LISTS AND ITERATORS

Abstract data types that represent a linear sequence of elements, with more general support for adding or removing elements at arbitrary positions.

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

```java
public interface List<E> {
    int size();
    boolean isEmpty();
    E get(int i) throws IndexOutOfBoundsException;
    E set(int i, E e) throws IndexOutOfBoundsException;
    void add(int i, E e) throws IndexOutOfBoundsException;
    E remove(int i) throws IndexOutOfBoundsException;
}
```

*Code Fragment 7.1: A simple version of the List interface.*
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, C)</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>
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).
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.
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 the worst case (\( i = 0 \)), this takes \( O(n) \) time.
In an array-based implementation of a list (array 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 implementation of a simple ArrayList class with bounded capacity
/** Inserts element e to be at index i, shifting all subsequent elements later. */
public void add(int i, E e) throws IndexOutOfBoundsException, IllegalArgumentException {
    checkIndex(i, size + 1);
    if (size == data.length) // not enough capacity
        throw new IllegalArgumentException("Array is full");
    for (int k=size-1; k >= i; k--)
        data[k+1] = data[k];
    data[i] = e; // ready to place the new element
    size++;
}

/** Removes/returns the element at index i, shifting subsequent elements earlier. */
public E remove(int i) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    E temp = data[i];
    for (int k=i; k < size-1; k++) // shift elements to fill hole
        data[k] = data[k+1];
    data[size-1] = null; // help garbage collection
    size--;
    return temp;
}

// utility method
/** Checks whether the given index is in the range [0, n-1]. */
protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
    if (i < 0 || i >= n)
        throw new IndexOutOfBoundsException("Illegal index: " + i);
}

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 \( t = S.length - 1 \) then

\[
A \leftarrow \text{new array of size } \ldots
\]

for \( i \leftarrow 0 \) to \( n-1 \) do

\[
A[i] \leftarrow S[i]
\]

\[
S \leftarrow A
\]

\( n \leftarrow n + 1 \)

\[
S[n-1] \leftarrow o
\]
IMPLEMENTING A DYNAMIC ARRAY

Provide means to “grow” the array A

1. Allocate a new array $B$ with larger capacity.
2. Set $B[k]=A[k]$, for $k=0, \ldots, n-1$, where $n$ denotes current number of items.
3. Set $A = B$, that is, we henceforth use the new array to support the list.
4. Insert the new element in the new array.
/** Resizes internal array to have given capacity \( \geq \) size. */
protected void resize(int capacity) {
    E[ ] temp = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
    for (int k=0; k < size; k++)
        temp[k] = data[k];
    data = temp;
    // start using the new array
}

/** Inserts element e to be at index i, shifting all subsequent elements later. */
public void add(int i, E e) throws IndexOutOfBoundsException {
    checkIndex(i, size + 1);
    if (size == data.length) // not enough capacity
        resize(2 * data.length); // so double the current capacity
    ... // rest of method unchanged...

Strategy #2: new array to have twice the capacity of the existing array
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 (amortization).

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., $T(n)/n$. 
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 $O(n)$. 
**Doubling Strategy Analysis**

- We replace the array \( k = \log_2 n \) times \((2^{k+1} - 1 = n; \text{ solve for } k)\)
- 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 \( O(1) \)

**Proposition A.12:** If \( k \geq 1 \) is an integer constant, then

\[
\sum_{i=1}^{n} i^k \text{ is } \Theta(n^{k+1}).
\]

Another common summation is the geometric sum, \( \sum_{i=0}^{n} a^i \), for any fixed real number \( 0 < a \neq 1 \).
DYNAMIC ARRAY: ANALYSIS EXAMPLE

Code Fragment 4.2: Two algorithms for composing a String with n copies of character c.

```java
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
    String answer = "";
    for (int j = 0; j < n; j++)
        answer += c;
    return answer;
}

/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
    StringBuilder sb = new StringBuilder();
    for (int j = 0; j < n; j++)
        sb.append(c);
    return sb.toString();
}
```

Figure 4.1: Chart of the results of the timing experiment from Code Fragment 4.2, displayed on a log-log scale. The divergent slopes demonstrate an order of magnitude difference in the growth of the running times.

Uses Regular Array: $O(n^2)$
Uses Dynamic Array: $O(n)$
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 $p$. 
**IMMEDIATE CHALLENGE IN DESIGNING THE ADT:**

- Challenge: Achieve constant time insertions and deletions at arbitrary locations:
  - We effectively need a reference to the node at which an element is stored.

- We introduce the concept of a *position*, which formalizes the intuitive notion of the “location” of an element relative to others in the list.

- Bad: ADT in which a node reference serves as the mechanism for describing a position.
  - Details of our implementation need to be known
  - Not a robust data structure (user can access or manipulate the nodes <= cause problems)
  - Bad encapsulating (implementation details exposed)
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).
  - **is-empty():** Returns true if list $L$ does not contain any elements.
  - **size():** Returns the number of elements in list $L$.

We can subsequently use the returned position to traverse the list:

```java
Position<String> cursor = guests.first();
while (cursor != null) {
    System.out.println(cursor.getElement());
    cursor = guests.after(cursor);
}
```
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>

position instances, we use variables such as p and q
PositionList interface

```
public interface PositionList<E> {
    /** An interface for positional lists. */
    public interface PositionList<E> {
        int size();
        boolean isEmpty();
        Position<E> first();
        Position<E> last();
        Position<E> before(Position<E> p) throws IllegalArgumentException;
        Position<E> after(Position<E> p) throws IllegalArgumentException;
        Position<E> addFirst(E e);
        Position<E> addLast(E e);
        Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException;
        Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException;
        E getElement() throws IllegalStateException;
        E set(Position<E> p, E e) throws IllegalArgumentException;
        E remove(Position<E> p) throws IllegalArgumentException;
    }
}
```

Code Fragment 7.8: The PositionalList interface.
The most natural way to implement a positional list is with a **doubly-linked list**.

- **NOTE:** Not the same as the DoublyLinkedList class in Ch3
  - Difference in the management of the positional abstraction
/** Implementation of a positional list stored as a doubly linked list. */
public class LinkedPositionalList<E> implements PositionalList<E> {
    //------------ nested Node class ------------
    private static class Node<E> implements Position<E> {
        private E element; // reference to the element stored at this node
        private Node<E> prev; // reference to the previous node in the list
        private Node<E> next; // reference to the subsequent node in the list

        public Node(E e, Node<E> p, Node<E> n) {
            element = e;
            prev = p;
            next = n;
        }

        public E getElement() throws IllegalStateException {
            if (next == null) // convention for defunct node
                throw new IllegalStateException("Position no longer valid");
            return element;
        }

        public Node<E> getPrev() {
            return prev;
        }

        public Node<E> getNext() {
            return next;
        }

        public void setElement(E e) {
            element = e;
        }

        public void setPrev(Node<E> p) {
            prev = p;
        }

        public void setNext(Node<E> n) {
            next = n;
        }
    } //------- end of nested Node class -------

The private `validate(p)` method is called anytime the user sends a `Position` instance as a parameter. It throws an exception if it determines that the position is invalid, and otherwise returns that instance, implicitly cast as a `Node`, so that methods of the `Node` class can subsequently be called.

The private `position(node)` method is used when about to return a `Position` to the user. Its primary purpose is to make sure that we do not expose either sentinel node to a caller, returning a null reference in such a case.
public accessor methods

```java
// public accessor methods
/** Returns the number of elements in the linked list. */
public int size() { return size; }

/** Tests whether the linked list is empty. */
public boolean isEmpty() { return size == 0; }

/** Returns the first Position in the linked list (or null, if empty). */
public Position<E> first() {
    return position(header.getNext());
}

/** Returns the last Position in the linked list (or null, if empty). */
public Position<E> last() {
    return position(trailer.getPrev());
}

/** Returns the Position immediately before Position p (or null, if p is first). */
public Position<E> before(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    return position(node.getPrev());
}

/** Returns the Position immediately after Position p (or null, if p is last). */
public Position<E> after(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    return position(node.getNext());
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>O(1)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>first(), last()</td>
<td>O(1)</td>
</tr>
<tr>
<td>before(p), after(p)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Insert a new node, q, between p and its successor.
91 // private utilities
92 /** Adds element e to the linked list between the given nodes. */
93 private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
94     Node<E> newest = new Node<>(e, pred, succ); // create and link a new node
95     pred.setNext(newest);
96     succ.setPrev(newest);
97     size++;
98     return newest;
99 }
100
public update methods, relying on a private `addBetween` method to unify the implementations of the various insertion operations.

<table>
<thead>
<tr>
<th></th>
<th>addFirst(e), addLast(e)</th>
<th>addBefore(p, e), addAfter(p, e)</th>
<th>set(p, e)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td><strong>O(1)</strong></td>
<td><strong>O(1)</strong></td>
<td><strong>O(1)</strong></td>
</tr>
</tbody>
</table>
Remove a node, p, from a doubly-linked list.
Public `remove` method. Note that it sets all fields of the removed node back to null—a condition we can later detect to recognize a defunct position.
The problem with using index number to keep track of an element: the index of an element e changes when other insertions or deletions occur before it.

Solution approach: Instead of storing the elements of L directly in array A, store a new kind of position object in each cell of A. A position p stores the element e as well as the current index i of that element within the list.

addFirst, addBefore, addAfter, and remove methods take $O(n)$ time
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:

```java
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.
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"
}
```

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