

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



Contents

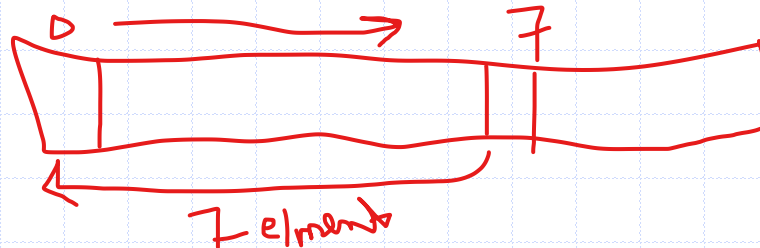
7.1	The List ADT	258
7.2	Array Lists	260
7.2.1	Dynamic Arrays	263
7.2.2	Implementing a Dynamic Array	264
7.2.3	Amortized Analysis of Dynamic Arrays	265
7.2.4	Java's StringBuilder class	269
7.3	Positional Lists	270
7.3.1	Positions	272
7.3.2	The Positional List Abstract Data Type	272
7.3.3	Doubly Linked List Implementation	276
7.4	Iterators	282
7.4.1	The Iterable Interface and Java's For-Each Loop	283
7.4.2	Implementing Iterators	284
7.5	The Java Collections Framework	288
7.5.1	List Iterators in Java	289
7.5.2	Comparison to Our Positional List ADT	290
7.5.3	List-Based Algorithms in the Java Collections Framework	291
7.6	Sorting a Positional List	293
7.7	Case Study: Maintaining Access Frequencies	294
7.7.1	Using a Sorted List	294
7.7.2	Using a List with the Move-to-Front Heuristic	297
7.8	Exercises	300

Introduction

- In this chapter, 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*.
- However, designing a single abstraction that is well suited for efficient implementation with either an array or a linked list is challenging, given the very different nature of these two fundamental data structures.

Introduction

- Locations within an array are easily described with an integer index.
- An index of an element **e** in a sequence is equal to the **number of elements before e** in that sequence.
- By this definition, the **first element of a sequence has index 0**, and the last has index **$n-1$** , assuming that **n** denotes the total **number of elements**.
- The notion of an element's index is well defined for a linked list as well.



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

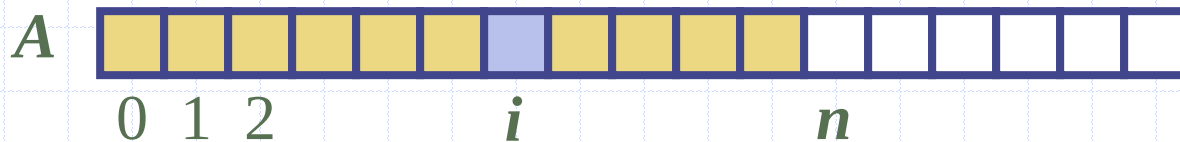
Example

- 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)

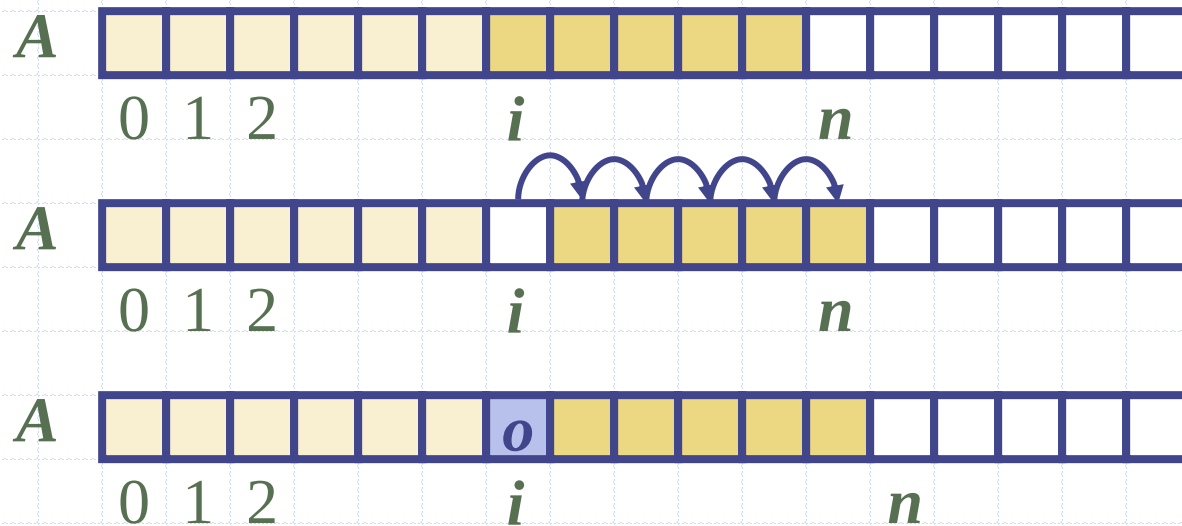
Array Lists

- 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).



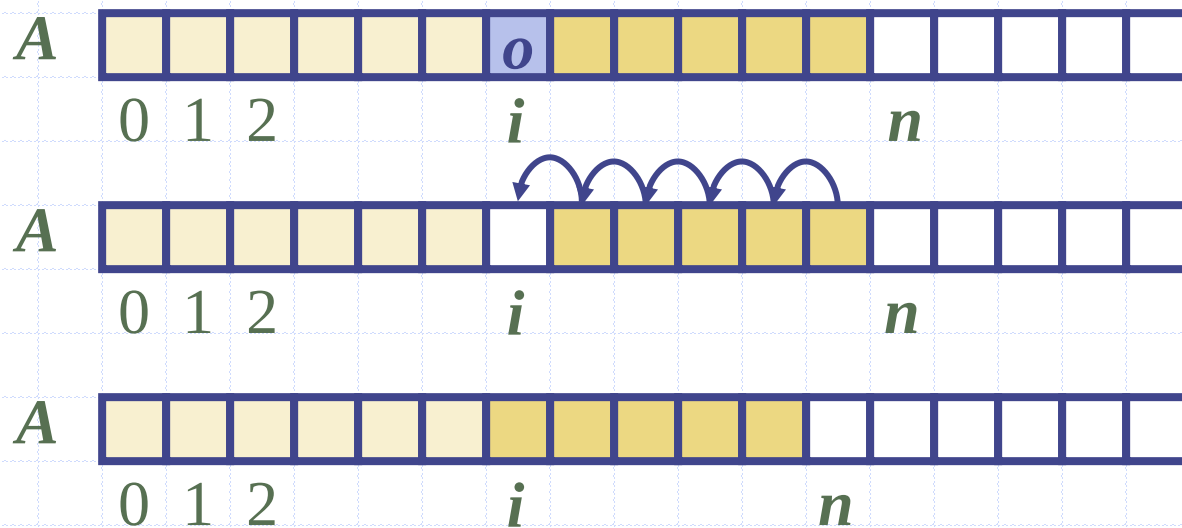
Insertion

- In an operation ***add***(*i*, *o*), we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \dots, A[n - 1]$
- 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 $A[i + 1], \dots, A[n - 1]$
- 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 ...

D-Array

A formal definition of the List interface;

```
1  /** A simplified version of the java.util.List interface. */
2  public interface List<E> {
3      /** Returns the number of elements in this list. */
4      int size();
5
6      /** Returns whether the list is empty. */
7      boolean isEmpty();
8
9      /** Returns (but does not remove) the element at index i. */
10     E get(int i) throws IndexOutOfBoundsException;
11
12     /** Replaces the element at index i with e, and returns the replaced element. */
13     E set(int i, E e) throws IndexOutOfBoundsException;
14
15     /** Inserts element e to be at index i, shifting all subsequent elements later. */
16     void add(int i, E e) throws IndexOutOfBoundsException;
17
18     /** Removes/returns the element at index i, shifting subsequent elements earlier. */
19     E remove(int i) throws IndexOutOfBoundsException;
20 }
```

Java Implementation

```
1 public class ArrayList<E> implements List<E> {  
2     // instance variables  
3     public static final int CAPACITY=16;    // default array capacity  
4     private E[] data;                       // generic array used for storage  
5     private int size = 0;                   // current number of elements  
6     // constructors  
7     public ArrayList() { this(CAPACITY); }  // constructs list with default capacity  
8     public ArrayList(int capacity) {        // constructs list with given capacity  
9         data = (E[]) new Object[capacity]; // safe cast; compiler may give warning  
10 }
```

Code Fragment 7.2: An implementation of a simple ArrayList class with bounded capacity. (Continues in Code Fragment 7.3.)

Java Implementation

```
11 // public methods
12 /** Returns the number of elements in the array list. */
13 public int size() { return size; }
14 /** Returns whether the array list is empty. */
15 public boolean isEmpty() { return size == 0; }
16 /** Returns (but does not remove) the element at index i. */
17 public E get(int i) throws IndexOutOfBoundsException {
18     checkIndex(i, size);
19     return data[i];
20 }
21 /** Replaces the element at index i with e, and returns the replaced element. */
22 public E set(int i, E e) throws IndexOutOfBoundsException {
23     checkIndex(i, size);
24     E temp = data[i];
25     data[i] = e;
26     return temp;
27 }
```

Java Implementation, 2

```
28  /** Inserts element e to be at index i, shifting all subsequent elements later. */
29  public void add(int i, E e) throws IndexOutOfBoundsException,
30                                     IllegalStateException {
31      checkIndex(i, size + 1);
32      if (size == data.length)           // not enough capacity
33          throw new IllegalStateException("Array is full");
34      for (int k=size-1; k >= i; k--)     // start by shifting rightmost
35          data[k+1] = data[k];
36      data[i] = e;                       // ready to place the new element
37      size++;
38  }
39  /** Removes/returns the element at index i, shifting subsequent elements earlier. */
40  public E remove(int i) throws IndexOutOfBoundsException {
41      checkIndex(i, size);
42      E temp = data[i];
43      for (int k=i; k < size-1; k++)      // shift elements to fill hole
44          data[k] = data[k+1];
45      data[size-1] = null;               // help garbage collection
46      size--;
47      return temp;
48  }
49  // utility method
50  /** Checks whether the given index is in the range [0, n-1]. */
51  protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52      if (i < 0 || i >= n)
53          throw new IndexOutOfBoundsException("Illegal index: " + i);
54  }
55 }
```

Dynamic Arrays

- ❑ 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.
- ❑ To provide this abstraction, Java relies on an algorithmic sleight of hand that is known as a dynamic array.

Dynamic Arrays

- ❑ In reality, elements of an ArrayList are stored in a traditional array, and the precise size of that traditional array must be internally declared in order for the system to properly allocate a consecutive piece of memory for its storage.
- ❑ For example, Figure 7.2 displays an array with 12 cells that might be stored in memory locations 2146 through 2157 on a computer system.



Figure 7.2: An array of 12 cells, allocated in memory locations 2146 through 2157.

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.

Implementing a Dynamic Array

- The key is to provide means to “grow” the array A , when more space is needed.
- We cannot grow that array, as its capacity is fixed. Instead, when a call to add a new element risks overflowing the current array, we perform the following additional steps:
 - 1. Allocate a new array B with larger capacity.
 - 2. Set $B[k]=A[k]$, for $k=0, \dots, n-1$, 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.

Implementing a Dynamic Array

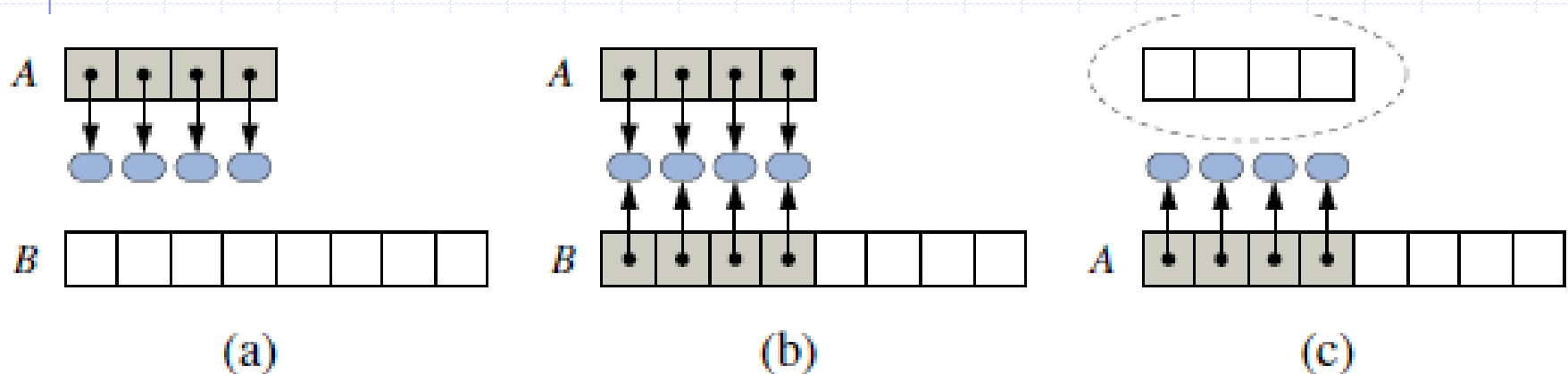


Figure 7.3: An illustration of “growing” a dynamic array: (a) create new array *B*; (b) store elements of *A* in *B*; (c) reassign reference *A* to the new array. Not shown is the future garbage collection of the old array, or the insertion of a new element.

An implementation of the ArrayList.resize method

```
/** 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.

Revision to the ArrayList.add method which calls the resize method

```
28  /** Inserts element e to be at index i, shifting all subsequent elements later. */
29  public void add(int i, E e) throws IndexOutOfBoundsException {
30      checkIndex(i, size + 1);
31      if (size == data.length)                // not enough capacity
32          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.

Growable Array-based Array List

- Let **push(o)** be the operation that adds element **o** at the end of the list *add (-1 + size) → 0*
- When the array is full, we replace the array with a larger one
- How large should the new array be?

- (A) **Incremental strategy**: increase the size by a constant c
- (B) **Doubling strategy**: double the size

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

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

Incremental Strategy Analysis

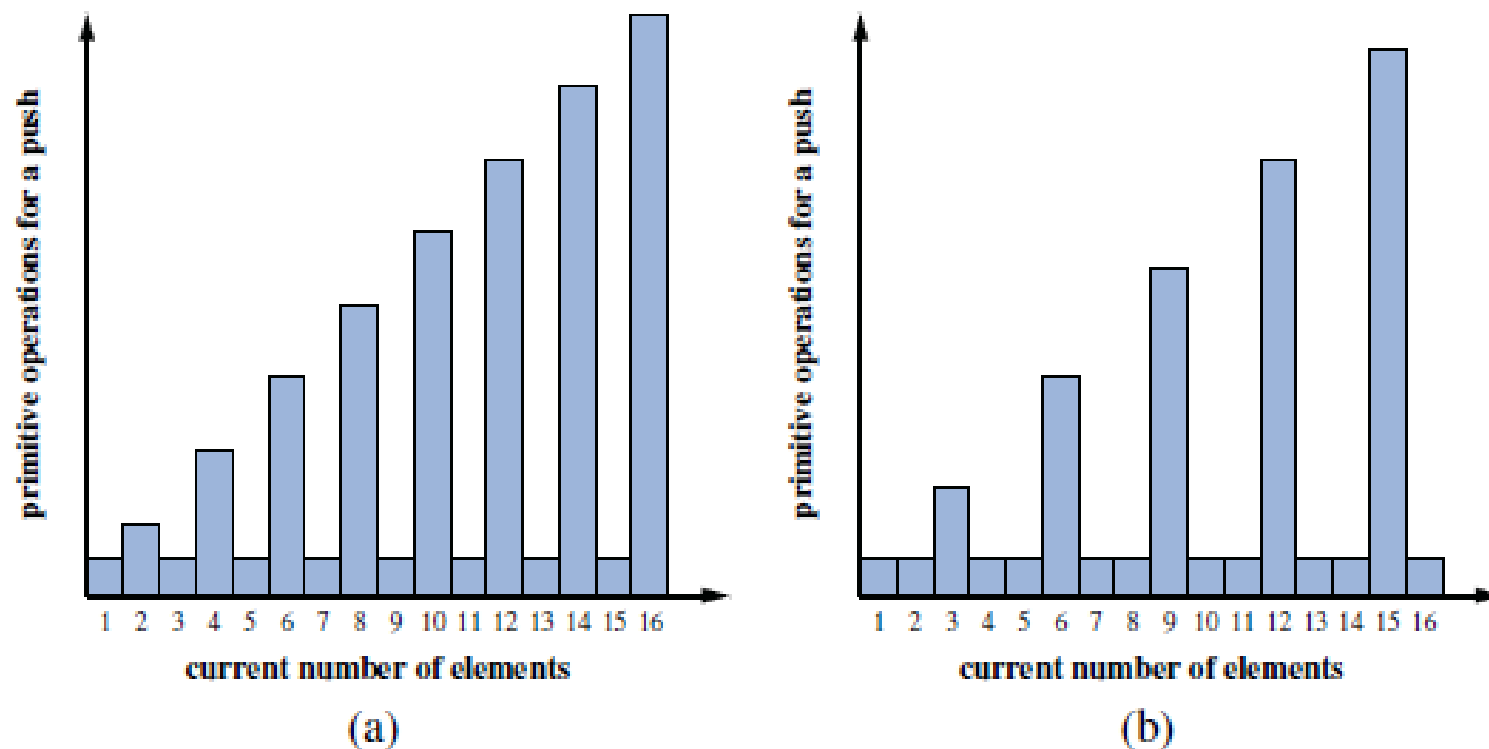


Figure 7.6: Running times of a series of push operations on a dynamic array using arithmetic progression of sizes. Part (a) assumes an increase of 2 in the size of the array, while part (b) assumes an increase of 3.

Incremental Strategy Analysis

- 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

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

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

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

- 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 $O(n)$

Incremental Strategy Analysis

Proposition 7.3: *Performing a series of n push operations on an initially empty dynamic array using a fixed increment with each resize takes $\Omega(n^2)$ time.*

Justification: Let $c > 0$ represent the fixed increment in capacity that is used for each resize event. During the series of n push operations, time will have been spent initializing arrays of size $c, 2c, 3c, \dots, mc$ for $m = \lceil n/c \rceil$, and therefore, the overall time is proportional to $c + 2c + 3c + \dots + mc$. By Proposition 4.3, this sum is

$$\sum_{i=1}^m ci = c \cdot \sum_{i=1}^m i = c \frac{m(m+1)}{2} \geq c \frac{\frac{n}{c}(\frac{n}{c} + 1)}{2} \geq \frac{1}{2c} \cdot n^2.$$

Therefore, performing the n push operations takes $\Omega(n^2)$ time. ■

Doubling Strategy Analysis

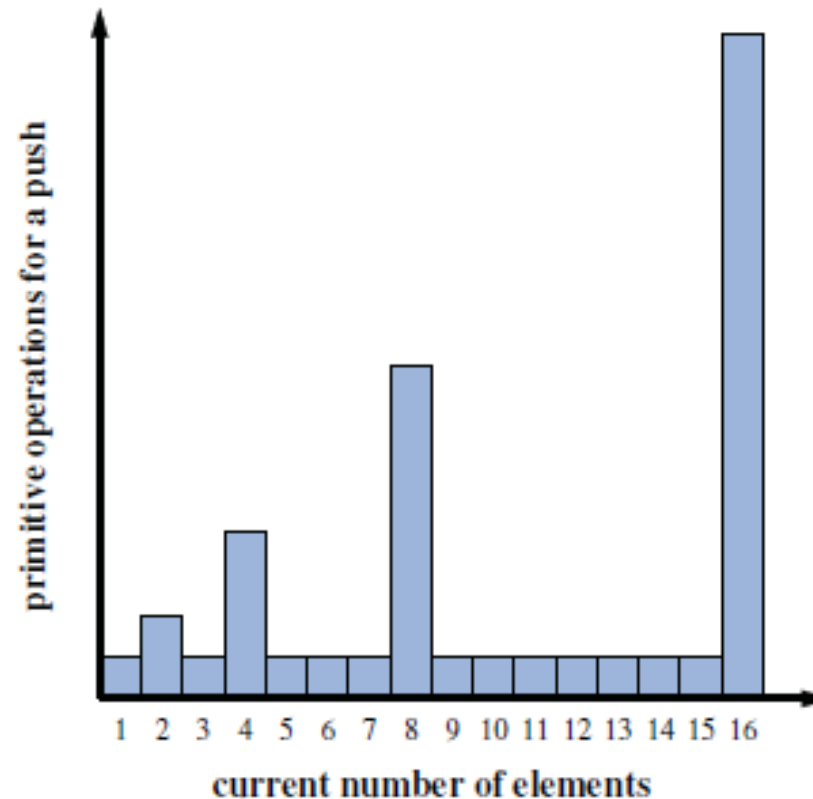


Figure 7.4: Running times of a series of push operations on a dynamic array.

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \dots + 2^k =$$

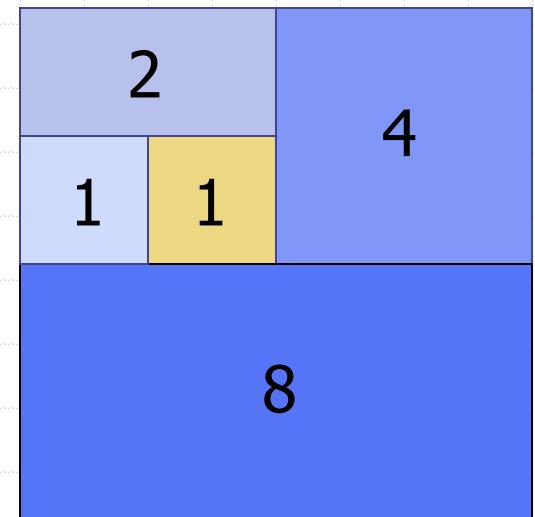
$$n + 2^{k+1} - 1 =$$

$$3n - 1$$

Handwritten annotations: A circle around $2^{k+1} - 1$ with an arrow pointing to $2^4 - 1$. To the right, a small diagram shows a stack of 15 items with arrows indicating a doubling process: $1 \rightarrow 2 \rightarrow 4 \rightarrow 8$, with a final arrow pointing to 15 and a $+1$ below it.

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$

geometric series



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:
 - $P.\text{getElement}()$: 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 .

A traversal of a positional list

- As a demonstration of a typical traversal of a positional list, Code Fragment 7.6 traverses a list, named `guests`, that stores string elements, and prints each element while traversing from the beginning of the list to the end.

```
1 Position<String> cursor = guests.first();
2 while (cursor != null) {
3     System.out.println(cursor.getElement());
4     cursor = guests.after(cursor);           // advance to the next position (if any)
5 }
```

Code Fragment 7.6: A traversal of a positional list.

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"	$(9_s, 7_p, 3_r)$

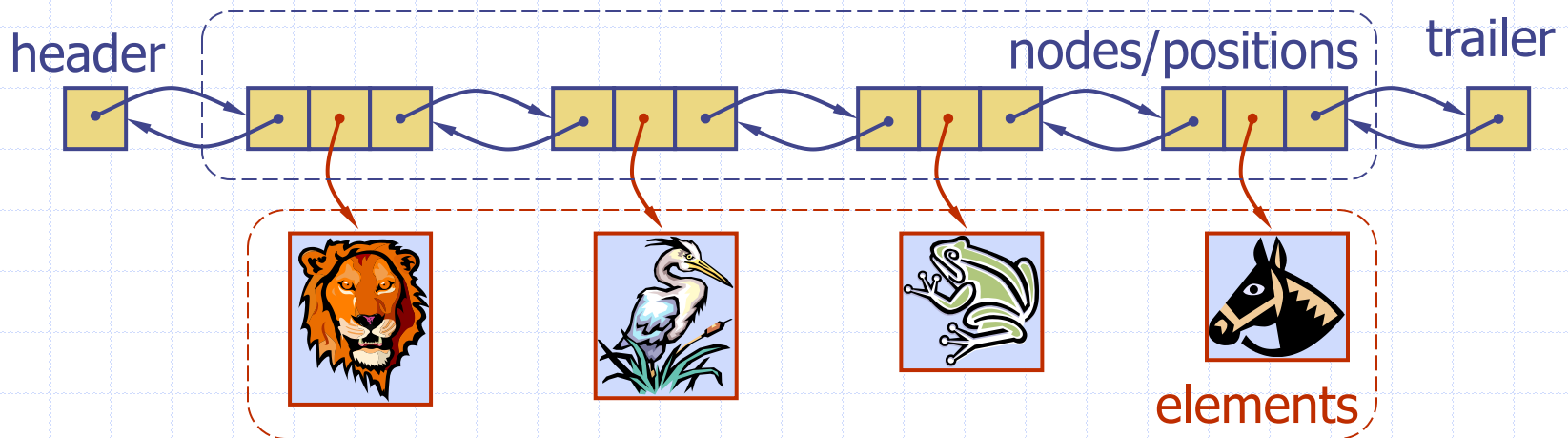
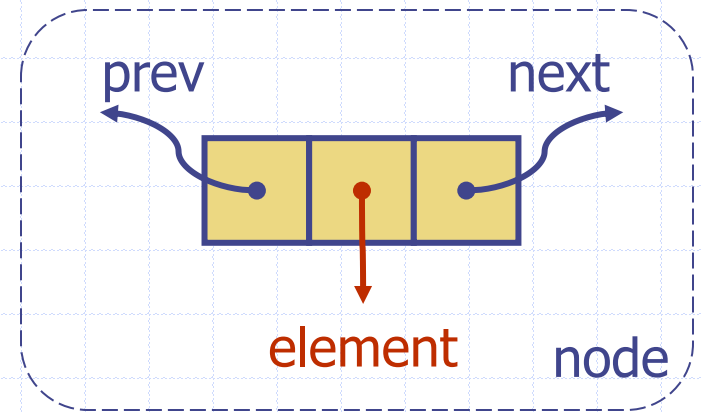
Java Interface Definitions-The Position interface

```
1 public interface Position<E> {  
2     /**  
3     * Returns the element stored at this position.  
4     *  
5     * @return the stored element  
6     * @throws IllegalStateException if position no longer valid  
7     */  
8     E getElement() throws IllegalStateException;  
9 }
```

Code Fragment 7.7: The Position interface.

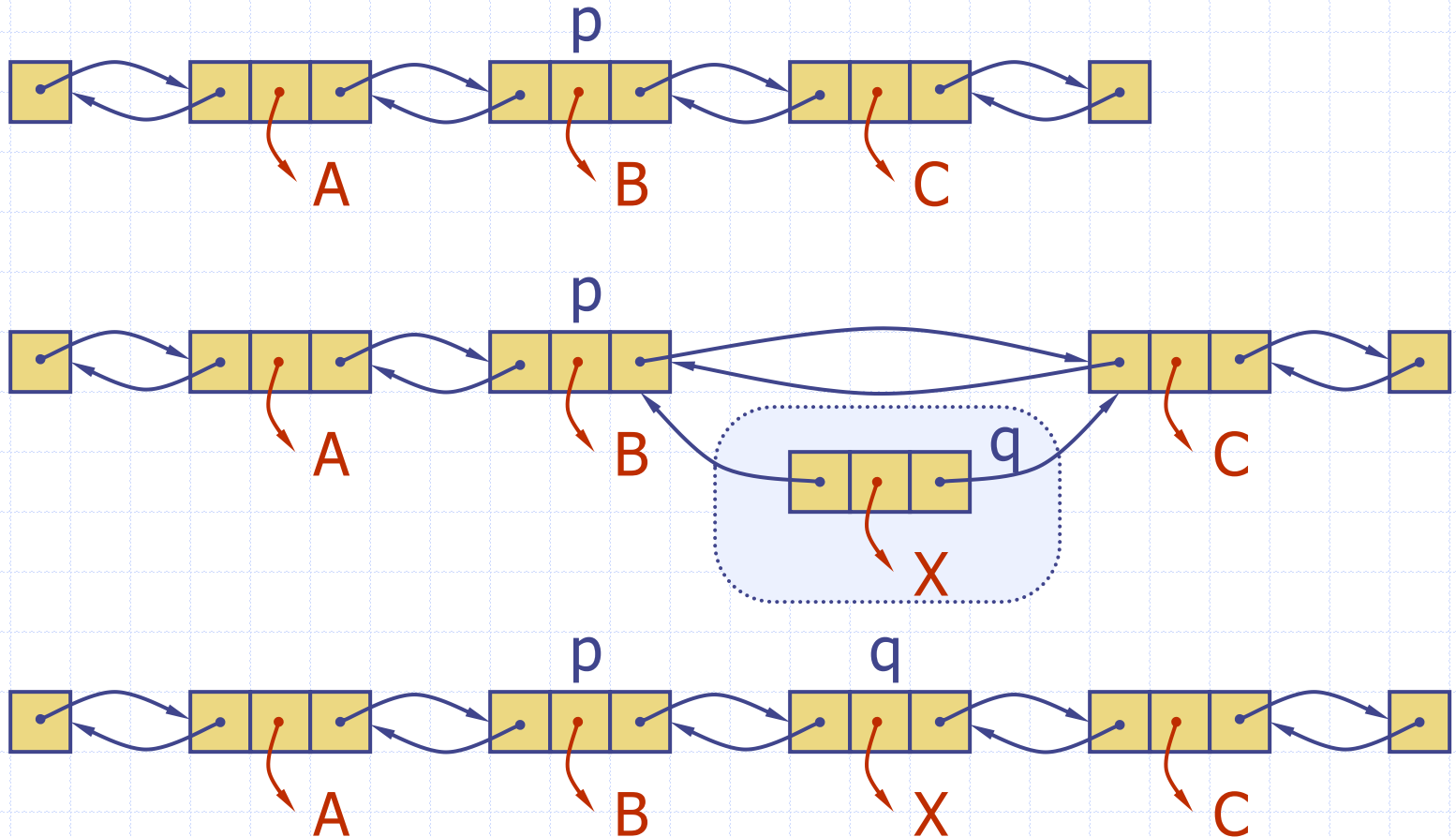
Positional List Implementation

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



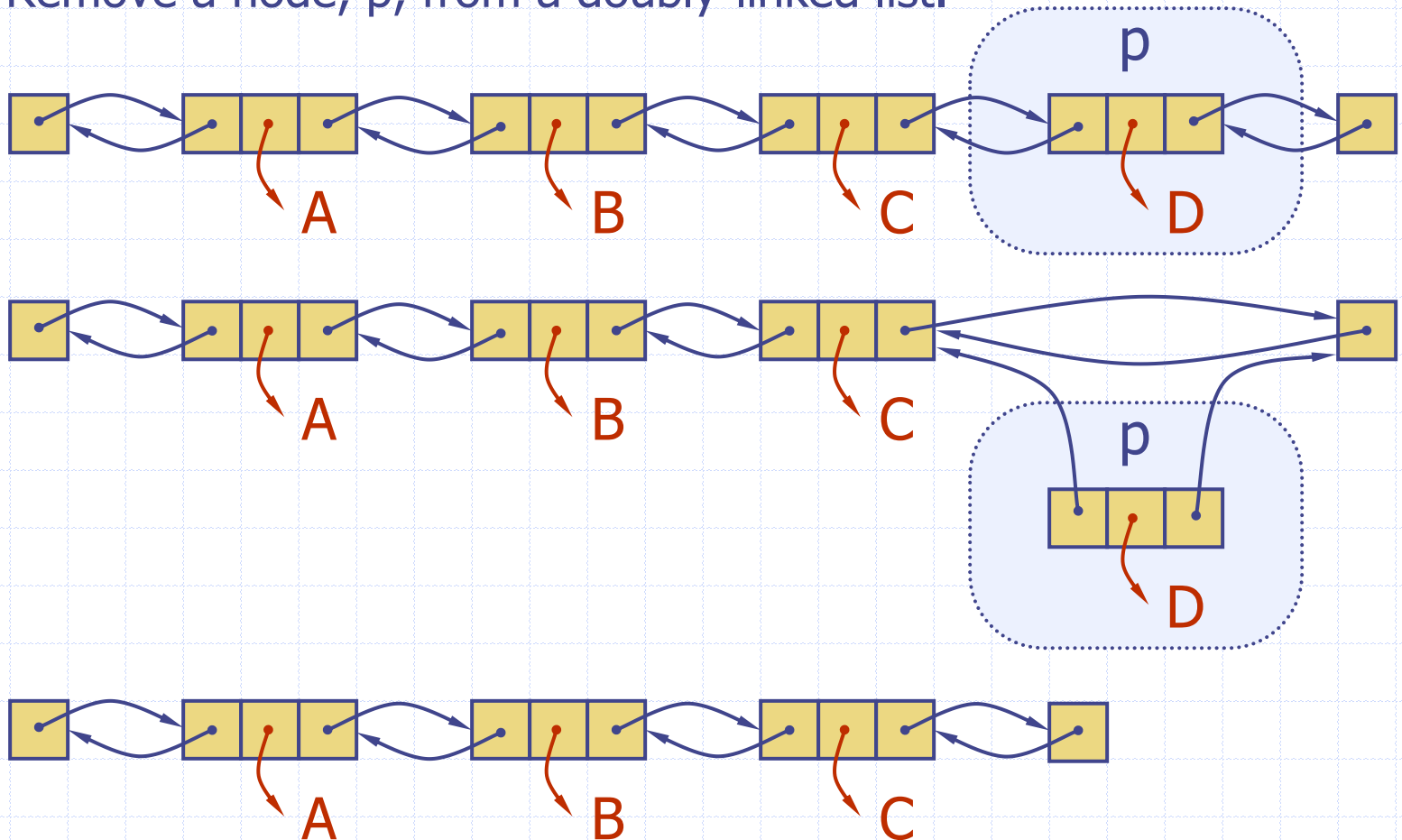
Insertion

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



Deletion

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



The Performance of a Linked Positional List

Method	Running Time
size()	$O(1)$
isEmpty()	$O(1)$
first(), last()	$O(1)$
before(p), after(p)	$O(1)$
addFirst(e), addLast(e)	$O(1)$
addBefore(p, e), addAfter(p, e)	$O(1)$
set(p, e)	$O(1)$
remove(p)	$O(1)$

Table 7.2: Performance of a positional list with n elements realized by a doubly linked list. The space usage is $O(n)$.

The PositionalList interface

```
1  /** An interface for positional lists. */
2  public interface PositionalList<E> {
3
4      /** Returns the number of elements in the list. */
5      int size();
6
7      /** Tests whether the list is empty. */
8      boolean isEmpty();
9
10     /** Returns the first Position in the list (or null, if empty). */
11     Position<E> first();
12
13     /** Returns the last Position in the list (or null, if empty). */
14     Position<E> last();
15
16     /** Returns the Position immediately before Position p (or null, if p is first). */
17     Position<E> before(Position<E> p) throws IllegalArgumentException;
18
19     /** Returns the Position immediately after Position p (or null, if p is last). */
20     Position<E> after(Position<E> p) throws IllegalArgumentException;
21
```

Code Fragment 7.8: The PositionalList interface.

The PositionalList interface (Cont.)

```
22  /** Inserts element e at the front of the list and returns its new Position. */
23  Position<E> addFirst(E e);
24
25  /** Inserts element e at the back of the list and returns its new Position. */
26  Position<E> addLast(E e);
27
28  /** Inserts element e immediately before Position p and returns its new Position. */
29  Position<E> addBefore(Position<E> p, E e)
30      throws IllegalArgumentException;
31
32  /** Inserts element e immediately after Position p and returns its new Position. */
33  Position<E> addAfter(Position<E> p, E e)
34      throws IllegalArgumentException;
35
36  /** Replaces the element stored at Position p and returns the replaced element. */
37  E set(Position<E> p, E e) throws IllegalArgumentException;
38
39  /** Removes the element stored at Position p and returns it (invalidating p). */
40  E remove(Position<E> p) throws IllegalArgumentException;
41  }
```

Code Fragment 7.8: The PositionalList interface.

Positional List-Doubly Linked List Implementation

```
1  /** Implementation of a positional list stored as a doubly linked list. */
2  public class LinkedPositionalList<E> implements PositionalList<E> {
3      //----- nested Node class -----
4      private static class Node<E> implements Position<E> {
5          private E element;           // reference to the element stored at this node
6          private Node<E> prev;        // reference to the previous node in the list
7          private Node<E> next;        // reference to the subsequent node in the list
8          public Node(E e, Node<E> p, Node<E> n) {
9              element = e;
10             prev = p;
11             next = n;
12         }
13     }
14 }
```

Code Fragment 7.9: An implementation of the `LinkedPositionalList` class.
(Continues in Code Fragments 7.10–7.12.)

Positional List-Doubly Linked List Implementation

```
13     public E getElement() throws IllegalStateException {
14         if (next == null) // convention for defunct node
15             throw new IllegalStateException("Position no longer valid");
16         return element;
17     }
18     public Node<E> getPrev() {
19         return prev;
20     }
21     public Node<E> getNext() {
22         return next;
23     }
24     public void setElement(E e) {
25         element = e;
26     }
27     public void setPrev(Node<E> p) {
28         prev = p;
29     }
30     public void setNext(Node<E> n) {
31         next = n;
32     }
33 } //----- end of nested Node class -----
```

Code Fragment 7.9: An implementation of the `LinkedPositionalList` class.
(Continues in Code Fragments 7.10–7.12.)

Positional List-Doubly Linked List Implementation

```
34
35 // instance variables of the LinkedPositionalList
36 private Node<E> header;           // header sentinel
37 private Node<E> trailer;         // trailer sentinel
38 private int size = 0;             // number of elements in the list
39
40 /** Constructs a new empty list. */
41 public LinkedPositionalList() {
42     header = new Node<>(null, null, null); // create header
43     trailer = new Node<>(null, header, null); // trailer is preceded by header
44     header.setNext(trailer); // header is followed by trailer
45 }
```

Code Fragment 7.9: An implementation of the `LinkedPositionalList` class.
(Continues in Code Fragments 7.10–7.12.)

Positional List-Doubly Linked List Implementation

```
46 // private utilities
47 /** Validates the position and returns it as a node. */
48 private Node<E> validate(Position<E> p) throws IllegalArgumentException {
49     if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
50     Node<E> node = (Node<E>) p; // safe cast
51     if (node.getNext() == null) // convention for defunct node
52         throw new IllegalArgumentException("p is no longer in the list");
53     return node;
54 }
55
56 /** Returns the given node as a Position (or null, if it is a sentinel). */
57 private Position<E> position(Node<E> node) {
58     if (node == header || node == trailer)
59         return null; // do not expose user to the sentinels
60     return node;
61 }
62
```

Code Fragment 7.10: An implementation of the `LinkedPositionalList` class.

(Continued from Code Fragment 7.9; continues in Code Fragments 7.11 and 7.12.)

Positional List-Doubly Linked List Implementation

```
63 // public accessor methods
64 /** Returns the number of elements in the linked list. */
65 public int size() { return size; }
66
67 /** Tests whether the linked list is empty. */
68 public boolean isEmpty() { return size == 0; }
69
70 /** Returns the first Position in the linked list (or null, if empty). */
71 public Position<E> first() {
72     return position(header.getNext());
73 }
74
75 /** Returns the last Position in the linked list (or null, if empty). */
76 public Position<E> last() {
77     return position(trailer.getPrev());
78 }
79
```

Code Fragment 7.10: An implementation of the `LinkedPositionalList` class.
(Continued from Code Fragment 7.9; continues in Code Fragments 7.11 and 7.12.)

Positional List-Doubly Linked List Implementation

```
80  /** Returns the Position immediately before Position p (or null, if p is first). */
81  public Position<E> before(Position<E> p) throws IllegalArgumentException {
82      Node<E> node = validate(p);
83      return position(node.getPrev());
84  }
85
86  /** Returns the Position immediately after Position p (or null, if p is last). */
87  public Position<E> after(Position<E> p) throws IllegalArgumentException {
88      Node<E> node = validate(p);
89      return position(node.getNext());
90  }
```

Code Fragment 7.10: An implementation of the `LinkedPositionalList` class.
(Continued from Code Fragment 7.9; continues in Code Fragments 7.11 and 7.12.)

Positional List-Doubly Linked List Implementation

```
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
101 // public update methods
102 /** Inserts element e at the front of the linked list and returns its new Position. */
103 public Position<E> addFirst(E e) {
104     return addBetween(e, header, header.getNext()); // just after the header
105 }
106
```

Code Fragment 7.11: An implementation of the `LinkedPositionalList` class.
(Continued from Code Fragments 7.9 and 7.10; continues in Code Fragment 7.12.)

Positional List-Doubly Linked List Implementation

```
107  /** Inserts element e at the back of the linked list and returns its new Position. */
108  public Position<E> addLast(E e) {
109      return addBetween(e, trailer.getPrev(), trailer);           // just before the trailer
110  }
111
112  /** Inserts element e immediately before Position p, and returns its new Position.*/
113  public Position<E> addBefore(Position<E> p, E e)
114      throws IllegalArgumentException {
115      Node<E> node = validate(p);
116      return addBetween(e, node.getPrev(), node);
117  }
118
119  /** Inserts element e immediately after Position p, and returns its new Position. */
120  public Position<E> addAfter(Position<E> p, E e)
121      throws IllegalArgumentException {
122      Node<E> node = validate(p);
123      return addBetween(e, node, node.getNext());
124  }
125
```

Code Fragment 7.11: An implementation of the `LinkedPositionalList` class.
(Continued from Code Fragments 7.9 and 7.10; continues in Code Fragment 7.12.)

Positional List-Doubly Linked List Implementation

```
126  /** Replaces the element stored at Position p and returns the replaced element. */
127  public E set(Position<E> p, E e) throws IllegalArgumentException {
128      Node<E> node = validate(p);
129      E answer = node.getElement();
130      node.setElement(e);
131      return answer;
132  }
```

Code Fragment 7.11: An implementation of the `LinkedPositionalList` class.
(Continued from Code Fragments 7.9 and 7.10; continues in Code Fragment 7.12.)

Positional List-Doubly Linked List Implementation

```
133  /** Removes the element stored at Position p and returns it (invalidating p). */
134  public E remove(Position<E> p) throws IllegalArgumentException {
135      Node<E> node = validate(p);
136      Node<E> predecessor = node.getPrev();
137      Node<E> successor = node.getNext();
138      predecessor.setNext(successor);
139      successor.setPrev(predecessor);
140      size--;
141      E answer = node.getElement();
142      node.setElement(null);           // help with garbage collection
143      node.setNext(null);             // and convention for defunct node
144      node.setPrev(null);
145      return answer;
146  }
147 }
```

Code Fragment 7.12: An implementation of the `LinkedPositionalList` class.
(Continued from Code Fragments 7.9–7.11.)

Case Study: InsertionSort

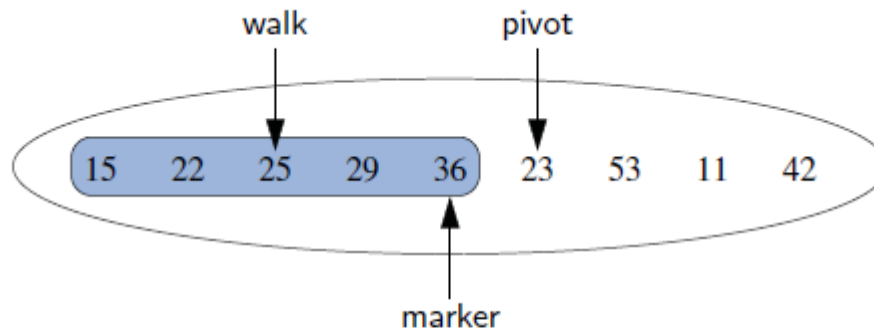


Figure 7.9: Overview of one step of our insertion-sort algorithm. The shaded elements, those up to and including marker, have already been sorted. In this step, the pivot's element should be relocated immediately before the walk position.

Case Study: InsertionSort

```
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 }
```

Code Fragment 7.15: Java code for performing insertion-sort on a positional list.

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.

Iterators

- ❑ The combination of these two methods allows a general loop construct for processing elements of the iterator.
- ❑ For example, if we let variable, `iter`, denote an instance of the `Iterator<String>` type, then we can write the following:

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

Iterators

- The `java.util.Iterator` interface contains a third method, which is optionally supported by some iterators:

`remove( )`: Removes from the collection the element returned by the most recent call to `next( )`.

Throws an `IllegalStateException` if `next` has not yet been called, or if `remove` was already called since the most recent call to `next`.

- This method can be used to filter a collection of elements, for example to discard all negative numbers from a data set.

The Iterable Interface

- 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 no way to “reset” the iterator back to the beginning of the sequence.
- However, a data structure that wishes to allow repeated iterations can support a method that returns a new iterator, each time it is called.
- To provide greater standardization, Java defines another parameterized interface, named Iterable.

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:

```
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"  
}
```

Filtering an Iterator-Example

The following loop can be used to remove all negative numbers from an ArrayList of floating-point values.

```
ArrayList<Double> data; // populate with random  
numbers (not shown)
```

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