CH7.
LIST AND ITERATOR ADTS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
LIST ADT

- **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 $[0, \text{size()} - 1]$.
- **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 $[0, \text{size()} - 1]$.
- **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 $[0, \text{size()}]$.
- **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 $[0, \text{size()} - 1]$.
### 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 $\text{get}(i)$ and $\text{set}(i, e)$ methods are easy to implement by accessing $A[i]$ (assuming $i$ is a legitimate index).
In an operation $\text{add}(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], ..., A[n - 1]$.

In the worst case ($i = 0$), this takes $O(n)$ time.
In an operation \texttt{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 $O(n)$
  • Indexing the element (get/set) 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 ...
EXERCISE:

• Implement the Deque ADT update functions using List functions
  • Deque update functions:
    • first(), last(), addFirst(e), addLast(e), removeFirst(), removeLast(), size(), isEmpty()
  • List functions:
    • get(i), set(i, e), add(i, e), remove(i), size(), isEmpty()
<table>
<thead>
<tr>
<th>Method</th>
<th>Array Fixed-Size or Expandable</th>
<th>List Singly or Doubly Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(i, e), remove(i)</td>
<td>$O(1)$ Best Case ($i = n$) $O(n)$ Worst Case $O(n)$ Average Case</td>
<td>?</td>
</tr>
<tr>
<td>get(i), set(i, e)</td>
<td>$O(1)$</td>
<td>?</td>
</tr>
<tr>
<td>size(), isEmpty()</td>
<td>$O(1)$</td>
<td>?</td>
</tr>
</tbody>
</table>
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 $L$. 
• 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.
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>
The most natural way to implement a positional list is with a doubly-linked list.
INSERTION, E.G., ADDAFTER (P, E)
REMOVE (p)
PERFORMANCE

• Assume doubly-linked list implementation of Positional List ADT
  • The space used by a list with $n$ elements is $O(n)$
  • The space used by each position of the list is $O(1)$
  • All the operations of the List ADT run in $O(1)$ time
## Positional List Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>List Singly-Linked</th>
<th>List Doubly-Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>first()</code>, <code>last()</code>, <code>addFirst()</code>, <code>addLast()</code>, <code>addAfter()</code></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
| `addBefore(p, e)`, `erase()`   | $O(n)$ Worst and Average case  
$O(1)$ Best case   | $O(1)$                   |
| `size()`, `isEmpty()`           | $O(1)$                   | $O(1)$                   |
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"
  }
  ```
INTERVIEW QUESTION 1

• Write code to partition a list around a value $x$, such that all nodes less than $x$ come before all nodes greater than or equal to $x$. 

INTERVIEW QUESTION 2

• Implement a function to check if a list is a palindrome.