Lists and Iterators

- Implement and analyze dynamic arrays
- Define the positional list ADT
- Implement positional lists using a doubly linked list.
- Define and implement Iterators
- List-based algorithms in Java Collections framework.

Stacks - Review

- An **Abstract Data Type** structure specifies:
  - Data stored
  - Operations on data
  - Error conditions associated with operations
- **Stacks** are LIFO data structures
  - Operations:
    - **push**(object): inserts an element
    - **pop**(): removes and returns the last inserted element
    - **top**(): returns the last inserted element without removing it
    - integer **size**(): returns the number of elements stored
    - boolean **isEmpty**(): indicates whether no elements are stored

Stacks - Review

- LIFO data structure
  - Direct applications
    - Page-visited history in a Web browser
    - Undo sequence in a text editor
    - Chain of method calls in the Java Virtual Machine
  - Indirect applications
    - Auxiliary data structure for algorithms
    - Component of other data structures
- Implementation of Stacks
  - Array based - The space used is $O(n)$, Each operation runs in time $O(1)$
  - Singly Linked List - top of the stack stored at the front of the list

Queues- Review

- Insertions and deletions follow the **first-in first-out** scheme
- Queue operations:
  - enqueue(object): **inserts** an element **at the end** of the queue
  - object dequeue(): **removes** and returns the element **at the front** of the queue
  - object first(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Direct applications - **Waiting li**sts, bureaucracy, **Access to shared resources** (e.g., printer), **Multiprogramming**
- Indirect applications - Auxiliary data structure for algorithms, Component of other data structures
- **Implementation of Queues**
  - Array-based queues - Use an array of size N in a **circular fashion,** use the **modulo** operator
  - Application: Round Robin Schedulers

Double-Ended Queues - Review

- Double-Ended Queues - A queue-like data structure that **supports insertion and deletion at both the front and the back of the queue**
- Deque Abstract Data Type:
  - **addFirst**(*e*): insert a new element *e* at the front of the deque.
  - **addLast**(*e*): insert a new element *e* at the back of the deque.
  - **removeFirst**( ): remove and return the first element of the deque (or null if the deque is empty).
  - **removeLast**( ): remove and return the last element of the deque (or null if the deque is empty).
  - **first**( ): returns the first element of the deque, without removing it (or null if the deque is empty).
  - **last**( ): returns the last element of the deque, without removing it (or null if the deque is empty).
  - **size**( ): returns the number of elements in the deque.
  - **isEmpty**( ): returns a boolean indicating whether the deque is empty.

Double-Ended Queues - Review

- **Implementing a Deque**
  - Using a Circular Array - a representation similar to the ArrayQueue class
  - Using a Doubly Linked List
    - The DoublyLinkedList class already implements the entire Deque interface; we simply need to add the declaration “implements Deque<E>” to that class definition in order to use it as a deque.
- **Performance of the Deque Operations**
  - every method runs in $O(1)$ time.

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Lists and Iterators

Lists and Iterators

- **Stack, queue, and deque** abstract data types represent a linearly ordered sequence of elements that **only allow insertions and deletions at the front or back of a sequence**.
- In this lesson, we explore several abstract data types that represent a linear sequence of elements, but with **more general support for adding or removing elements at arbitrary positions**.
- Java defines a general interface, java.util.**List**, that includes the following **index-based methods**:

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 $[0, \text{size}() - 1]$.

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 $[0, \text{size}() - 1]$.

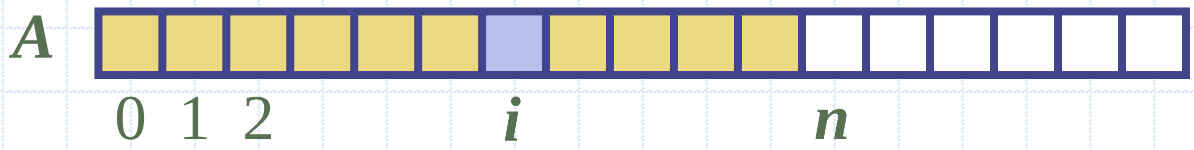
Example

- A sequence of List operations:

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

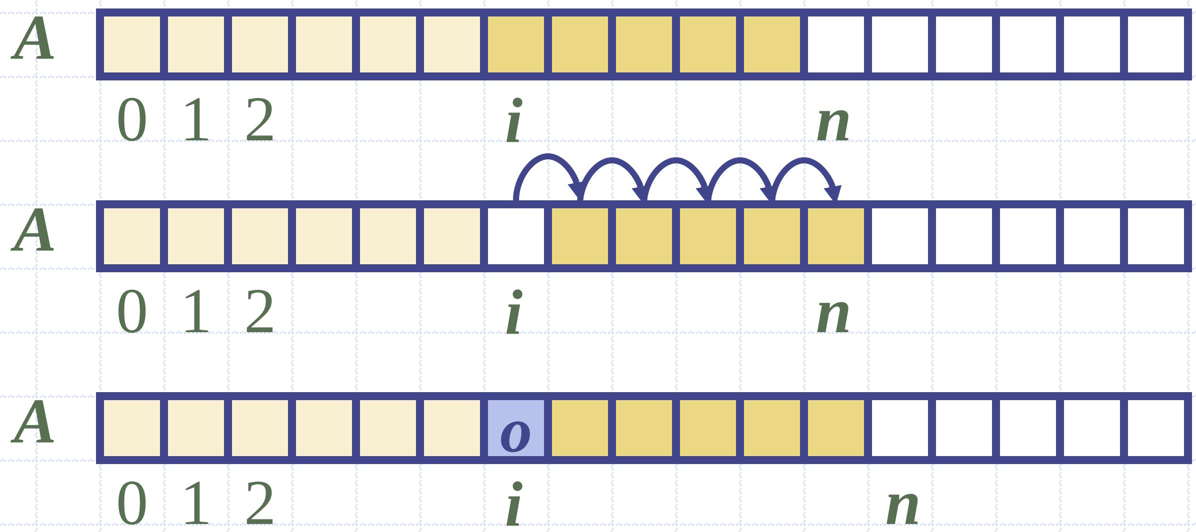
Array Lists

- 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).



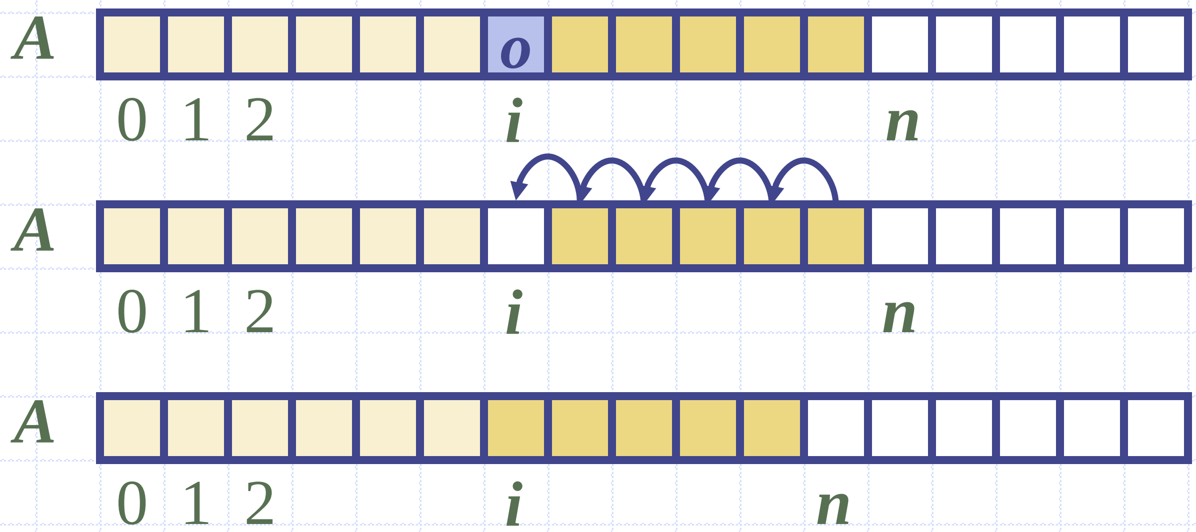
Insertion

- 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



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



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

```
// public methods
/** Returns the number of elements in the array list. */
public int size( ) { return size; }
/** Returns whether the array list is empty. */
public boolean isEmpty( ) { return size == 0; }
/** Returns (but does not remove) the element at index i. */
public E get(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  return data[i];
}
/** Replaces the element at index i with e, and returns the replaced element. */
public E set(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

Java Implementation, 2

```
/** Inserts element e to be at index i, shifting all subsequent elements later. */
public void add(int i, E e) throws IndexOutOfBoundsException,
                                   IllegalStateException {
  checkIndex(i, size + 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++;
}
/** 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);
}
}
```

Dynamic Arrays

- The ArrayList implementation in previous code requires that a **fixed maximum capacity** be declared, throwing an exception if attempting to add an element once full.
- Java's **ArrayList class** provides a more robust abstraction, allowing a user to add elements to the list, with **no apparent limit on the overall capacity**.
- An array list instance **maintains an internal array** that often has greater capacity than the current length of the list.

Dynamic Arrays

- If a user continues to add elements to a list, all reserved capacity in the underlying array will eventually be exhausted.
- In that case, the class requests a new, larger array from the system, and **copies all references from the smaller array into the beginning of the new array**.
- At that point in time, **the old array is no longer needed**, so it can be reclaimed by the system.

Dynamic Array-based Array List

- When the array is full, we replace the array with a larger one:
  1. *Allocate a new array B with larger capacity.*
  2. *Set B[k]=A[k], for k=0, . . . ,n−1, where n denotes current number of items.*
  3. *Set A = B, that is, reassign reference A to the new array.*
  4. *Insert the new element in the new array.*

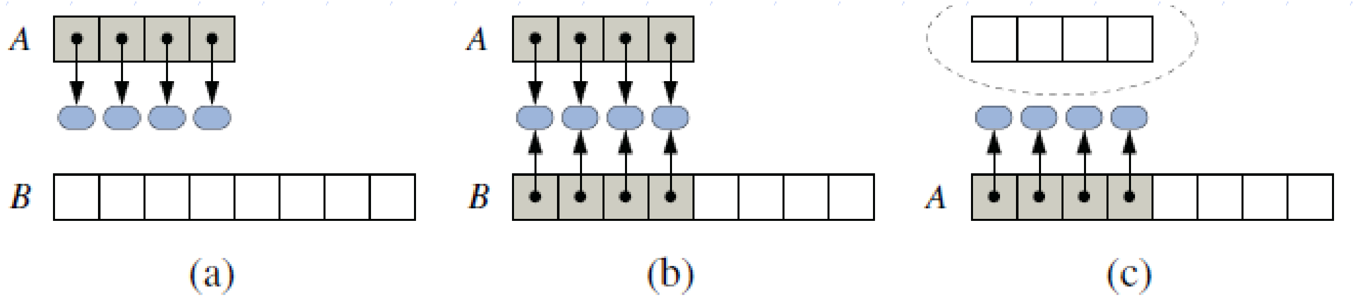


Figure 7.3: An illustration of "growing" a dynamic array: (a) create new array *B*; (b) store elements of *A* in *B*; (c) reassign reference *A* to the new array. Not shown is the future garbage collection of the old array, or the insertion of a new element.

Dynamic Array-based Array List

- Code Fragment 7.4 provides a **concrete implementation of a resize method**, which should be included as a protected method within the original ArrayList class.
- The instance variable data corresponds to array *A* in the above discussion, and local variable temp corresponds to array *B*.
- ```
  /** Resizes internal array to have given capacity >= 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
  }
  ```

Code Fragment 7.4: An implementation of the ArrayList.resize method.
  ```

Dynamic Array-based Array List

- How large should the new array be?
  - a commonly used rule is for the new array to have **twice the capacity of the existing array** that has been filled:
- We redesign the **add** method so that it calls the new resize utility when detecting that the current array is filled (rather than throwing an exception).

```
/** 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...
```

Code Fragment 7.5: A revision to the ArrayList.add method, originally from Code Fragment 7.3, which calls the resize method of Code Fragment 7.4 when more capacity is needed.

Amortized Analysis of Dynamic Arrays

- Let push(o) be the operation that adds element o at the end of the list
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size
- We show that **performing a sequence of push operations** on a dynamic array is actually quite efficient.

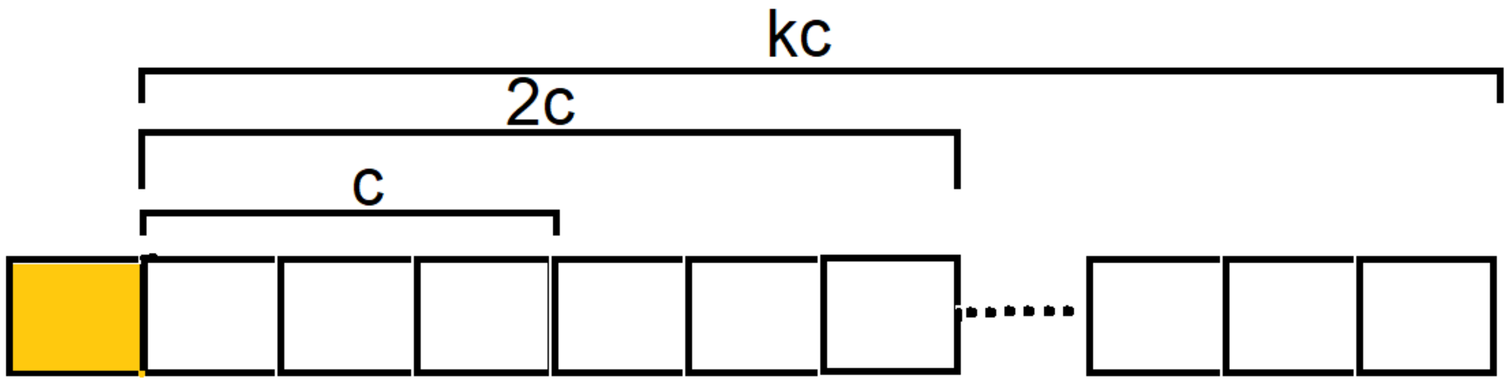
Algorithm *push*(o)
  if $t = A.length - 1$ **then**
    $B \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $B[i] \leftarrow A[i]$
    $A \leftarrow B$
  $n \leftarrow n + 1$
  $A[n-1] \leftarrow o$

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 *n* operations**, i.e., $T(n)/n$

Incremental Strategy Analysis

- Assume that over n push operations, time will have been spent initializing arrays of size c, $2c$, $3c$, . . . , kc .



- This means, we initialize the arrays $k = n/c$ times, where c is a constant.

Incremental Strategy Analysis

- The total time $T(n)$ of a series of n push operations includes initializing arrays of size $c, 2c, 3c, \ldots, kc$
- Other operations of ***push*** algorithm take constant time, therefore, $T(n)$ is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

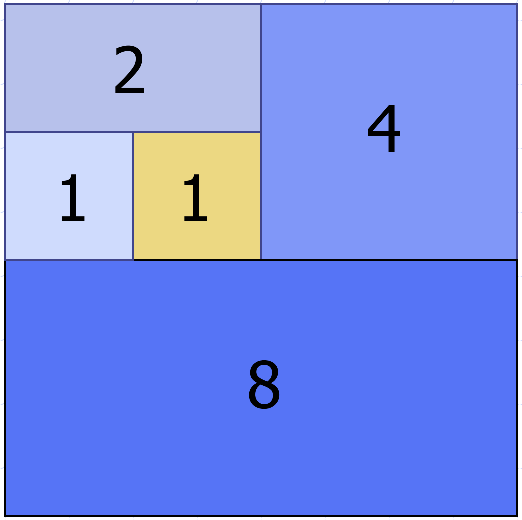
$$n + ck(k + 1)/2$$

- 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 using incremental strategy is $T(n)/n = O(n^2)/n = O(n)$

Doubling Strategy Analysis

- Assume that over n push operations, time will have been spent initializing arrays of size $1, 2, 4, 8, \ldots, 2^k$.
- This means for the n^{th} push the array size to be initialized is 2^k.
  - We write $n \rightarrow 2^k, log_2 n \rightarrow k\ log_2 2 = k$, so, $log_2 n \rightarrow k$
- This means, we initialize the arrays $k = \log_2 n$ times.

geometric series



Doubling Strategy Analysis

- The total time $T(n)$ of a series of n push operations includes initializing arrays of size $1 + 2 + 4 + 8 + \ldots + 2^k$.
- Other operations of ***push*** algorithm take constant time, therefore, $T(n)$ is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 \;= n + 2^{\log_2 n + 1} - 1$$
$$= n + 2 * 2^{\log_2 n} - 1 = n + 2n - 1 = 3n - 1$$

- $T(n)$ is $O(n)$ => the amortized time of a push operation is $O(1)$.

Positional Lists

- Numeric indices are **not a good choice for describing positions within a linked list** because, knowing only an element's index, the only way to reach it is to traverse the list incrementally from its beginning or end.
- Indices are not a good abstraction for describing a more local view of a position in a sequence, because the **index of an entry changes over time** due to insertions or deletions that happen earlier in the sequence.

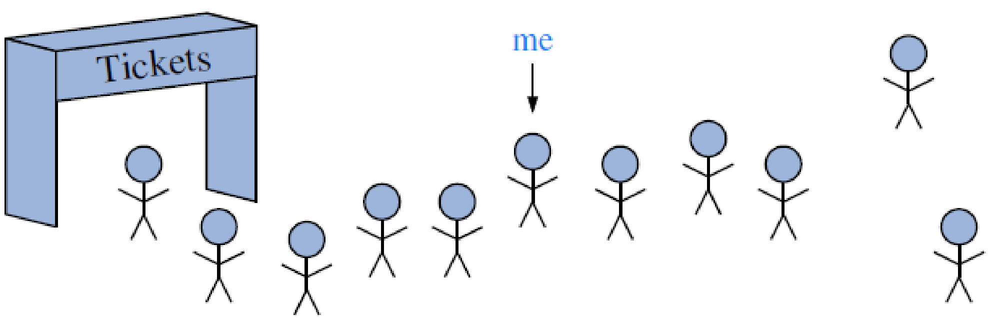


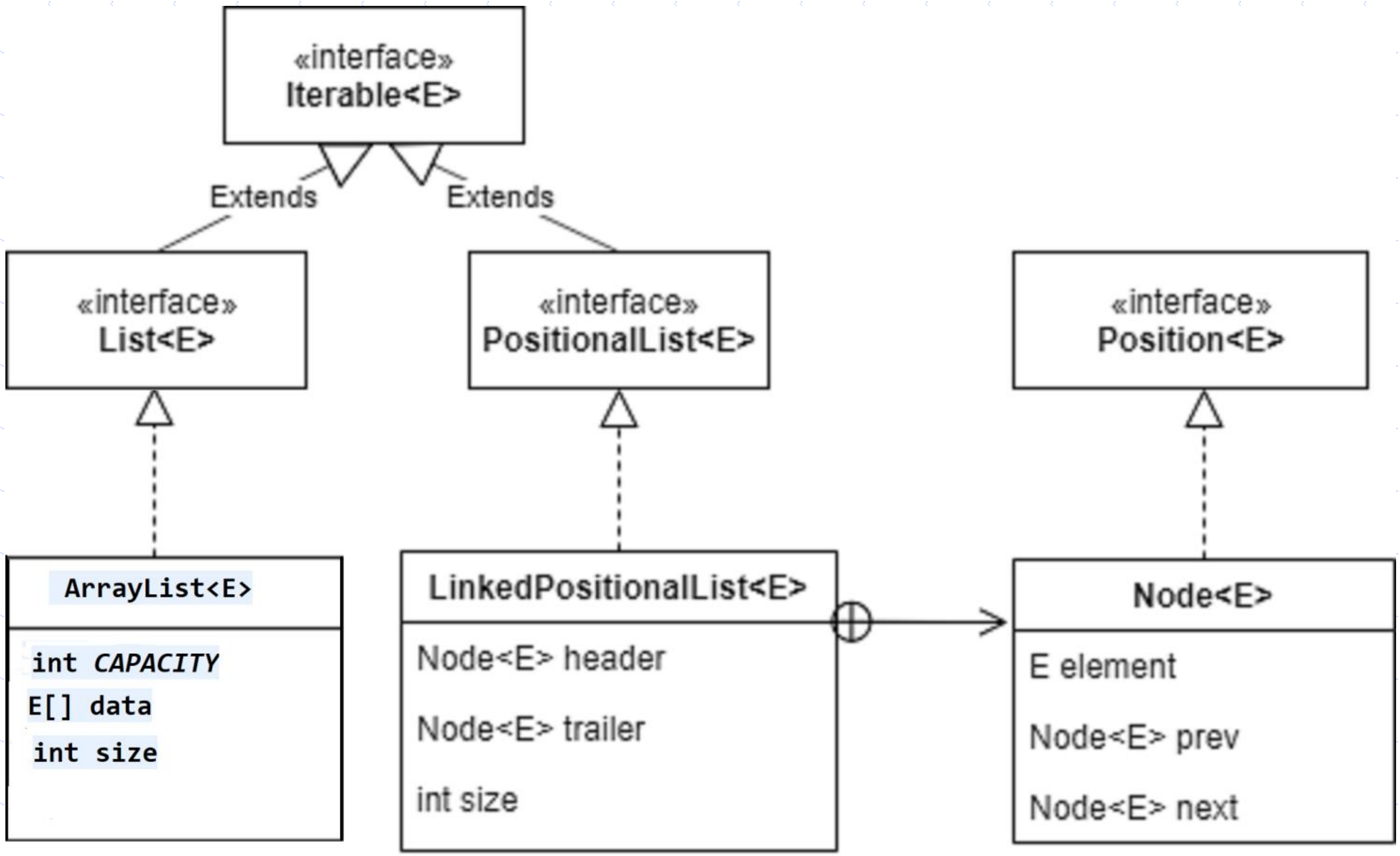
Figure 7.7: We wish to be able to identify the position of an element in a sequence without the use of an integer index. The label "me" represents some abstraction that identifies the position.

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.

ArrayList & Positional List ADT

- Interfaces and Classes:



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.

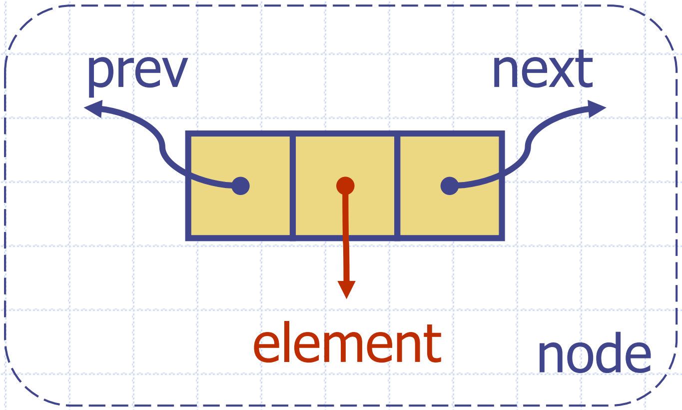
Example

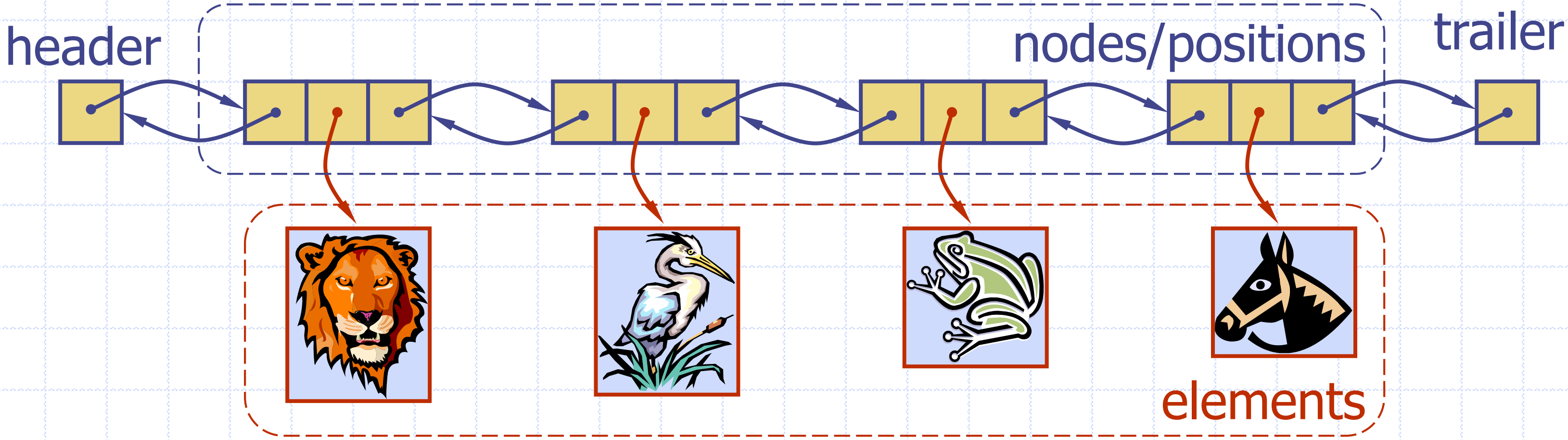
- A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first()	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement()	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last())	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

Positional List Implementation

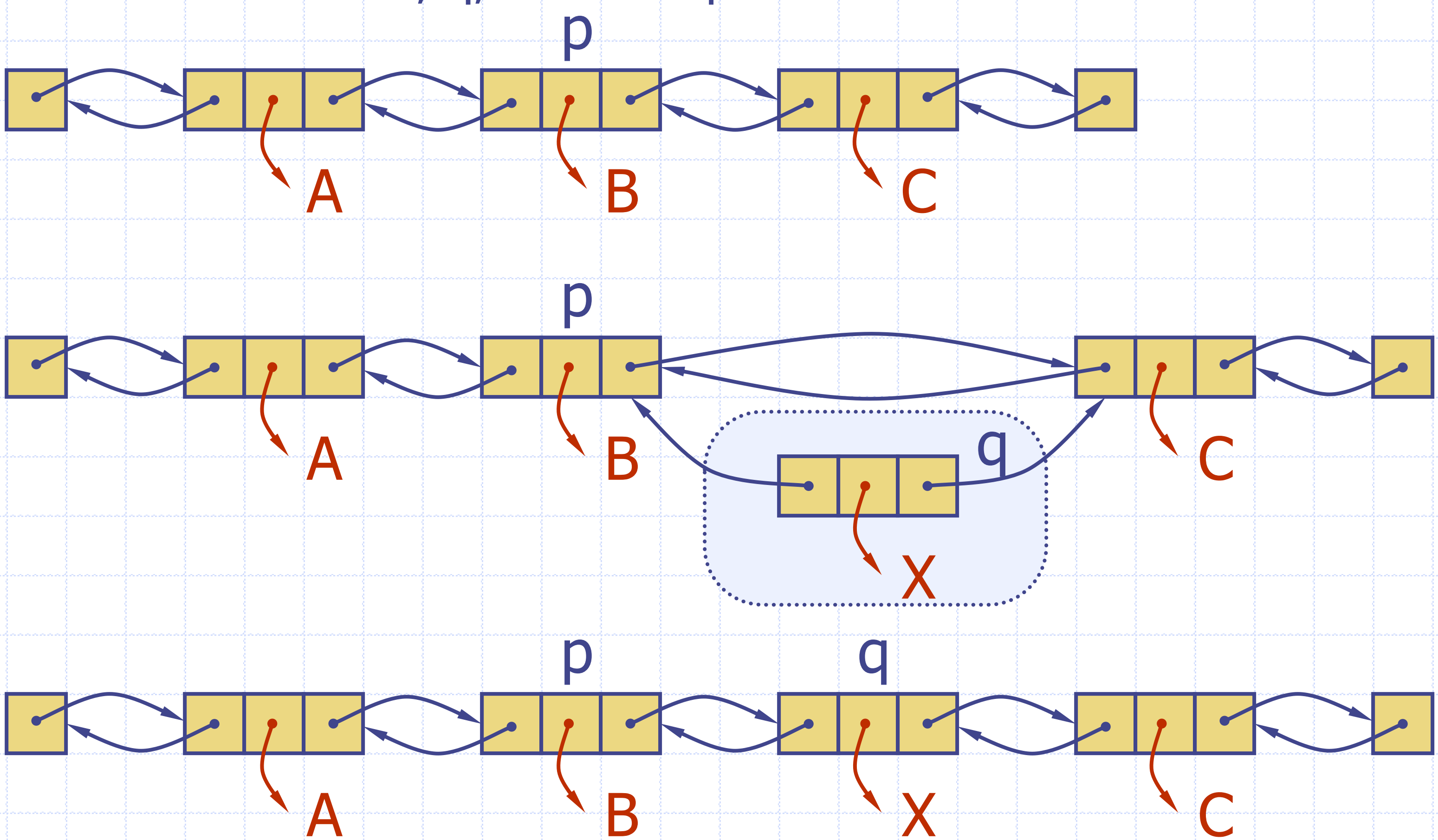
- The most natural way to implement a positional list is with a **doubly-linked list**.





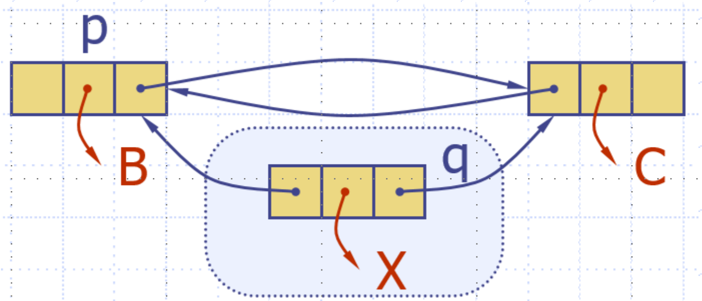
Insertion

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



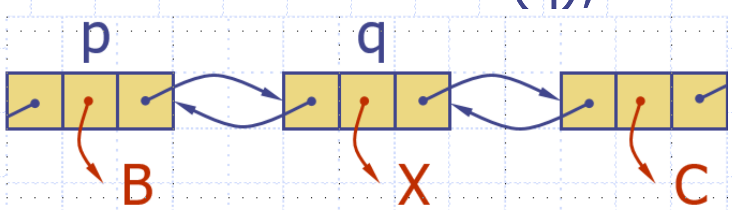
Insertion

- The newly constructed node q has a link to predecessor and a link to successor:



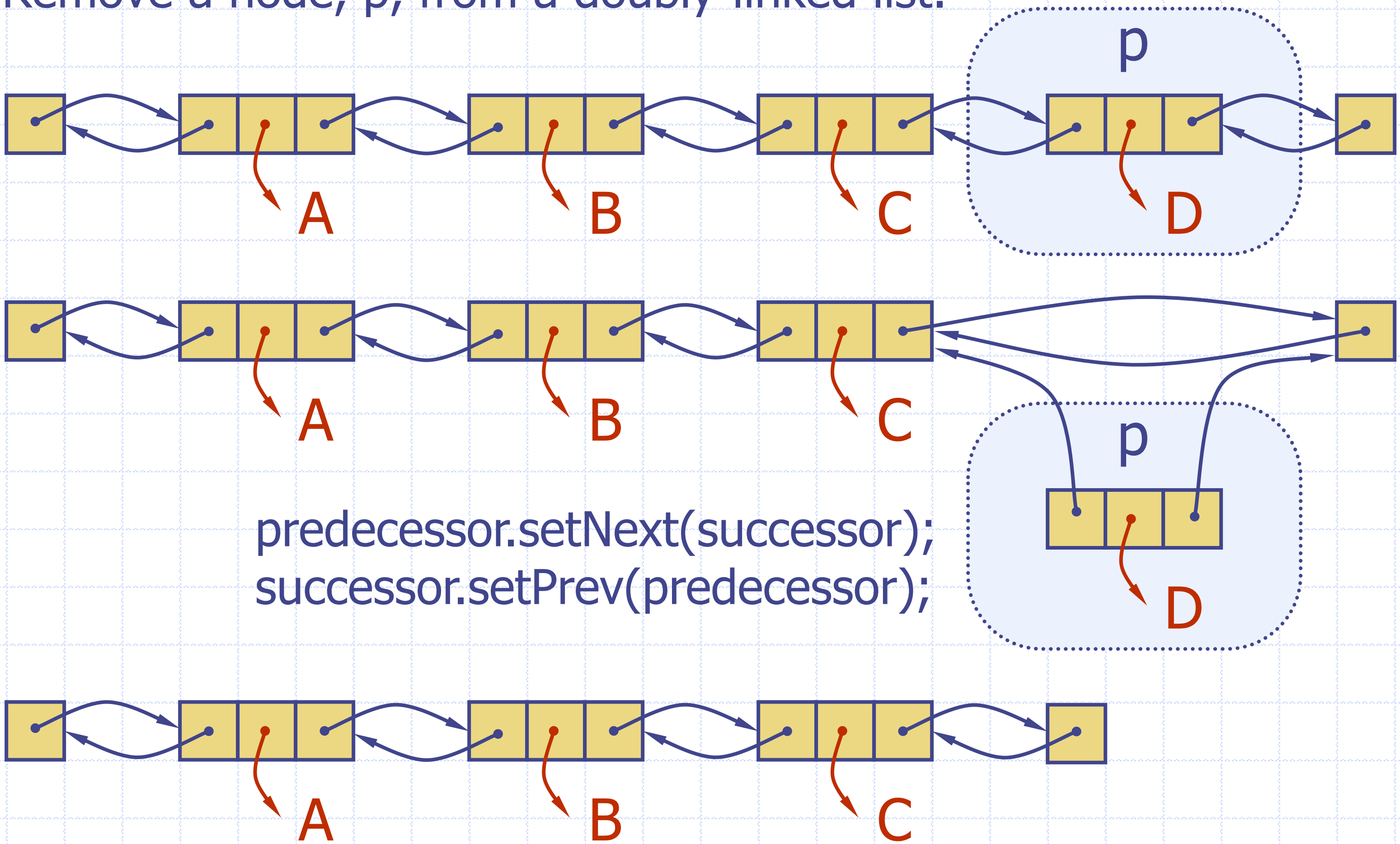
- Node p needs to set a link to q
- The old successor of p needs to set a link to q

```
p.setNext(q);
succ.setPrev(q);
```



Deletion

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




```
predecessor.setNext(successor);
successor.setPrev(predecessor);
```

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 (to be used for traversing 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:

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

is equivalent to:

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

Iterator example

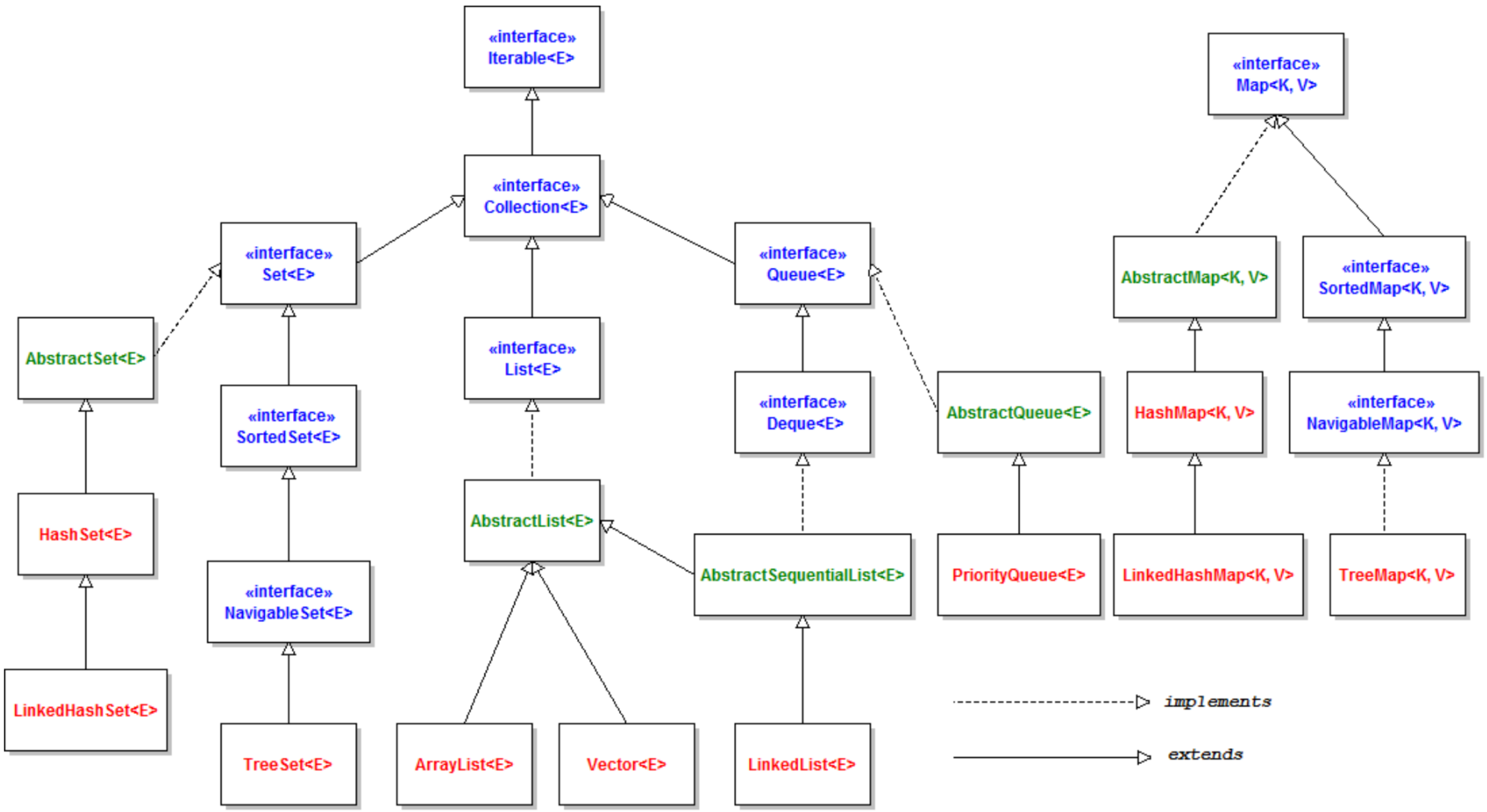
```
public static void main(String[] args) {
  int N = 50;
  Random r = new Random();
  ArrayList<Double> data;      // populate with random numbers (not shown)
  data = new ArrayList<>(N);
  for (int i = 0; i < N; i++)
    data.add(r.nextGaussian());
  Iterator<Double> walk = data.iterator();
  while (walk.hasNext())
    if (walk.next() < 0.0)
      walk.remove();
  System.out.println("Length of remaining data set: " + data.size());
  System.out.println(data);

}
```

The Java Collections Framework

- This framework, which is part of the **java.util** package, includes versions of several of the data structures we are studying in this course.
- The root interface in the Java collections framework is named **Collection**.
- The **Collection** interface includes many methods, including some we have already seen (e.g., **size**( ), **isEmpty**( ), **iterator**( )).
- It is a **superinterface** for other interfaces in the Java Collections Framework that can hold elements, including the **java.util** interfaces **Deque**, **List**, and **Queue**, and other subinterfaces discussed later in this book, including **Set**.

The Java Collections Framework



Class diagram of Java Collections framework

https://www.codejava.net/java-core/collections/overview-of-java-collections-framework-api-uml-diagram

The Java Collections Framework

- The Java Collections Framework also includes **concrete classes implementing various interfaces** with a combination of properties and underlying representations.
- Robust classes provide support for ***concurrency***, allowing multiple processes to share use of a data structure in a thread-safe manner.
- **LinkedList** class uses a **ListIterator**, returned by the list's **listIterator**() method, to provide forward and backward traversal methods.
- Java Collections Framework, contains a number of simple **algorithms** implemented as static methods in the **java.util.Collections** class.