CS121 Data Structures A, C
Iterators

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Iterators

An **iterator** is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time

The iterator interface (defined in `java.util.Iterator`) supports the two methods:

- 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 interface uses generics with a parameterized element type

`Scanner` formally implements the `Iterator<String>` interface

And an *optionally* supported method:

- remove(): Removes from the collection the element returned by the most recent call to `next()` (or an error is generated)

Iterators (cont'd)

If the `next()` method of an iterator is called when no further elements are available, a `NoSuchElementException` is thrown

The `hasNext()` method can be used to detect that condition before calling `next()`

If the variable `iter` denotes some instance of type `Iterator<String>`, then we can write:

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

The `remove()` method can be used to filter a collection of elements, e.g. to discard all negative numbers from a data set

The Iterable Interface

There is no way to "reset" the iterator back to the beginning of 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

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

A general loop construct for processing elements of the iterator:

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

supported for any instance *collection* of an iterable class, is equivalent to:

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

ElementType = the type of object returned by the iterator

variable takes on element values with the *loopBody*

Example: Removing Negative Numbers

The iterator's `remove` method cannot be invoked when using the for-each loop syntax. *Why?*

Example: Removing Negative Numbers

The iterator's `remove` method cannot be invoked when using the for-each loop syntax. *Why?*

We must explicitly use an iterator

```
ArrayList<Double> data;
// populate with random numbers (not shown)
Iterator<Double> walk = data.iterator();
while (walk.hasNext()) {
  if (walk.next() < 0.0)
    walk.remove();
}
```

If removal is not supported, an `UnsupportedOperationException` is conventionally thrown

Implementing Iterators

There are two general styles for implementing iterators:

- A **snapshot iterator** maintains its own private copy of the sequence of elements, which is constructed at the time the iterator object is created. Requires $O(n)$ time and $O(n)$ auxiliary space, upon construction, to copy and store a collection of n elements.

- A **lazy iterator** does not make an upfront copy, but performs a piecewise traversal of the primary structure only when the `next()` method is called. Can be implemented with $O(1)$ space and $O(1)$ construction time. Downside: affected by changes in the primary structure.

Many of the iterators in Java's libraries implement a "fail-fast" behaviour that immediately invalidates such an iterator if its underlying collection is modified unexpectedly

```
1   //---------- nested ArrayIterator class ----------
2   /**
3    * A (nonstatic) inner class. Note well that each instance contains an implicit
4    * reference to the containing list, allowing it to access the list's members.
5    */
6   private class ArrayIterator implements Iterator<E> {
7     private int j = 0;                          // index of the next element to report
8     private boolean removable = false;          // can remove be called at this time?
9
10    /**
11     * Tests whether the iterator has a next object.
12     * @return true if there are further objects, false otherwise
13     */
14    public boolean hasNext( ) { return j < size; }  // size is field of outer instance
15
16    /**
17     * Returns the next object in the iterator.
18     *
19     * @return next object
20     * @throws NoSuchElementException if there are no further elements
21     */
22    public E next( ) throws NoSuchElementException {
23      if (j == size) throw new NoSuchElementException("No next element");
24      removable = true;     // this element can subsequently be removed
25      return data[j++];     // post-increment j, so it is ready for future call to next
26    }
27
28    /**
29     * Removes the element returned by most recent call to next.
30     * @throws IllegalStateException if next has not yet been called
31     * @throws IllegalStateException if remove was already called since recent next
32     */
33    public void remove( ) throws IllegalStateException {
34      if (!removable) throw new IllegalStateException("nothing to remove");
35      ArrayList.this.remove(j-1);     // that was the last one returned
36      j--;                            // next element has shifted one cell to the left
37      removable = false;              // do not allow remove again until next is called
38    }
39  } //---------- end of nested ArrayIterator class ----------
40
41  /** Returns an iterator of the elements stored in the list. */
42  public Iterator<E> iterator( ) {
43    return new ArrayIterator( );        // create a new instance of the inner class
44  }
```

Iterations with the LinkedPositionalList Class

Should we support iteration of the *elements* of the list or the *positions* of the list?

If we allow a user to iterate through all positions of the list, those positions could be used to access the underlying elements.

So support for position iteration is more general.

However, it is more standard for a container class to support iteration of the core elements by default, so that the for-each loop syntax could be used, e.g.

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

assuming that variable `waitlist` has type `LinkedPositionalList<String>`.

For maximum convenience, we will support *both* forms of iteration.

Iterations with the LinkedPositionalList Class (cont'd)

The standard `iterator()` method returns an iterator of the elements of the list.

We provide a `positions()` method that returns an instance that is `Iterable` so that we can write:

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

We define three new inner classes:

- `PositionIterator`—the core functionality of our list iterations
- `PositionIterable`—constructs and returns a new `PositionIterator` object each time `iterator()` is called
- `ElementIterator`—adapts the `PositionIterator` class, and lazily manages a position iterator instance, while returning the element stored at each position when `next()` is called

The `positions()` method of the top-level class returns a new `PositionIterable` instance

```
1   //---------- nested PositionIterator class ----------
2   private class PositionIterator implements Iterator<Position<E>> {
3     private Position<E> cursor = first( );    // position of the next element to report
4     private Position<E> recent = null;        // position of last reported element
5     /** Tests whether the iterator has a next object. */
6     public boolean hasNext( ) { return (cursor != null);    }
7     /** Returns the next position in the iterator. */
8     public Position<E> next( ) throws NoSuchElementException {
9       if (cursor == null) throw new NoSuchElementException("nothing left");
10      recent = cursor;                // element at this position might later be removed
11      cursor = after(cursor);
12      return recent;
13    }
14    /** Removes the element returned by most recent call to next. */
15    public void remove( ) throws IllegalStateException {
16      if (recent == null) throw new IllegalStateException("nothing to remove");
17      LinkedPositionalList.this.remove(recent);                // remove from outer list
18      recent = null;                    // do not allow remove again until next is called
19    }
20  } //---------- end of nested PositionIterator class ----------
21
22  //---------- nested PositionIterable class ----------
23  private class PositionIterable implements Iterable<Position<E>> {
24    public Iterator<Position<E>> iterator( ) { return new PositionIterator( ); }
25  } //---------- end of nested PositionIterable class ----------
26
27  /** Returns an iterable representation of the list's positions. */
28  public Iterable<Position<E>> positions( ) {
29    return new PositionIterable( );          // create a new instance of the inner class
30  }
31
32  //---------- nested ElementIterator class ----------
33  /* This class adapts the iteration produced by positions() to return elements. */
34  private class ElementIterator implements Iterator<E> {
35    Iterator<Position<E>> posIterator = new PositionIterator( );
36    public boolean hasNext( ) { return posIterator.hasNext( ); }
37    public E next( ) { return posIterator.next( ).getElement( ); } // return element!
38    public void remove( ) { posIterator.remove( ); }
39  }
40
41  /** Returns an iterator of the elements stored in the list. */
42  public Iterator<E> iterator( ) { return new ElementIterator( ); }
```

The Java Collections Framework

Java provides many data structure interfaces and classes, which together form the **Java Collections Framework**

It is part of the `java.util` package

The root interface in the Java collections framework is named `Collection`

This is a general interface for any data structure, such as a list, that represents a collection of elements and it includes methods like `size()`, `isEmpty()`, `iterator()`, etc.

`Collection` is a superinterface for other interfaces like `Deque`, `List`, `Queue`, `Set`, `Map`, etc.

The Java Collections Framework also includes concrete classes implementing various interfaces with a combination of properties and underlying representations

Several Classes in the Java Collections Framework

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

List Iterators in Java

The `java.util.LinkedList` class does not expose a position concept to users in its API, as we do in our positional list ADT

Instead, the preferred way to access and update a `LinkedList` object in Java 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 its current position as being

- before the first element
- between two elements
- after the last element

That is, it uses a list **cursor**

The `java.util.ListIterator` Interface

The `java.util.ListIterator` interface includes the 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 of the iterator
- previous(): Returns the element *e* before the current position and sets the current position to be before *e*
- next(): Returns the element *e* after the current position 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
- remove(): Removes the element returned by the most recent `next` or `previous` operation
- set(*e*): Replaces the element returned by the most recent call to the `next` or `previous` operation with *e*

Comparison to Our Positional List ADT

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)		$O(n)$ time

Sorting a Positional List

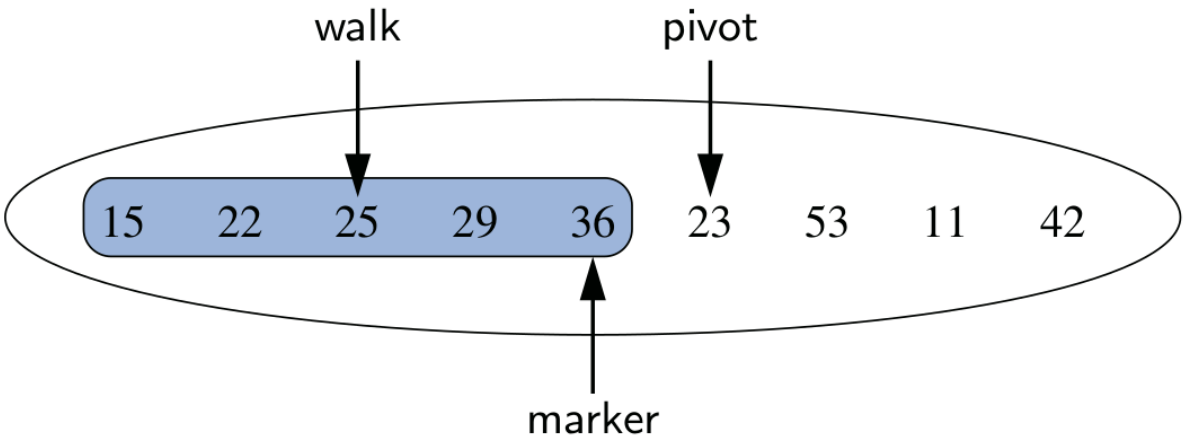
We consider an implementation of **Insertion Sort** that operates on a `PositionalList`

Each element is placed relative to a growing collection of previously sorted elements

The variable `marker` represents the rightmost position of the currently sorted portion of a list

During each pass, we consider the position just past the marker as the `pivot`

We use `walk` to identify where the `pivot`'s element belongs relative to the sorted portion



Insertion Sort on a Positional List

```
1  /** Insertion-sort of a positional list of integers into nondecreasing order */
2  public static void insertionSort(PositionalList<Integer> list) {
3    Position<Integer> marker = list.first( );      // last position known to be sorted
4    while (marker != list.last( )) {
5      Position<Integer> pivot = list.after(marker);
6      int value = pivot.getElement( );            // number to be placed
7      if (value > marker.getElement( ))           // pivot is already sorted
8        marker = pivot;
9      else {                                      // must relocate pivot
10       Position<Integer> walk = marker;          // find leftmost item greater than value
11       while (walk != list.first( ) && list.before(walk).getElement( ) > value)
12         walk = list.before(walk);
13       list.remove(pivot);                       // remove pivot entry and
14       list.addBefore(walk, value);              // reinsert value in front of walk
15     }
16   }
17 }
```

What is the running time of this method for each implementation of a positional list?

Case Study: Maintaining Access Frequencies

We consider maintaining a collection of elements while keeping track of the number of times each element is accessed

Keeping such access counts allows us to know which elements are among the most popular

We will model this with a new **favorites list ADT** that supports the `size` and `isEmpty` methods as well as the following:

access(e): Accesses the element e, adding it to the favorites list if it is not already present, and increments its access count

remove(e): Removes element e from the favorites list, if present

getFavorites(k): Returns an iterable collection of the k most accessed elements

Using a Sorted List and the Composition Pattern

We store elements in a linked list, keeping them in nonincreasing order of access counts

Access or remove: search the list from the most frequently accessed to the least frequently accessed

The k most accessed elements are the first k entries of the list

After we access an element, we use a technique similar to a single pass of the insertion sort algorithm to locate a valid position to relocate the element

The object-oriented design pattern named the **composition pattern** is used to define a single object that is composed of two or more other objects

We define a nonpublic nested class `Item` that stores the element and its access count as a single instance

```
1  /** Maintains a list of elements ordered according to access frequency. */
2  public class FavoritesList<E> {
3    // ---------------- nested Item class ----------------
4    protected static class Item<E> {
5      private E value;
6      private int count = 0;
7      /** Constructs new item with initial count of zero. */
8      public Item(E val) { value = val; }
9      public int getCount( ) { return count; }
10     public E getValue( ) { return value; }
11     public void increment( ) { count++; }
12   } //---------- end of nested Item class ----------
13
14   PositionalList<Item<E>> list = new LinkedPositionalList<>( );    // list of Items
15   public FavoritesList( ) { }                    // constructs initially empty favorites list
16
17   // nonpublic utilities
18   /** Provides shorthand notation to retrieve user's element stored at Position p. */
19   protected E value(Position<Item<E>> p) { return p.getElement( ).getValue( ); }
20
21   /** Provides shorthand notation to retrieve count of item stored at Position p. */
22   protected int count(Position<Item<E>> p) {return p.getElement( ).getCount( );}
23
24   /** Returns Position having element equal to e (or null if not found). */
25   protected Position<Item<E>> findPosition(E e) {
26     Position<Item<E>> walk = list.first( );
27     while (walk != null && !e.equals(value(walk)))
28       walk = list.after(walk);
29     return walk;
30   }
```

```
31    /** Moves item at Position p earlier in the list based on access count. */
32    protected void moveUp(Position<Item<E>> p) {
33      int cnt = count(p);                          // revised count of accessed item
34      Position<Item<E>> walk = p;
35      while (walk != list.first( ) && count(list.before(walk)) < cnt)
36        walk = list.before(walk);                  // found smaller count ahead of item
37      if (walk != p)
38        list.addBefore(walk, list.remove(p));      // remove/reinsert item
39    }
40
41    // public methods
42    /** Returns the number of items in the favorites list. */
43    public int size( ) { return list.size( ); }
44
45    /** Returns true if the favorites list is empty. */
46    public boolean isEmpty( ) { return list.isEmpty( ); }
47
48    /** Accesses element e (possibly new), increasing its access count. */
49    public void access(E e) {
50      Position<Item<E>> p = findPosition(e);       // try to locate existing element
51      if (p == null)
52        p = list.addLast(new Item<E>(e));          // if new, place at end
53      p.getElement( ).increment( );                // always increment count
54      moveUp(p);                                   // consider moving forward
55    }
56
57    /** Removes element equal to e from the list of favorites (if found). */
58    public void remove(E e) {
59      Position<Item<E>> p = findPosition(e);       // try to locate existing element
60      if (p != null)
61        list.remove(p);
62    }
63
64    /** Returns an iterable collection of the k most frequently accessed elements. */
65    public Iterable<E> getFavorites(int k) throws IllegalArgumentException {
66      if (k < 0 || k > size( ))
67        throw new IllegalArgumentException("Invalid k");
68      PositionalList<E> result = new LinkedPositionalList<>( );
69      Iterator<Item<E>> iter = list.iterator( );
70      for (int j=0; j < k; j++)
71        result.addLast(iter.next( ).getValue( ));
72      return result;
73    }
74  }
```

Summary

Reading

Sections 7.4–7.8 of the main textbook

Questions?