Lists and Iterators

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 `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.

```
\[
\begin{array}{cccccc}
  & A & 0 & 1 & 2 & i & n \\
 0 & & & & & & \\
\end{array}
\]
```

Element Removal

- 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.

```
\[
\begin{array}{cccccc}
  & A & 0 & 1 & 2 & o & n \\
 0 & & & & & i & n \\
\end{array}
\]
```
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 ...

---

Java Implementation

```java
// public methods
11  /** Returns the number of elements in the array list. */
12  public int size() { return size; }
14  /** Returns whether the array list is empty. */
15  public boolean isEmpty() { return size == 0; }
16  /** Returns (but does not remove) the element at index i. */
17  public E get(int i) throws IndexOutOfBoundsException {
18      checkIndex(i, size);
19      return data[i];
20  }
21  /** Replaces the element at index i with e, and returns the replaced element. */
22  public E set(int i, E e) throws IndexOutOfBoundsException {
23      checkIndex(i, size);
24      E temp = data[i];
25      data[i] = e;
26      return temp;
27  }
```

© 2014 Goodrich, Tamassia, Goldwasser
Java Implementation, 2

```java
public void add(int i, E e) throws IndexOutOfBoundsException, IllegalStateException {
    checkIndex(i, i + 1);
    if (size == data.length) // not enough capacity
        throw new IllegalStateException("Array is full");
    for (int k = size-1; k >= i; k--) // start by shifting rightmost
        data[k+1] = data[k];
    data[i] = e; // ready to place the new element
    size++;
}
```

Let `push(o)` be the operation that adds element `o` at the end of the list.

When the array is full, we replace the array with a larger one.

How large should the new array be?

- **Incremental strategy**: increase the size by a constant `c`
- **Doubling strategy**: double the size

Growable Array-based Array List

- Let `push(o)` be the operation that adds element `o` at the end of the list.
- When the array is full, we replace the array with a larger one.
- How large should the new array be?
  - Incremental strategy: increase the size by a constant `c`.
  - Doubling strategy: double the size.

Algorithm `push(o)`

```
if i == S.length - 1 then
    A ← new array of size ...
    for i ← 0 to n-1 do
        A[i] ← S[i]
    S ← A
    n ← n + 1
    S[n-1] ← o
```
Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of $n$ push operations.
- We assume that we start with an empty list represented by a growable array of size 1.
- We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e., $T(n)/n$.

Incremental Strategy Analysis

- Over $n$ push operations, we replace the array $k = n/c$ times, where $c$ is a constant.
- The total time $T(n)$ of a series of $n$ push operations is proportional to:
  $$n + c + 2c + 3c + 4c + \ldots + kc = n + c(1 + 2 + 3 + \ldots + k) = n + ck(k + 1)/2$$
- Since $c$ is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$.
- Thus, the amortized time of a push operation is $O(n)$. 

© 2014 Goodrich, Tamassia, Goldwasser  Lists and Iterators
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times.
- The total time $T(n)$ of a series of $n$ push operations is proportional to
  \[ n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 = 3n - 1 \]
- $T(n)$ is $O(n)$.
- The amortized time of a push operation is $O(1)$.

Positional Lists

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a positional list ADT.
- A position acts as a marker or token within the broader positional list.
- A position $p$ is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - $P$.getElement(): Return the element stored at position $p$. 
### Positional List ADT

#### Accessor methods:

- **first()**: Returns the position of the first element of $L$ (or null if empty).
- **last()**: Returns the position of the last element of $L$ (or null if empty).
- **before($p$)**: Returns the position of $L$ immediately before position $p$ (or null if $p$ is the first position).
- **after($p$)**: Returns the position of $L$ immediately after position $p$ (or null if $p$ is the last position).
- **isEmpty()**: Returns true if list $L$ does not contain any elements.
- **size()**: Returns the number of elements in list $L$.

---

### Positional List ADT, 2

#### Update methods:

- **addFirst($e$)**: Inserts a new element $e$ at the front of the list, returning the position of the new element.
- **addLast($e$)**: Inserts a new element $e$ at the back of the list, returning the position of the new element.
- **addBefore($p$, $e$)**: Inserts a new element $e$ in the list, just before position $p$, returning the position of the new element.
- **addAfter($p$, $e$)**: Inserts a new element $e$ in the list, just after position $p$, returning the position of the new element.
- **set($p$, $e$)**: Replaces the element at position $p$ with element $e$, returning the element formerly at position $p$.
- **remove($p$)**: Removes and returns the element at position $p$ in the list, invalidating the position.
Example

A sequence of Positional List operations:

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>List Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>addLast(8)</td>
<td>p</td>
<td>(8p)</td>
</tr>
<tr>
<td>first()</td>
<td>p</td>
<td>(8p)</td>
</tr>
<tr>
<td>addAfter(p, 5)</td>
<td>q</td>
<td>(8p, 5q)</td>
</tr>
<tr>
<td>before(q)</td>
<td>p</td>
<td>(8p, 5q)</td>
</tr>
<tr>
<td>addBefore(q, 3)</td>
<td>r</td>
<td>(8p, 3r, 5q)</td>
</tr>
<tr>
<td>r.getElement()</td>
<td>3</td>
<td>(8p, 3r, 5q)</td>
</tr>
<tr>
<td>after(p)</td>
<td>r</td>
<td>(8p, 3r, 5q)</td>
</tr>
<tr>
<td>before(p)</td>
<td>null</td>
<td>(8p, 3r, 5q)</td>
</tr>
<tr>
<td>addFirst(9)</td>
<td>s</td>
<td>(9s, 8p, 3r, 5q)</td>
</tr>
<tr>
<td>remove(last())</td>
<td>5</td>
<td>(9s, 8p, 3r)</td>
</tr>
<tr>
<td>set(p, 7)</td>
<td>8</td>
<td>(9s, 7p, 3r)</td>
</tr>
<tr>
<td>remove(q)</td>
<td>“error”</td>
<td>(9s, 7p, 3r)</td>
</tr>
</tbody>
</table>

Positional List Implementation

The most natural way to implement a positional list is with a doubly-linked list.
Insertion

- Insert a new node, $q$, between $p$ and its successor.

```
A  B  C
```

```
A  B  C
```

Deletion

- Remove a node, $p$, from a doubly-linked list.

```
A  B  C
```

```
A  B  C
```

© 2014 Goodrich, Tamassia, Goldwasser

Lists and Iterators
Iterators

- 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.

The Iterable Interface

- 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.
The for-each Loop

- Java’s Iterable class also plays a fundamental role in support of the “for-each” loop syntax:

  ```java
  for (ElementType variable : collection) {
    loopBody // may refer to "variable"
  }
  ```

  is equivalent to:

  ```java
  Iterator<ElementType> iter = collection.iterator();
  while (iter.hasNext()) {
    ElementType variable = iter.next();
    loopBody // may refer to "variable"
  }
  ```