

CS 228: Introduction to Data Structures

Lecture 22

Monday, October 17, 2016

DoublyLinkedList (continued)

The ListIterator Methods

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.

The `remove()` Method

The **`remove()`** method is like `set()`, except that after it has been called it changes `direction` to `NONE`, to disallow a subsequent `remove()`, unless there is another call to `next()` or `previous()`.

```

public void remove()
{
    if (direction == NONE)
    {
        throw new IllegalStateException();
    }
    else
    {
        if (direction == AHEAD)
        {
            // remove node at cursor and move
            // to next node
            Node n = cursor.next;
            unlink(cursor);
            cursor = n;
        }
        else
        {
            // remove node behind cursor and
            // adjust index
            unlink(cursor.previous);
            --index;
        }
    }
    --size;
    direction = NONE;
}

```

The worst-case running time of this method is $O(1)$.

Array Implementation

The `ResizingArray2<E>` class, posted on Blackboard, implements the `List` interface using an array as the backing store. This is similar to what we did for `FirstCollection`, the array-based implementation of `Collection` that we saw some weeks ago. (Note that Java `ArrayLists` are array-based lists.)

The basic list `get()` and `set()` methods are $O(1)$. An `add()` to the end of the list takes $O(1)$ (amortized) time. Adding and removing elements at a specified index require shifting and are therefore $O(n)$. List iterators must keep track of their current position and the direction they are coming from. The ideas are intuitive; we ask you to study the implementation on your own. We just note two things.

- `ResizingArray2<E>` implements the `List` interface *directly*, rather than through an intermediate, predefined, abstract class, as was the case for `DoublyLinkedList`. This means that `ResizingArray2` has to implement ***all*** the methods of the `List` interface.
- In addition to array *expansion* during `add()`, `ResizingArray2` also uses array ***contraction*** during `remove()` to avoid wasting too much space in unused array slots: When the occupancy rate (number of

elements in the list divided by the total number of slots in the backing array) drops below $1/4$, the list is copied into an array of half the size.

Other Implementations

We have seen `Lists` and `ListIterators`, presented a doubly-linked list implementation with two dummy nodes, and sketched an alternative array implementation. Many other implementations are possible, for example:

- Null-terminated list with no dummy nodes or one dummy node
- Circular list with or without dummy node
- Singly-linked list, circular or null-terminated, with or without dummy nodes

While these implementations differ in specific details, they don't involve any major new ideas. Once you make the basic design decisions, writing the code is a matter of following your intuition: For each operation, draw a “before” picture of the list, then draw an “after” picture after

the operation. Now, look at how the links have to change, and write the code to make it so. Try this!

It is important to remember that, from the client's point of view, the *logical* behavior of all implementations of the `List` and `ListIterator` interfaces should be exactly the same, regardless of the underlying mechanics. Of course, the *running times* can be quite different.

Time Complexity

We now compare the running times of the various operations on linked lists and array lists. We first consider the `List` methods.

`get(i)`: *Getting an element at a given index*

- linked list: $O(n)$
 - Reason: We have to traverse the list to find the position
- array list: $O(1)$

contains(item)

- linked list: $O(n)$
- array list: $O(n)$

size()

- linked list: $O(1)$
- array list: $O(1)$

add(item): *Adding a new element at the end*

- linked list: $O(1)$
- array list: $O(1)$ (*)

(*) This is not quite right, because during a sequence of such operations, we may have to resize the array several times. In practice we don't worry about this, because we can prove that adding n elements to an array list requires time $O(n)$. That is, the ***amortized*** cost per add is $O(1)$.

add(i, item): *Adding or removing at a given position*

- linked list: $O(n)$
 - Reason: We have to traverse the list to find the position
- array list: $O(n)$
 - Reason, we have to shift elements to add/remove in the middle of the array

remove(item): *Removing a given element*

- linked list: $O(n)$
 - Reason: Have to find the element
- array list: $O(n)$
 - Reason: Have to find the element, then shift elements to remove, but $O(n) + O(n) = O(n)$

Next time, we will consider the **ListIterator** methods.