Data Abstraction
Lists and Iterators (7.1, 7.4)

Janyl Jumadinova

November 9, 2020
The java.util.List ADT

The java.util.List interface includes the following methods:

- `size()`: Returns the number of elements in the list.
- `isEmpty()`: Returns a boolean indicating whether the list is empty.
- `get(i)`: Returns the element of the list having index `i`; an error condition occurs if `i` is not in range `[0, size() – 1]`.
- `set(i, e)`: Replaces the element at index `i` with `e`, and returns the old element that was replaced; an error condition occurs if `i` is not in range `[0, size() – 1]`.
- `add(i, e)`: Inserts a new element `e` into the list so that it has index `i`, moving all subsequent elements one index later in the list; an error condition occurs if `i` is not in range `[0, size()]`.
- `remove(i)`: Removes and returns the element at index `i`, moving all subsequent elements one index earlier in the list; an error condition occurs if `i` is not in range `[0, size() – 1]`. 
Example:

A sequence of List operations:

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>List Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(0, A)</td>
<td>–</td>
<td>(A)</td>
</tr>
<tr>
<td>add(0, B)</td>
<td>–</td>
<td>(B, A)</td>
</tr>
<tr>
<td>get(1)</td>
<td>A</td>
<td>(B, A)</td>
</tr>
<tr>
<td>set(2, C)</td>
<td>“error”</td>
<td>(B, A)</td>
</tr>
<tr>
<td>add(2, C)</td>
<td>–</td>
<td>(B, A, C)</td>
</tr>
<tr>
<td>add(4, D)</td>
<td>“error”</td>
<td>(B, A, C)</td>
</tr>
<tr>
<td>remove(1)</td>
<td>A</td>
<td>(B, C)</td>
</tr>
<tr>
<td>add(1, D)</td>
<td>–</td>
<td>(B, D, C)</td>
</tr>
<tr>
<td>add(1, E)</td>
<td>–</td>
<td>(B, E, D, C)</td>
</tr>
<tr>
<td>get(4)</td>
<td>“error”</td>
<td>(B, E, D, C)</td>
</tr>
<tr>
<td>add(4, F)</td>
<td>–</td>
<td>(B, E, D, C, F)</td>
</tr>
<tr>
<td>set(2, G)</td>
<td>D</td>
<td>(B, E, G, C, F)</td>
</tr>
<tr>
<td>get(2)</td>
<td>G</td>
<td>(B, E, G, C, F)</td>
</tr>
</tbody>
</table>
Array Lists

- An obvious choice for implementing the list ADT is to use an array, A, where A[i] stores (a reference to) the element with index i.
- With a representation based on an array A, the get(i) and set(i, e) methods are easy to implement by accessing A[i] (assuming i is a legitimate index).
Insertion

- In an operation \( \text{add}(i, o) \), we need to make room for the new element by shifting forward the \( n - i \) elements \( A[i], \ldots, A[n - 1] \).

In the worst case \((i = 0)\), this takes \( O(n) \) time.
**Insertion**

- In an operation `add(i, o)`, we need to make room for the new element by shifting forward the `n - i` elements `A[i]`, ..., `A[n - 1]`.
- In the worst case (`i = 0`), this takes \( O(n) \) time.
Element Removal

In an operation \( \text{remove}(i) \), we need to fill the hole left by the removed element by shifting backward the \( n - i - 1 \) elements \( A[i + 1], \ldots, A[n - 1] \).
In an operation `remove(i)`, we need to fill the hole left by the removed element by shifting backward the \( n - i - 1 \) elements \( A[i+1], \ldots, A[n-1] \).

In the worst case (\( i = 0 \)), this takes \( O(n) \) time.
Performance

In an array-based implementation of a dynamic list:

- The space used by the data structure is $O(n)$
Performance

In an array-based implementation of a dynamic list:

- The space used by the data structure is $O(n)$
- Indexing the element at $i$ takes $O(1)$ time
Performance

In an array-based implementation of a dynamic list:

- The space used by the data structure is $O(n)$
- Indexing the element at $i$ takes $O(1)$ time
- `add` and `remove` run in $O(n)$ time
Performance

In an array-based implementation of a dynamic list:

- The space used by the data structure is $O(n)$
- Indexing the element at $i$ takes $O(1)$ time
- add and remove run in $O(n)$ time

In an add operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
An **iterator** is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

- **hasNext()**: Returns true if there is at least one additional element in the sequence, and false otherwise.
- **next()**: Returns the next element in the sequence.
Java defines a parameterized interface, named Iterable, that includes the following single method:
- `iterator()`: Returns an iterator of the elements in the collection.
Java defines a parameterized interface, named Iterable, that includes the following single method:

- iterator( ): Returns an iterator of the elements in the collection.

An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the iterator( ) method.
Java defines a parameterized interface, named `Iterable`, that includes the following single method:
- `iterator()`: Returns an iterator of the elements in the collection.

An instance of a typical collection class in Java, such as an `ArrayList`, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the `iterator()` method.

Each call to `iterator()` returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.