  private int index;
  private int direction;
```

The following class invariants express the meanings of the instance variables.

Class Invariants

1. The logical cursor position is always between `cursor.previous` and `cursor`.
2. After a call to `next()`, `cursor.previous` refers to the node just returned
3. After a call to `previous()`, `cursor` refers to the node just returned
4. `index` is always the logical index of node pointed to by `cursor`.
5. direction is `BEHIND` if last operation was `next()`, `AHEAD` if last operation was `previous()`, `NONE` otherwise.

We need to provide two constructors.

```
    public DoublyLinkedIterator(int pos)
    {
      if (pos < 0 || pos > size)
        throw new
          IndexOutOfBoundsException
            ("" + pos);

      cursor = findNodeByIndex(pos);
      index = pos;
      direction = NONE;
    }

    public DoublyLinkedIterator()
    {
      this(0);
    }
```

`add()` inserts a new item between `previous` and `next`.

```
    public void add(E item)
    {
      Node temp = new Node(item);
      link(cursor.previous, temp);
      ++index;
      ++size;
      direction = NONE;
    }
```

This method takes O(1) time.

`hasNext()`, `hasPrevious()`, `nextIndex()`, and `previousIndex()` do the obvious. They all take O(1) time.

```
    public boolean hasNext()
    {
      return index < size;
    }

    public boolean hasPrevious()
    {
      return index > 0;
    }

    public int nextIndex()
    {
      return index;
    }

    public int previousIndex()
    {
      return index - 1;
    }
```

`next()` and `previous()` not only move `cursor` forward or backward, but must also set the direction from which the cursor is coming, `BEHIND` or `AHEAD`.

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

  E ret = cursor.data;
  cursor = cursor.next;
  ++index;
  direction = BEHIND;
  return ret;
}

public E previous()
{
  if (!hasPrevious())
    throw new NoSuchElementException();

  cursor = cursor.previous;
  --index;
  direction = AHEAD;
  return cursor.data;
}
```

Both `next()` and `previous()` take O(1) time.

`set()` and **`remove()`** need to know the direction we are coming from — `AHEAD`, `BEHIND`, or `NONE` — to determine which element to set/remove. `set()` can be called multiple times, even if the cursor has not moved. Thus, it should leave `direction` unchanged.

```
    public void set(E item)
    {
      if (direction == NONE)
      {
        throw new IllegalStateException();
      }

      if (direction == AHEAD)
      {
        cursor.data = item;
      }
      else
      {
        cursor.previous.data = item;
      }
    }
```

This method takes O(1) time.

`remove()` is slightly more complex than `set()`, in part because it must set `direction = NONE` to disallow another `remove()` immediately after. We will study `remove()` next time.