Data Structures

Augustin Cosse.

Spring 2021

March 25, 2021

Lists and Iterator ADTs (continued)

- An iterator is a software design pattern that abstract the process of scanning through a a sequence of elements, one element at a time
- The elements might be stored in a container class, streaming through a network or generated by a series of computations
- For all those purposes, java provides the **java.util.Iterator** interface with the following two methods

hasNext()	Returns true if there is at least one additional element in the sequence
next()	Returns the next element in thr sequence.

Lists and Iterator ADTs (continued)

- The interface uses Java's generic framework with the next method returning a parametrized element type
- As an example, the **Scanner** class (which we used with Java.io) formally implements the **Iterator**<**String**> interface, with its **next()** method returning a String instance.
- If the **next()** method of an iterator is called when nno further elements are available, a NoSuchElementException is thrown
- Of course, the hasNext() method can be used to detect that condition before calling next().

Lists and Iterator ADTs (continued)

- The combination of the two **hasNext()** and **next()** methods makes it possible to implement a loop for processing a sequence of elements

```
while(iter.hasNext()){
  String value = iter.next();
  System.out.println(value);
}
```

- The **java.util.iterator** interface contains a third method, which is optionally supported by some iterators

remove() Remove from the collection the element returned by the most recent call to **next()**. Throws IllegalStateException if next() has not yet been called or if remove() was already called since the most recent call to next.

Lists and Iterator ADTs (continued)

- A single iterator instance supports only one pass through a collection
- calls to next can be made until all elements have been reported, but there is now way to "reset" the iterator back to the beginning of the sequence
- A data structure that wishes to allow repeated iterations can support a method that returns a new iterator each time it is called
- Following this idea, Java defines another parametrized interface, named **iterable()** that includes the following method

iterator()	Returns an iterator of the elements in the collection

Lists and Iterator ADTs (continued)

- An instance of a typical collection class in Java such as the Java implementation of ArrayList is iterable
- The collection class can be used to produce an iterator as the return value of the function iterator()

```
ArrayList<Double> data;
Iterator<Double> walk = data.iterator();
while(walk.hasNext())
  if (walk.next()<0.0)
    walk.remove();
```

Lists and Iterator ADTs (continued)

- Java's iterable class also plays a fundamental role in support for the "for each" loop syntax

```
for (ElementType variable : collection){
   loopBody
}
```

- The syntax above is supported for any instance, collection of an iterable class (see the example below)

```
List<String> list = new ArrayList><();
list.add("one");
list.add("two");
list.add("three");

for( String element : list ){
    System.out.println( element.toString() );}
```

Lists and Iterator ADTs (continued)

- ElementType must be the type of object returned by its iterator and variable will take on element values within the loop body

- The two statements below can in fact be considered equivalent

```java
for (ElementType variable : collection){
  loopBody}
```

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

Lists and Iterator ADTs (continued)

- The function remove cannot be invoked within a for loop without the explicit instantiation of an iterator. I.e.

```
ArrayList<Double> data;
Iterator<Double> walk = data.iterator();
while(walk.hasNext())
  if(walk.next() < 0.0)
    walk.remove();
```

Lists and Iterator ADTs (continued)

- There are two general styles for implementing iterators
  - The snapshot iterator maintains its own private copy of the sequence of elements which is constructed at the time the iterator object is created (i.e. it records a snapshot of the sequence of elements at the time the iterator is created). This first iterator is thus unaffected by any subsequent changes to the primary collection that may occur.
  - The lazy iterator is an iterator that does not make an upfront copy but instead performs a piecewise traversal of the primary structure when the `next()` method is called to request another element.

Lists and Iterator ADTs (continued)

- There are two general styles for implementing iterators
  - The downside of this style of iterator is that it requires $O(n)$ times and $O(n)$ auxilliary space, upon construction to copy and store a collection of n elements
  - The advantage of lazy iterator is that it can typically be implemented so that the iterator requires only $O(1)$ space and $O(1)$ construction time. A downside of lazy iterators is that its behavior is affected if the primary structure is modified before the iteration completes. Many of the iterators in Java's libraries implement a "fail-fast" behavior that immediately invalidates such an iterator if its underlying collection is modified unexpectedly.

Lists and Iterator ADTs (continued)

- To have our original ArrayList class implement the Iterable¡E¿ interface, we must add an iterate() method to that class definition.
- For that purpose, we define the non static nested class ArrayIterator.
- The advantage of having the iterator as an inner class is that it can access private fields (such as the array A) that are members of the containing list

Lists and Iterator ADTs (continued)

- Each iterator maintains a field j that represents the index of the next element to be returned. it is initialized to 0, and when j reaches size(), there are no more elements to return

```
private class ArrayIterator implements Iterator<E> {
  private int j = 0; // index of next element
  private boolean removable = false;
  public boolean hasNext( ) { return j < size; }
  // size is field of outer instance
  /*@return next object
    NoSuchElementException if no more elems */
  public E next( ) throws NoSuchElementException {
    if (j == size) throw new NoSuchElementException
                              ("No next element");
    removable = true; // the element can be removed
    return data[j++]; }
// post-increment j for future call to next
```

Lists and Iterator ADTs (continued)

- Each iterator maintains a field j that represents the index of the next element to be returned. it is initialized to 0, and when j reaches size(), there are no more elements to return

```
private class ArrayIterator implements Iterator<E> {
  // continued

/* Removes elem returned by most recent call to next.*/
  public void remove( ) throws IllegalStateException {
    if (!removable) throw new IllegalStateException
               ("nothing to remove");
    ArrayList.this.remove(j-1); // was last one returned
    j--; // next element has shifted one cell left
    removable = false;
// do not allow remove again until next is called
 }}
```

Lists and Iterator ADTs (continued)

- Finally the iterator() method returns a new instance of the

```
public Iterator<E> iterator( ) {
  return new ArrayIterator( );
}
```

iterations with the LinkedPositionalList class

- When considering iterators for the PositionalList class, the first question to ask is whether we want to define the iterator on the nodes, or on the positions
- If we decide to allow the user to iterate through the positions of the list, those positions can be used to access the elements as well so support for position iterations is more general.
- It is however more standard for a container class to support iteration of the core elements so that the for-each loop syntax can be used to write code such as below

```
for (String guest : waitlist)
```

iterations with the LinkedPositionalList class

- A solution is to implement both approaches and consider the standard iterator() method (which should return an iterator for the elements of the list) and implement a positions() method (which we will choose to return an instance that is iterable instead of an iterator)
- The motivation for returning an iterable instance in the case of the position() method is to be able to use the simple syntax

```
for (Position<String> p : waitlist.positions())
```

- For such a syntax to be valid, position() should return an iterable instance.

Iterations with the LinkedPositionalList class

- To provide support for the iteration on position and nodes of LinkedPositional() we define three inner classes :
    - We first provide a PositionIterator class which provides the core functionality of the list iterations. While the ArrayList iterator maintained the index of the next element to be returned, the PositionIterator class maintains the position of the next element to be returned
    - To support iteration on the positions through the method(), and return an iterable instance, we define a second PositionIterable inner class which construct and returns a new PositionIterator each time the iterator() method is called. The position() method
    - Finally we need a top level iterator() to return an iterator on positions()

Iterations with the LinkedPositionalList class

```
//nested PositionIterator class
private class PositionIterator
              implements Iterator<Position<E>> {
  private Position<E> cursor = first( ); // next to report
  private Position<E> recent = null; // last reported
  public boolean hasNext( ) { return (cursor != null); }
  /** Returns the next position in the iterator. */
  public Position<E> next( ) throws NoSuchElementException {
    if (cursor == null) throw new
                NoSuchElementException("nothing left");
    recent = cursor;
    cursor = after(cursor);
    return recent;}
  /** Removes element returned by call to next. */
  public void remove( ) throws IllegalStateException {
    if (recent == null) throw new
             IllegalStateException("nothing to remove");
    LinkedPositionalList.this.remove(recent);
    recent = null; // don't allow remove until next is called
   }}
```

Iterations with the LinkedPositionalList class

```
private class PositionIterable implements
                                    Iterable<Position<E>> {
  public Iterator<Position<E>> iterator( ) {
       return new PositionIterator( ); }
   }
/** Returns iterable instance. */
public Iterable<Position<E>> positions( ) {
  return new PositionIterable( );
}
```

Iterations with the LinkedPositionalList class

- Finally the iterator on the list elements themselves can be obtained by adapting the PositionIterator class

```
private class ElementIterator implements Iterator<E> {
  Iterator<Position<E>> posIterator = new PositionIterator( );
  public boolean hasNext( ) { return posIterator.hasNext( ); }
  public E next( ) {
     return posIterator.next( ).getElement( );
}
  public void remove( ) { posIterator.remove( ); }
}
/** Returns an iterator of the elements stored in the list. */
public Iterator<E> iterator( ) {
    return new ElementIterator( );
}
```

The Java collections framework

- Java provides many data structures interfaces and classes which together form the Java Collections Framework
- This framework which is part of the **java.util.package** includes versions of several of the data structures discussed in this course.
- The root interface of the java collection framework is named **Collection**. This is a general interface for any data structure, such as a list, that represents a collection of elements.
- This interface is a superinterface for other interfaces in the java Collections Framework that can hold elements, such as the Deque, List, and Queue discussed in this course.
- The Collection interface includes many methods such as **size(), isEmpty(), Iterator(), ..**

The Java collections framework

- The Java collections framework includes concrete classes implementing interfaces with multiple properties

- Robust classes provide support for concurrency, allowing multiple processes to share use of a data structure in a thread safe manner.

	Interfaces			Properties			Storage	
Class	Queue	Deque	List	Capacity Limit	Thread-Safe	Blocking	Array	Linked List
ArrayBlockingQueue	✓			✓	✓	✓	✓	
LinkedBlockingQueue	✓			✓	✓	✓		✓
ConcurrentLinkedQueue	✓				✓		✓	
ArrayDeque	✓	✓					✓	
LinkedBlockingDeque	✓	✓		✓	✓	✓		✓
ConcurrentLinkedDeque	✓	✓			✓			✓
ArrayList			✓				✓	
LinkedList	✓	✓	✓					✓

The Java collections framework

- If the structure is designated as blocking, a call to retrieve an element from an empty collection waits until some other process inserts an element. Similarly, a call to insert into a full blocking structure must wait until room becomes available.

	Interfaces			**Properties**			**Storage**	
Class	Queue	Deque	List	Capacity Limit	Thread-Safe	Blocking	Array	Linked List
ArrayBlockingQueue	✓			✓	✓	✓	✓	
LinkedBlockingQueue	✓			✓	✓	✓		✓
ConcurrentLinkedQueue	✓				✓		✓	
ArrayDeque	✓	✓					✓	
LinkedBlockingDeque	✓	✓		✓	✓	✓		✓
ConcurrentLinkedDeque	✓	✓			✓			✓
ArrayList			✓				✓	
LinkedList	✓	✓	✓					✓

Liste iterators in Java

- The **java.util.LinkedList** class does not expose a position concept to users in its API as we do in our implementation of the PositioinalList ADT.
- Instead the preferred way to access and update a LinkedList object in Java, without using indices, is to use a ListIterator that is returned by the list's listiterator() method. Such an iterator provides forward and backward traversal methods as well as local update methods.
- It views his positions as being before the first element, after the last element or between two elements
- That is it uses a list cursor

Liste iterators in Java

- The **java.util.ListIterator** interface includes the following methods

add(e)	Adds the element e at the current position of the iterator
hasNext()	Returns true if there is an element after the current position of the iterator
hasPrevious()	Returns true if there is an element before the current position
previous	return element e before current position and sets current position to be before e
next()	Returns the element e after current positiion and sets the current position to be after e
nextIndex()	Returns the index of the next element
previousIndex()	Returns the index of the previous element

Liste iterators in Java

- The **java.util.ListIterator** interface includes the following methods

remove()	Removes the element returned by the most recent or previous operation
set(e)	Replaces the element returned by the most recent call to the next or previous operation with e

Comparison between the two PositionalList ADTs

Positional List ADT Method	java.util.List Method	ListIterator Method	Notes
size()	size()		$O(1)$ time
isEmpty()	isEmpty()		$O(1)$ time
	get(i)		A is $O(1)$, L is $O(\min\{i, n-i\})$
first()	listIterator()		first element is next
last()	listIterator(size())		last element is previous
before(p)		previous()	$O(1)$ time
after(p)		next()	$O(1)$ time
set(p, e)		set(e)	$O(1)$ time
	set(i, e)		A is $O(1)$, L is $O(\min\{i, n-i\})$
	add(i, e)		$O(n)$ time
addFirst(e)	add(0, e)		A is $O(n)$, L is $O(1)$
addFirst(e)	addFirst(e)		only exists in L, $O(1)$
addLast(e)	add(e)		$O(1)$ time
addLast(e)	addLast(e)		only exists in L, $O(1)$
addAfter(p, e)		add(e)	insertion is at cursor; A is $O(n)$, L is $O(1)$
addBefore(p, e)		add(e)	insertion is at cursor; A is $O(n)$, L is $O(1)$
remove(p)		remove()	deletion is at cursor; A is $O(n)$, L is $O(1)$
	remove(i)		A is $O(1)$, L is $O(\min\{i, n-i\})$

Converting lists into arrays

- Lists are a beautiful concept and they can be applied in a number of different contexts but there are instances where it can be useful to treat a list like an array
- The **java.util.Collection** includes the following methods for generating an array that has the same element as the given collection:

toArray()	Returns an array of elements of type Object containing all the elements in this collection
toArray(A)	Returns an array of elements of the same element type as A containing all the elements in this collection.