Implementations I
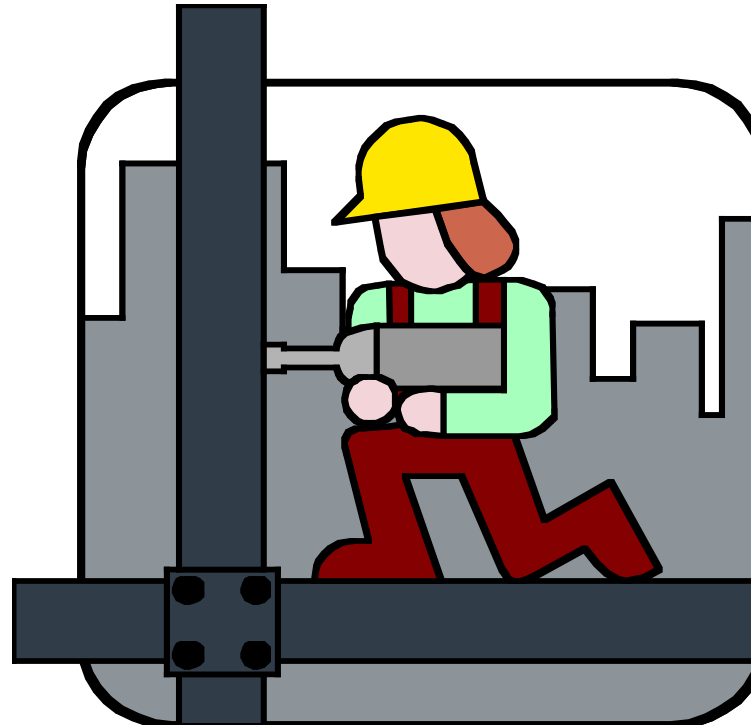
Agenda

- **Inner classes and implementation of `ArrayList`**
    Nested classes and inner classes
    The `AbstractCollection` class
    Implementation of `ArrayList`

- **Stack and queues**
    Array-based implementations
    Linked list-based implementations

- **Linked lists**
    Doubly linked lists, circularly linked lists,
    sorted linked lists

Iterator design using nested class

Notice the `static`
specification

```
1  package weiss.ds;
2
3  public class MyContainer
4  {
5      private Object [ ] items;
6      private int size = 0;
7      // Other methods for MyContainer not shown
8
9      public Iterator iterator( )
10       { return new LocalIterator( this ); }
11
12     // The iterator class as a nested class
13     private static class LocalIterator implements Iterator
14     {
15         private int current = 0;
16         private MyContainer container;
17
18         private LocalIterator( MyContainer c )
19           { container = c; }
20
21         public boolean hasNext( )
22           { return current < container.size; }
23
24         public Object next( )
25           { return container.items[ current++ ]; }
26     }
27 }
```

Iterator design using inner class

```
1  package weiss.ds;
2
3  public class MyContainer
4  {
5      private Object [ ] items;
6      private int size = 0;
7
8      // Other methods for MyContainer not shown
9
10     public Iterator iterator( )
11       { return new LocalIterator( ); }
12
13     // The iterator class as an inner class
14     private class LocalIterator implements Iterator
15     {
16         private int current = 0;
17
18         public boolean hasNext( )
19           { return current < MyContainer.this.size; }
20
21         public Object next( )
22           { return MyContainer.this.items[ current++ ]; }
23     }
24  }
```

4

```
1    // The iterator class as an inner class
2    private class LocalIterator implements Iterator
3    {
4        private int current = 0;
5
6        public boolean hasNext( )
7          { return current < size; }
8
9        public Object next( )
10         { return items[ current++ ]; }
11    }
```

figure 15.9

Inner class;
Outer.this may be
optional.

5

Nested classes and inner classes

A **static nested class** interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

As with instance methods and variables, an **inner class** is associated with an instance of its enclosing class and has direct access to that object's methods and fields.

Instance of InnerClass

Instance of OuterClass
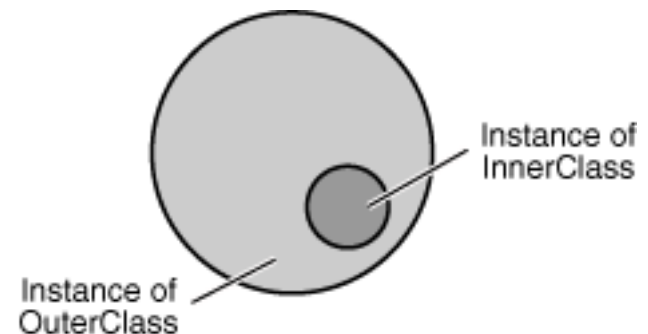
Abstract collections

A **set** is an unordered collection of elements. No duplicates are allowed.

A **list** is an ordered collection of elements. Duplicates are allowed. Lists are also known an *sequences*.

A map is an unordered collection of key-value pairs. The keys must be unique. Maps are also known as *dictionaries*.

Interfaces for collections

java.util.*
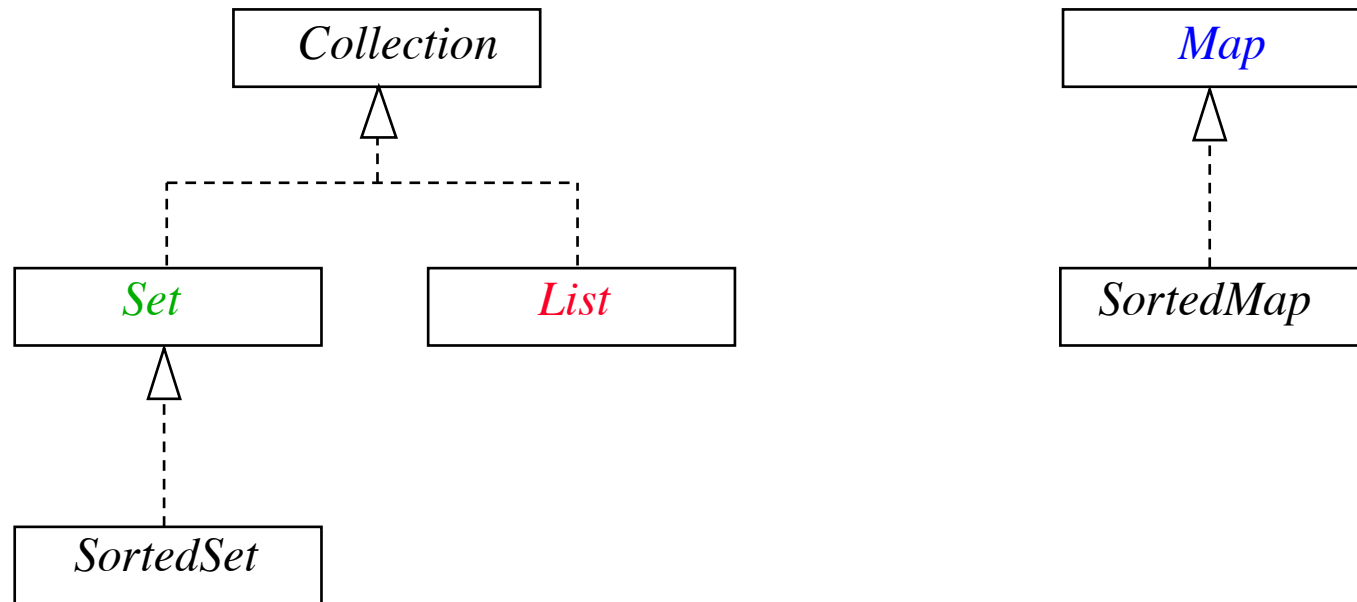
interface **Collection<E>**

```
boolean add(E o)
boolean addAll(Collection<? extends E> c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int size()
Object[] toArray()
<T> T[] toArray(T[] a)
```

```java
1   package weiss.util;
2
3   /**
4    * AbstractCollection provides default implementations for
5    * some of the easy methods in the Collection interface.
6    */
7   public abstract class AbstractCollection<AnyType> implements Collection<AnyType>
8   {
9       /**
10       * Tests if this collection is empty.
11       * @return true if the size of this collection is zero.
12       */
13      public boolean isEmpty( )
14      {
15          return size( ) == 0;
16      }
17
18      /**
19       * Change the size of this collection to zero.
20       */
21      public void clear( )
22      {
23          Iterator<AnyType> itr = iterator( );
24          while( itr.hasNext( ) )
25          {
26              itr.next( );
27              itr.remove( );
28          }
29      }
30
31      /**
32       * Adds x to this collections.
33       * This default implementation always throws an exception.
34       * @param x the item to add.
35       * @throws UnsupportedOperationException always.
36       */
37      public boolean add( AnyType x )
38      {
39          throw new UnsupportedOperationException( );
40      }
```

figure 15.10

Sample implementation of `AbstractCollection` (part 1) of 3

10

figure 15.11

Sample
implementation of
AbstractCollection
(part 2)

```
41    /**
42     * Returns true if this collection contains x.
43     * If x is null, returns false.
44     * (This behavior may not always be appropriate.)
45     * @param x the item to search for.
46     * @return true if x is not null and is found in
47     * this collection.
48     */
49    public boolean contains( Object x )
50    {
51        if( x == null )
52            return false;
53
54        for( AnyType val : this )
55            if( x.equals( val ) )
56                return true;
57
58        return false;
59    }
60
61    /**
62     * Removes non-null x from this collection.
63     * (This behavior may not always be appropriate.)
64     * @param x the item to remove.
65     * @return true if remove succeeds.
66     */
67    public boolean remove( Object x )
68    {
69        if( x == null )
70            return false;
71
72        Iterator itr = iterator( );
73        while( itr.hasNext( ) )
74            if( x.equals( itr.next( ) ) )
75            {
76                itr.remove( );
77                return true;
78            }
79
80        return false;
81    }
```

11

```
82      /**
83       * Obtains a primitive array view of the collection.
84       * @return the primitive array view.
85       */
86      public Object [ ] toArray( )
87      {
88          Object [ ] copy = new Object[ size( ) ];
89          int i = 0;
90
91          for( AnyType val : this )
92              copy[ i++ ] = val;
93
94          return copy;
95      }
96
97      public <OtherType> OtherType [ ] toArray( OtherType [ ] arr )
98      {
99          int theSize = size( );
100
101         if( arr.length < theSize )
102             arr = ( OtherType [ ] ) java.lang.reflect.Array.newInstance(
103                            arr.getClass( ).getComponentType( ), theSize );
104         else if( theSize < arr.length )
105             arr[ theSize ] = null;
106
107         Object [ ] copy = arr;
108         int i = 0;
109
110         for( AnyType val : this )
111             copy[ i++ ] = val;
112
113         return copy;
114     }
115
116     /**
117      * Return a string representation of this collection.
118      */
119     public String toString( )
120     {
121         StringBuilder result = new StringBuilder( "[ " );
122
123         for( AnyType obj : this )
124             result.append( obj + " " );
125
126         result.append( "]" );
127
128         return result.toString( );
129     }
130 }
```

Flaw

A Collection may contain itself!

figure 15.12

Sample implementation of `AbstractCollection` (part 3)

12

toString as implemented in **java.util.AbstractCollection**

```java
public String toString() {
    Iterator<E> it = iterator();
    if (! it.hasNext())
        return "[]";

    StringBuilder sb = new StringBuilder();
    sb.append('[');
    for (;;) {
        E e = it.next();
        sb.append(e == this ? "(this Collection)" : e);
        if (! it.hasNext())
            return sb.append(']').toString();
        sb.append(',').append(' ');
    }
}
```

ArrayList

An array list (`ArrayList`) is a list that uses an array to store its elements.

italic: interface
normal: class

```
Collection
```

```
AbstractCollection    List
```

```
ArrayList
```

implements

extends

14

figure 15.13
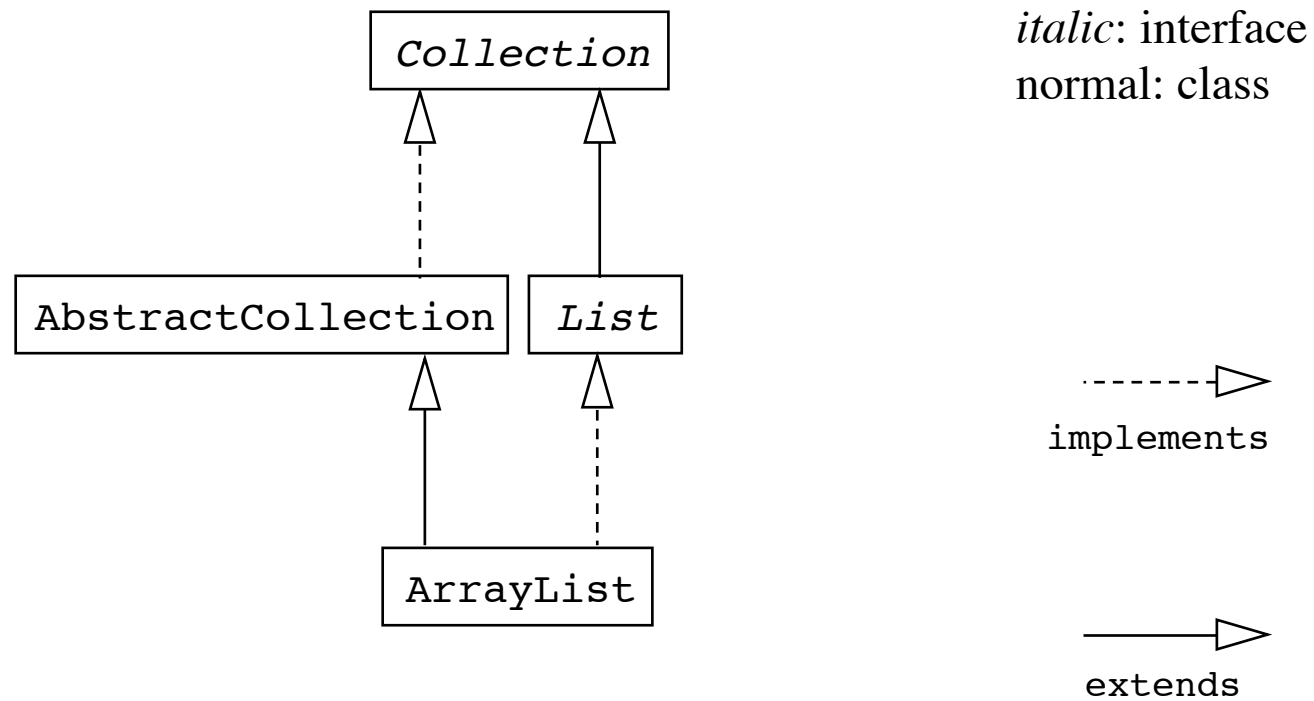
ArrayList
implementation
(part 1) of 3

```
 1  package weiss.util;
 2
 3  public class ArrayList<AnyType> extends AbstractCollection<AnyType>
 4                                  implements List<AnyType>
 5  {
 6      private static final int DEFAULT_CAPACITY = 10;
 7      private static final int NOT_FOUND = -1;
 8
 9      private AnyType [ ] theItems;
10      private int theSize;
11      private int modCount = 0;
12
13      public ArrayList( )
14        { clear( ); }
15
16      public ArrayList( Collection<AnyType> other )
17      {
18          clear( );
19          for( AnyType obj : other )
20              add( obj );
21      }
22
23      public int size( )
24        { return theSize; }
25
26      public void clear( )
27      {
28          theSize = 0;
29          theItems = (AnyType []) new Object[ DEFAULT_CAPACITY ];
30          modCount++;
31      }
32
33      public AnyType get( int idx )
34      {
35          if( idx < 0 || idx >= size( ) )
36              throw new ArrayIndexOutOfBoundsException( );
37          return theItems[ idx ];
38      }
39
40      public AnyType set( int idx, AnyType newVal )
41      {
42          if( idx < 0 || idx >= size( ) )
43              throw new ArrayIndexOutOfBoundsException( );
44          AnyType old = theItems[ idx ];
45          theItems[ idx ] = newVal;
46
47          return old;
48      }
49
50      public boolean contains( Object x )
51        { return findPos( x ) != NOT_FOUND; }
```

modCount represents the number of structural modifications (adds, removes) made to the ArrayList.

The idea is that when an iterator is constructed, the iterator saves this value in its data member expectedModCount.

When any iterator operation is performed, the iterator's expectedModCount member is compared with the ArrayList's modCount, and if they disagree, a ConcurrentModificationException can be thrown.

15

```java
52    private int findPos( Object x )
53    {
54        for( int i = 0; i < size( ); i++ )
55            if( x == null )
56            {
57                if( theItems[ i ] == null )
58                    return i;
59            }
60            else if( x.equals( theItems[ i ] ) )
61                return i;
62
63        return NOT_FOUND;
64    }
65
66    public boolean add( AnyType x )
67    {
68        if( theItems.length == size( ) )
69        {
70            AnyType [ ] old = theItems;
71            theItems = (AnyType []) new Object[ theItems.length * 2 + 1 ];
72            for( int i = 0; i < size( ); i++ )
73                theItems[ i ] = old[ i ];
74        }
75        theItems[ theSize++ ] = x;
76        modCount++;
77        return true;
78    }
79
80    public boolean remove( Object x )
81    {
82        int pos = findPos( x );
83
84        if( pos == NOT_FOUND )
85            return false;
86        else
87        {
88            remove( pos );
89            return true;
90        }
91    }
92
93    public AnyType remove( int idx )
94    {
95        AnyType removedItem = theItems[ idx ];
96        for( int i = idx; i < size( ) - 1; i++ )
97            theItems[ i ] = theItems[ i + 1 ];
98        theSize--;
99        modCount++;
100       return removedItem;
101   }
```

figure 15.14

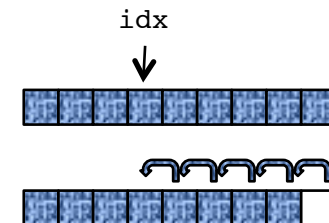ArrayList
implementation
(part 2)

idx

figure 15.15

ArrayList
implementation
(part 3)

listIterator(int idx):
Returns a list iterator of the
elements in this list (in proper
sequence), starting at the specified
position in this list. The specified
index indicates the first element
that would be returned by an initial
call to the next method. An initial
call to the previous method
would return the element with the
specified index minus one.

```java
102    public Iterator<AnyType> iterator( )
103      { return new ArrayListIterator( 0 ); }
104
105    public ListIterator<AnyType> listIterator( int idx )
106      { return new ArrayListIterator( idx ); }
107
108    // This is the implementation of the ArrayListIterator
109    private class ArrayListIterator implements ListIterator<AnyType>
110    {
111        private int current;
112        private int expectedModCount = modCount;
113        private boolean nextCompleted = false;
114        private boolean prevCompleted = false;
115
116        ArrayListIterator( int pos )
117        {
118            if( pos < 0 || pos > size( ) )
119                throw new IndexOutOfBoundsException( );
120            current = pos;
121        }
122
123        public boolean hasNext( )
124        {
125            if( expectedModCount != modCount )
126                throw new ConcurrentModificationException( );
127            return current < size( );
128        }
129
130        public boolean hasPrevious( )
131        {
132            if( expectedModCount != modCount )
133                throw new ConcurrentModificationException( );
134            return current > 0;
135        }
```

17

figure 15.16

ArrayList
implementation
(part 4)

```
136        public AnyType next( )
137        {
138            if( !hasNext( ) )
139                throw new NoSuchElementException( );
140            nextCompleted = true;
141            prevCompleted = false;
142            return theItems[ current++ ];
143        }
144
145        public AnyType previous( )
146        {
147            if( !hasPrevious( ) )
148                throw new NoSuchElementException( );
149            prevCompleted = true;
150            nextCompleted = false;
151            return theItems[ --current ];
152        }
153
154        public void remove( )
155        {
156            if( expectedModCount != modCount )
157                throw new ConcurrentModificationException( );
158
159            if( nextCompleted )
160                ArrayList.this.remove( --current );
161            else if( prevCompleted )
162                ArrayList.this.remove( current );
163            else
164                throw new IllegalStateException( );
165
166            prevCompleted = nextCompleted = false;
167            expectedModCount++;
168        }
169    }
170 }
```

Does either of these proposed implementations of `clear` for `AbstractCollection` work?

```
 1  public void clear( )    // Version #1
 2  {
 3      Iterator<AnyType> itr = iterator( );
 4      while( !isEmpty( ) )
 5          remove( itr.next( ) );
 6  }
 7
 8  public void clear( )    // Version #2
 9  {
10      while( !isEmpty( ) )
11          remove( iterator( ).next( ) );
12  }
```

Proposed implementations of clear for AbstractCollection

Version #1 will throw a `ConcurrentModificationException`

Version #2 works

19

Does this proposed implementation of `clear` work?
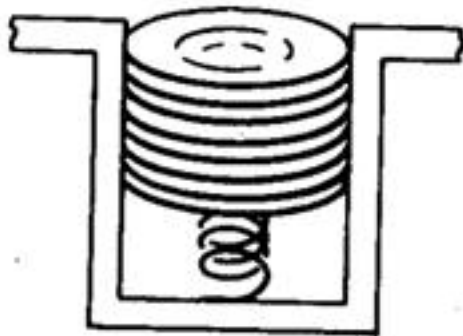
```
public void clear( )    // Version #3
{
    Iterator<AnyType> itr = iterator( );
    while( itr.hasNext( ) )
        itr.remove( );
}
```

Version #3 will throw an `IllegalStateException`
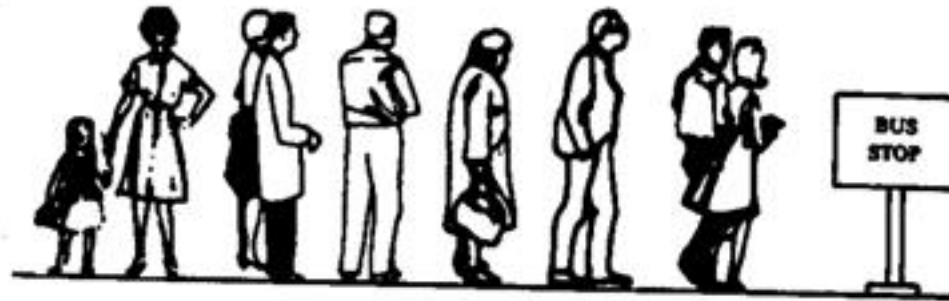
```
public void clear( )    // Version #4
{
    Iterator<AnyType> itr = iterator( );
    while( itr.hasNext( ) )
        itr.next( ).remove( );
}
```

Version #4 works

Stacks and queues

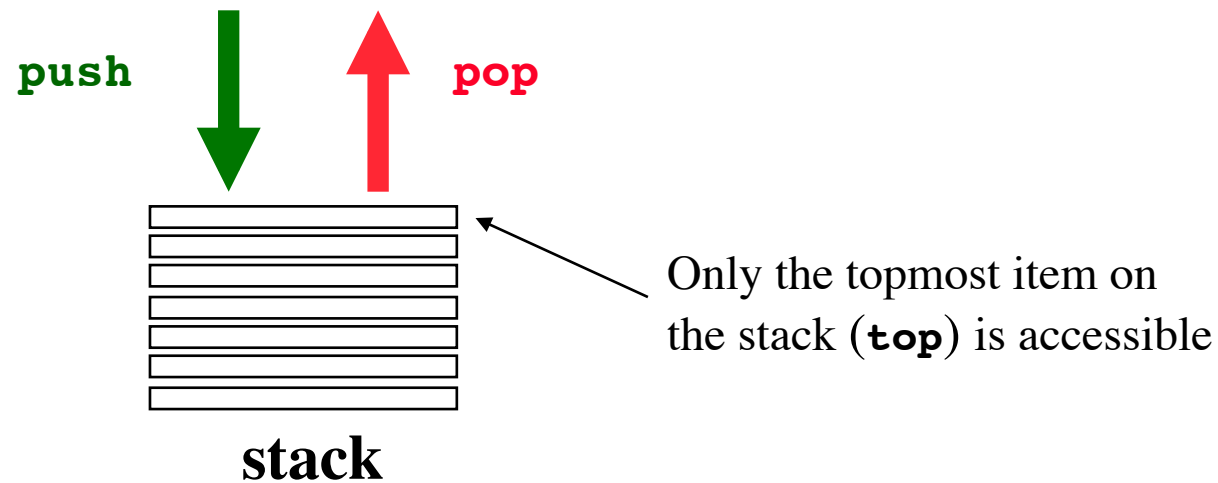(a) Stack of dishes.

(b) Queue waiting for a bus.

BUS STOP

Stack

(**LIFO** = **L**ast**I**n**F**irst**O**ut)

A **stack** is a sequence of items of the same type that provides the following two operations:

push(x): Add the item **x** to the top of the stack

pop: Remove the top item from the stack

push ↓ **pop** ↑

Only the topmost item on the stack (**top**) is accessible

stack

Stack implemented with an array

figure 16.1

How the stack
routines work:
(a) empty stack;
(b) push(a);
(c) push(b);
(d) pop( )

(a)   tos (-1)

(b)   a   tos (0)

(c)   b   tos (1)
       a

(d)   a   tos (0)
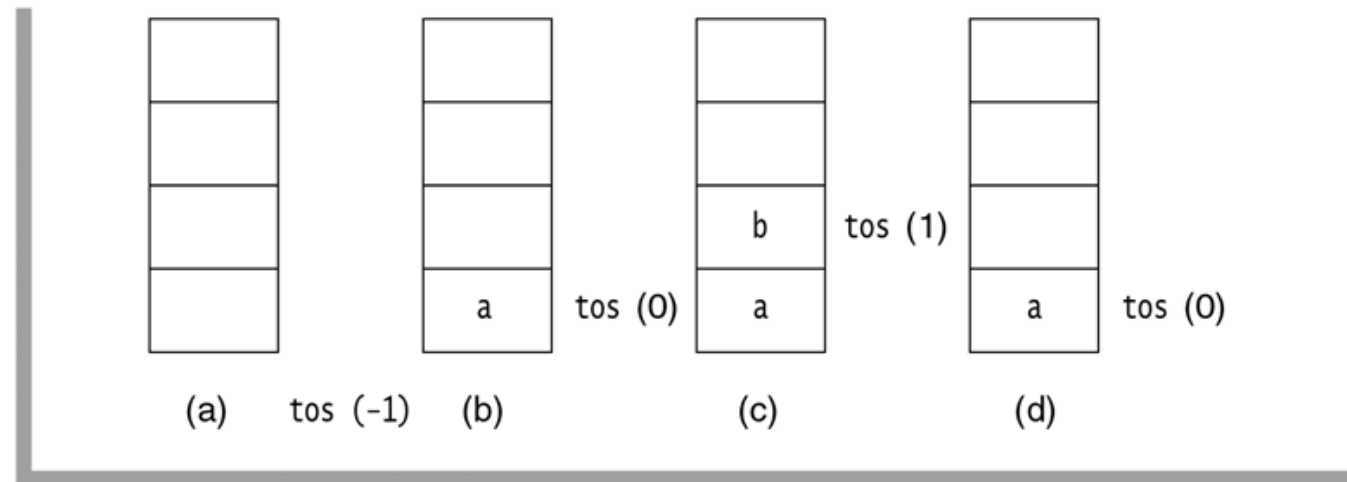
The integer `tos` (*top of stack*) provides the array index of the top element of the stack

23

figure 16.2

Skeleton for the
array-based stack
class

```
1  package weiss.nonstandard;
2
3  // ArrayStack class
4  //
5  // CONSTRUCTION: with no initializer
6  //
7  // ******************PUBLIC OPERATIONS*********************
8  // void push( x )          --> Insert x
9  // void pop( )             --> Remove most recently inserted item
10 // AnyType top( )          --> Return most recently inserted item
11 // AnyType topAndPop( )    --> Return and remove most recent item
12 // boolean isEmpty( )      --> Return true if empty; else false
13 // void makeEmpty( )       --> Remove all items
14 // ******************ERRORS********************************
15 // top, pop, or topAndPop on empty stack
16
17 public class ArrayStack<AnyType> implements Stack<AnyType>
18 {
19     public ArrayStack( )
20       { /* Figure 16.3 */ }
21
22     public boolean isEmpty( )
23       { /* Figure 16.4 */ }
24     public void makeEmpty( )
25       { /* Figure 16.4 */ }
26     public Object top( )
27       { /* Figure 16.6 */ }
28     public void pop( )
29       { /* Figure 16.6 */ }
30     public AnyType topAndPop( )
31       { /* Figure 16.7 */ }
32     public void push( AnyType x )
33       { /* Figure 16.5 */ }
34
35     private void doubleArray( )
36       { /* Implementation in online code  */ }
37
38     private AnyType [ ] theArray;
39     private int        topOfStack;
40
41     private static final int DEFAULT_CAPACITY = 10;
42 }
```

24

figure 16.3

The zero-parameter constructor for the ArrayStack class

```
1    /**
2     * Construct the stack.
3     */
4    public ArrayStack( )
5    {
6        theArray = (AnyType []) new Object[ DEFAULT_CAPACITY ];
7        topOfStack = -1;
8    }
```

figure 16.4

The isEmpty and makeEmpty routines for the ArrayStack class

```
1    /**
2     * Test if the stack is logically empty.
3     * @return true if empty, false otherwise.
4     */
5    public boolean isEmpty( )
6    {
7        return topOfStack == -1;
8    }
9
10   /**
11    * Make the stack logically empty.
12    */
13   public void makeEmpty( )
14   {
15       topOfStack = -1;
16   }
```

figure 16.5

The push method for
the ArrayStack class

```
1      /**
2       * Insert a new item into the stack.
3       * @param x the item to insert.
4       */
5      public void push( AnyType x )
6      {
7          if( topOfStack + 1 == theArray.length )
8              doubleArray( );
9          theArray[ ++topOfStack ] = x;
10     }
```

```
1     /**
2      * Get the most recently inserted item in the stack.
3      * Does not alter the stack.
4      * @return the most recently inserted item in the stack.
5      * @throws UnderflowException if the stack is empty.
6      */
7     public AnyType top( )
8     {
9         if( isEmpty( ) )
10            throw new UnderflowException( "ArrayStack top" );
11        return theArray[ topOfStack ];
12    }
13
14    /**
15     * Remove the most recently inserted item from the stack.
16     * @throws UnderflowException if the stack is empty.
17     */
18    public void pop( )
19    {
20        if( isEmpty( ) )
21            throw new UnderflowException( "ArrayStack pop" );
22        topOfStack--;
23    }
```

```
1    /**
2     * Return and remove the most recently inserted item
3     * from the stack.
4     * @return the most recently inserted item in the stack.
5     * @throws Underflow if the stack is empty.
6     */
7    public AnyType topAndPop( )
8    {
9        if( isEmpty( ) )
10           throw new UnderflowException( "ArrayStack topAndPop" );
11       return theArray[ topOfStack-- ];
12   }
```

figure 16.7

The topAndPop method for the ArrayStack class

Amortized running time

Array doubling is expensive in running time: $O(N)$.

However, array doubling is infrequent because an array doubling that involves N elements must be preceded by at least $N/2$ pushes that do not involve an array doubling.

Consequently we can charge the $O(N)$ cost of the doubling over these $N/2$ easy pushes, thereby effectively raising the cost of each push by only a small constant.

In the "long run" `push` runs in $O(1)$ time.

Amortization: The paying off of debt in regular installments over a period of time.

Stack implemented with a linked list

topOfStack

ListNode
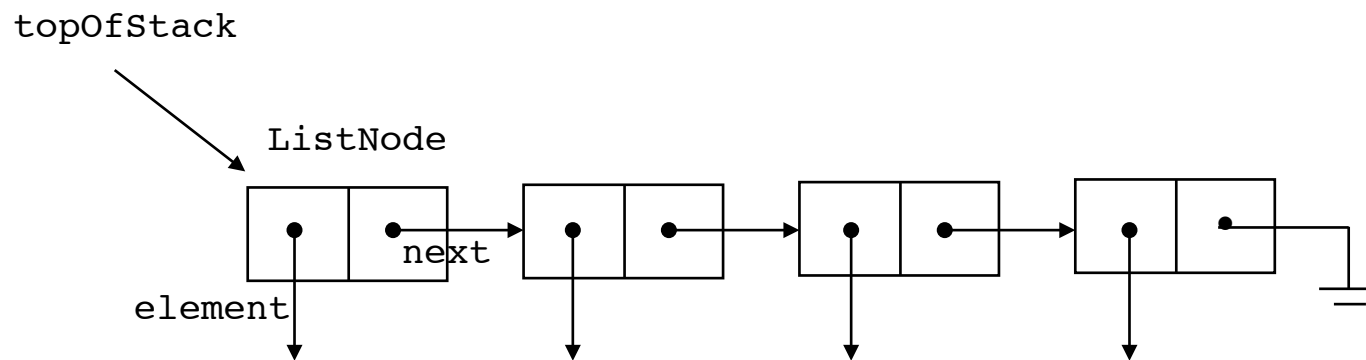
next

element

```
class ListNode<AnyType> {
    ListNode(AnyType e, ListNode n) { element = e; next = n; }
    AnyType element;
    ListNode next;
}
```

```
 1  package weiss.nonstandard;
 2
 3  // ListStack class
 4  //
 5  // CONSTRUCTION: with no initializer
 6  //
 7  // ******************PUBLIC OPERATIONS********************
 8  // void push( x )          --> Insert x
 9  // void pop( )             --> Remove most recently inserted item
10  // AnyType top( )          --> Return most recently inserted item
11  // AnyType topAndPop( )    --> Return and remove most recent item
12  // boolean isEmpty( )      --> Return true if empty; else false
13  // void makeEmpty( )       --> Remove all items
14  // ****************ERRORS********************************
15  // top, pop, or topAndPop on empty stack
16
17  public class ListStack<AnyType> implements Stack<AnyType>
18  {
19      public boolean isEmpty( )
20        { return topOfStack == null; }
21      public void makeEmpty( )
22        { topOfStack = null; }
23
24      public void push( AnyType x )
25        { /* Figure 16.20 */ }
26      public void pop( )
27        { /* Figure 16.20 */ }
28      public AnyType top( )
29        { /* Figure 16.21 */ }
30      public AnyType topAndPop( )
31        { /* Figure 16.21 */ }
32
33      private ListNode<AnyType> topOfStack = null;
34  }
35
36  // Basic node stored in a linked list.
37  // Note that this class is not accessible outside
38  // of package weiss.nonstandard
39  class ListNode<AnyType>
40  {
41      public ListNode( AnyType theElement )
42        { this( theElement, null ); }
43
44      public ListNode( AnyType theElement, ListNode<AnyType> n )
45        { element = theElement; next = n; }
46
47      public AnyType    element;
48      public ListNode next;
49  }
```

figure 16.19

Skeleton for
linked list-based
stack class

figure 16.20

The push and pop
routines for the
ListStack class

```
1        /**
2         * Insert a new item into the stack.
3         * @param x the item to insert.
4         */
5        public void push( AnyType x )
6        {
7            topOfStack = new ListNode<AnyType>( x, topOfStack );
8        }
9
10       /**
11        * Remove the most recently inserted item from the stack.
12        * @throws UnderflowException if the stack is empty.
13        */
14       public void pop( )
15       {
16           if( isEmpty( ) )
17               throw new UnderflowException( "ListStack pop" );
18           topOfStack = topOfStack.next;
19       }
```

figure 16.21

The top and topAndPop
routines for the
ListStack class

```
1      /**
2       * Get the most recently inserted item in the stack.
3       * Does not alter the stack.
4       * @return the most recently inserted item in the stack.
5       * @throws UnderflowException if the stack is empty.
6       */
7      public AnyType top( )
8      {
9          if( isEmpty( ) )
10             throw new UnderflowException( "ListStack top" );
11         return topOfStack.element;
12     }
13
14     /**
15      * Return and remove the most recently inserted item
16      * from the stack.
17      * @return the most recently inserted item in the stack.
18      * @throws UnderflowException if the stack is empty.
19      */
20     public AnyType topAndPop( )
21     {
22         if( isEmpty( ) )
23             throw new UnderflowException( "ListStack topAndPop" );
24
25         AnyType topItem = topOfStack.element;
26         topOfStack = topOfStack.next;
27         return topItem;
28     }
```

33

figure 16.28

A simplified
Collections-style
Stack class, based on
the ArrayList class

```
1  package weiss.util;
2
3  /**
4   * Stack class. Unlike java.util.Stack, this is not extended from
5   * Vector. This is the minimum respectable set of operations.
6   */
7  public class Stack<AnyType> implements java.io.Serializable
8  {
9      public Stack( )
10     {
11         items = new ArrayList<AnyType>( );
12     }
13
14     public AnyType push( AnyType x )
15     {
16         items.add( x );                              O(1) amortized time
17         return x;
18     }
19
20     public AnyType pop( )
21     {
22         if( isEmpty( ) )
23             throw new EmptyStackException( );
24         return items.remove( items.size( ) - 1 );    O(1) time
25     }
26
27     public AnyType peek( )
28     {
29         if( isEmpty( ) )
30             throw new EmptyStackException( );
31         return items.get( items.size( ) - 1 );
32     }
33
34     public boolean isEmpty( )
35     {
36         return size( ) == 0;
37     }
38
39     public int size( )
40     {
41         return items.size( );
42     }
43
44     public void clear( )
45     {
46         items.clear( );
47     }
48
49     private ArrayList<AnyType> items;
50  }
```

top

34

Class **Stack** in **java.util**

Method Summary	
boolean	**empty**() Tests if this stack is empty.
Object	**peek**() Looks at the object at the top of this stack without removing it from the stack.
Object	**pop**() Removes the object at the top of this stack and returns that object as the value of this function.
Object	**push**(Object item) Pushes an item onto the top of this stack.
int	**search**(Object o) Returns the 1-based position where an object is on this stack.

A more complete and consistent set of LIFO stack operations is provided by the Deque (double-ended queue) interface and its implementations, which should be used in preference to this class.
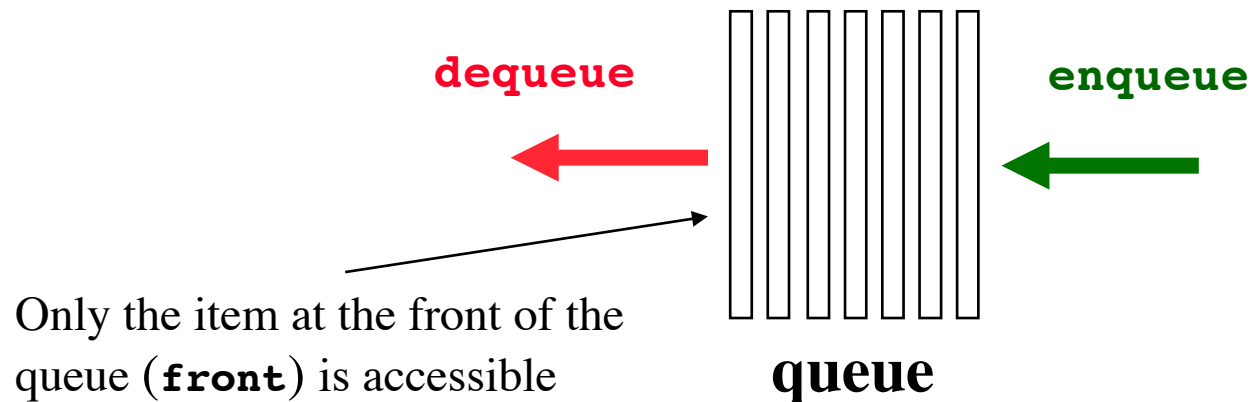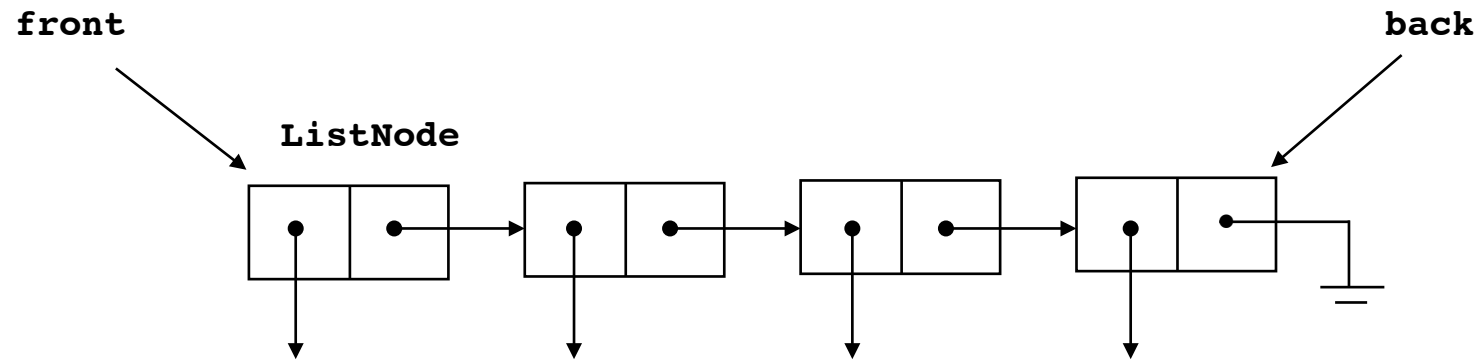
Queue

(**FIFO** = **F**irst**I**n**F**irst**O**ut)

A **queue** is a sequence of items of the same type that provides the following two operations:

enqueue(x): Insert the item **x** at the back of the queue

dequeue: Remove the item at the front of the queue

dequeue

enqueue

Only the item at the front of the queue (**front**) is accessible

queue

Queue implemented with a linked list

front   **back**

ListNode

```
class ListNode<AnyType) {
    ListNode(AnyType e) { element = e; }
    AnyType element;
    ListNode next;
}
```

figure 16.23

Skeleton for the
linked list-based
queue class

```
1  package weiss.nonstandard;
2
3  // ListQueue class
4  //
5  // CONSTRUCTION: with no initializer
6  //
7  // ********************PUBLIC OPERATIONS********************
8  // void enqueue( x )        --> Insert x
9  // AnyType getFront( )      --> Return least recently inserted item
10 // AnyType dequeue( )       --> Return and remove least recent item
11 // boolean isEmpty( )       --> Return true if empty; else false
12 // void makeEmpty( )        --> Remove all items
13 // ********************ERRORS********************
14 // getFront or dequeue on empty queue
15
16 public class ListQueue<AnyType>
17 {
18     public ListQueue( )
19       { /* Figure 16.24 */ }
20     public boolean isEmpty( )
21       { /* Figure 16.27 */ }
22     public void enqueue( AnyType x )
23       { /* Figure 16.25 */ }
24     public AnyType dequeue( )
25       { /* Figure 16.25 */ }
26     public AnyType getFront( )
27       { /* Figure 16.27 */ }
28     public void makeEmpty( )
29       { /* Figure 16.27 */ }
30
31     private ListNode<AnyType> front;
32     private ListNode<AnyType> back;
33 }
```

38

figure 16.24

Constructor for the
linked list-based
ListQueue class

```
1    /**
2     * Construct the queue.
3     */
4    public ListQueue( )
5    {
6        front = back = null;
7    }
```

```
1    /**
2     * Insert a new item into the queue.
3     * @param x the item to insert.
4     */
5    public void enqueue( AnyType x )
6    {
7        if( isEmpty( ) )     // Make a queue of one element
8            back = front = new ListNode<AnyType>( x );
9        else                        // Regular case
10           back = back.next = new ListNode<AnyType>( x );
11   }
12
13   /**
14    * Return and remove the least recently inserted item
15    * from the queue.
16    * @return the least recently inserted item in the queue.
17    * @throws UnderflowException if the queue is empty.
18    */
19   public AnyType dequeue( )
20   {
21       if( isEmpty( ) )
22           throw new UnderflowException( "ListQueue dequeue" );
23
24       AnyType returnValue = front.element;
25       front = front.next;
26       return returnValue;
27   }
```

figure 16.25

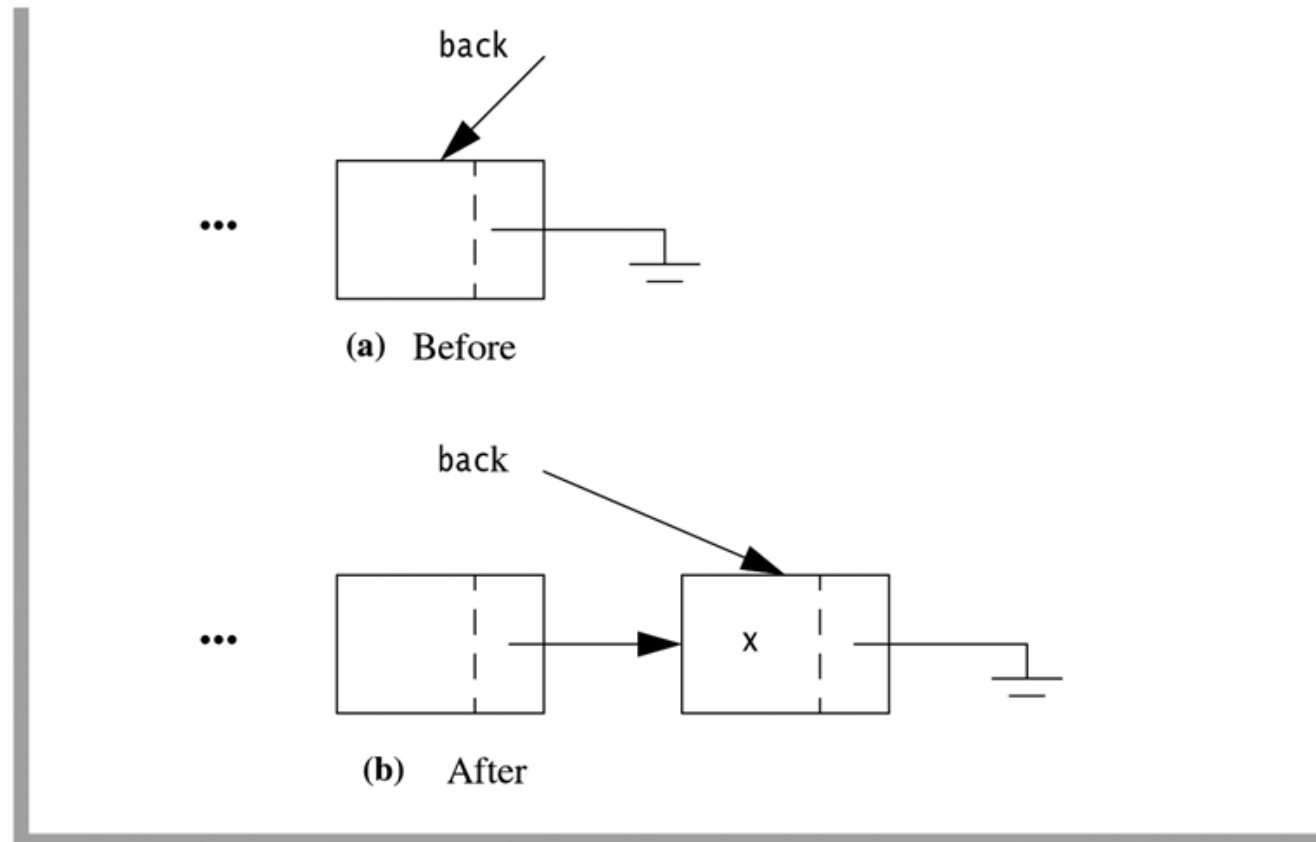The enqueue and dequeue routines for the ListQueue class

$O(1)$ time

Note the execution order (right-to-left)

$O(1)$ time

40

figure 16.26

The **enqueue** operation for the linked list-based implementation

(a) Before

(b) After

```
back = back.next = new ListNode<AnyType>( x );
```

figure 16.27

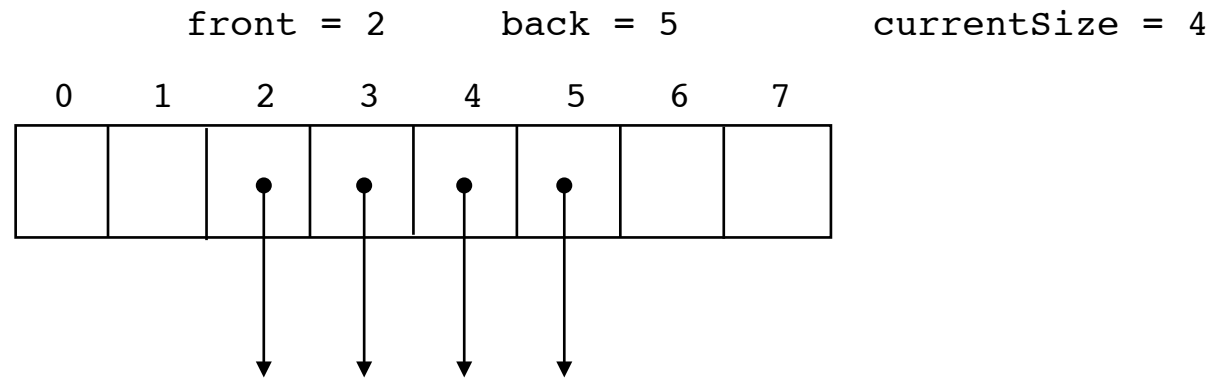Supporting routines
for the `ListQueue`
class

```
1    /**
2     * Get the least recently inserted item in the queue.
3     * Does not alter the queue.
4     * @return the least recently inserted item in the queue.
5     * @throws UnderflowException if the queue is empty.
6     */
7    public AnyType getFront( )
8    {
9        if( isEmpty( ) )
10           throw new UnderflowException( "ListQueue getFront" );
11       return front.element;
12   }
13
14   /**
15    * Make the queue logically empty.
16    */
17   public void makeEmpty( )
18   {
19       front = null;
20       back = null;
21   }
22
23   /**
24    * Test if the queue is logically empty.
25    */
26   public boolean isEmpty( )
27   {
28       return front == null;
29   }
```

Queue implemented with an array

```
front = 0                    back = 5
    0    1    2    3    4    5    6    7
```
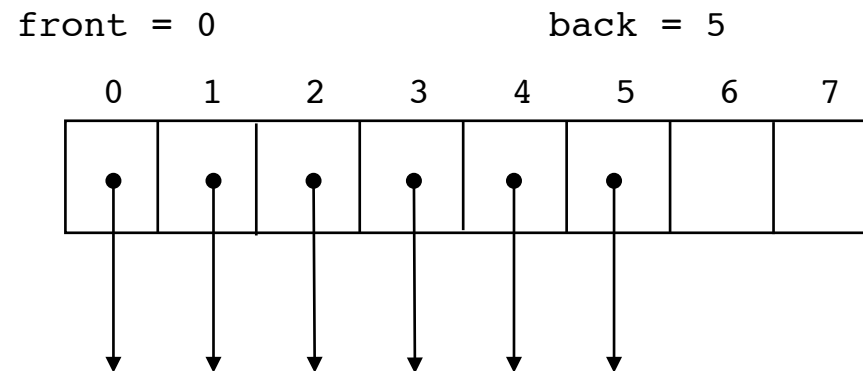
We must avoid shifting the items when an dequeue operation is executed.

Circular array

front = 2   back = 5   currentSize = 4

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
|   |   | • | • | • | • |   |   |
+---+---+---+---+---+---+---+---+
```

dequeue:

front = 3 back = 5   currentSize = 3

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
|   |   |   | • | • | • |   |   |
+---+---+---+---+---+---+---+---+
```

44

enqueue:

front = 3   back = 6   currentSize = 4

```
    0    1    2    3    4    5    6    7
  +----+----+----+----+----+----+----+----+
  |    |    |    | •  | •  | •  | •  |    |
  +----+----+----+----+----+----+----+----+
                  |    |    |    |
                  ↓    ↓    ↓    ↓
```

enqueue with *wraparound*

back = 0   front = 3   currentSize = 6

```
    0    1    2    3    4    5    6    7
  +----+----+----+----+----+----+----+----+
  | •  |    |    | •  | •  | •  | •  | •  |
  +----+----+----+----+----+----+----+----+
    |             |    |    |    |    |
    ↓             ↓    ↓    ↓    ↓    ↓
```
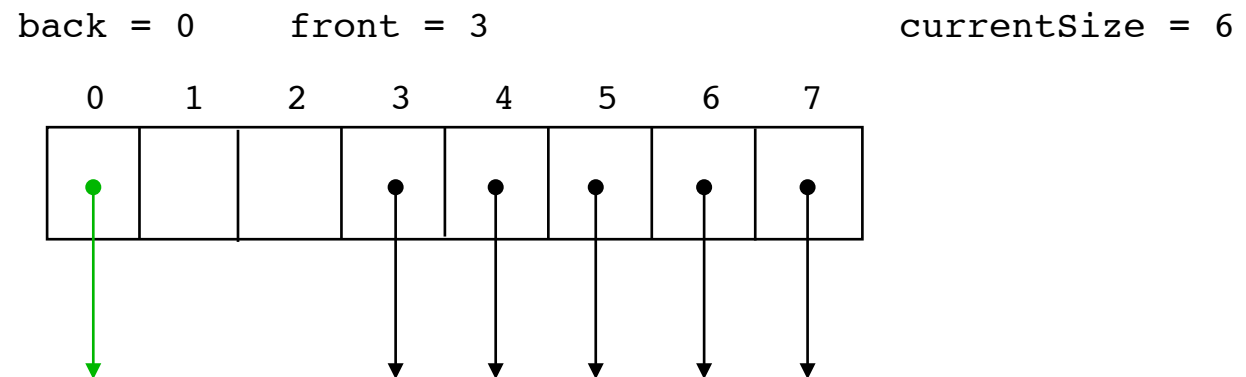
figure 16.8

Basic array
implementation of
the queue

makeEmpty( )
size = 0

enqueue(a)
size = 1

enqueue(b)
size = 2

dequeue( )
size = 1
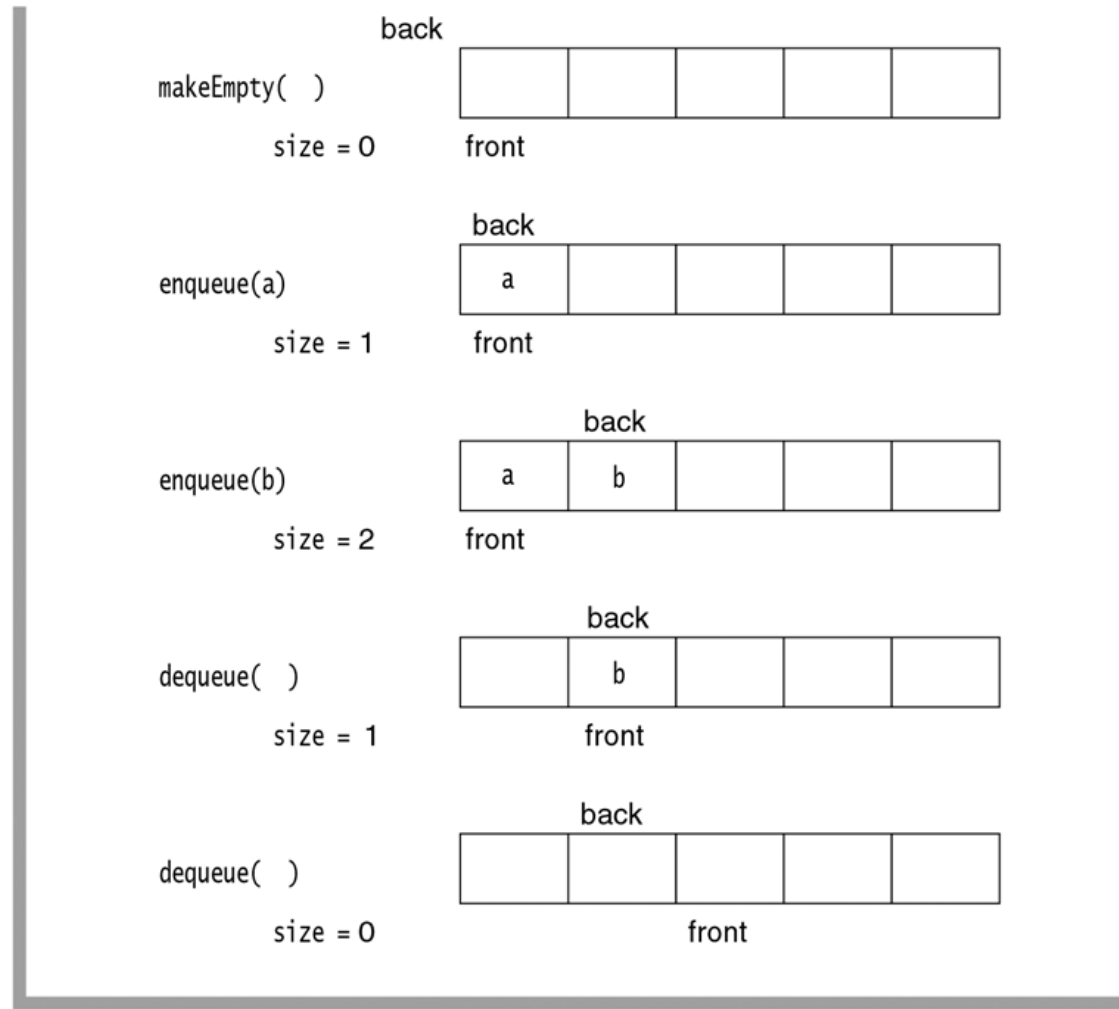
dequeue( )
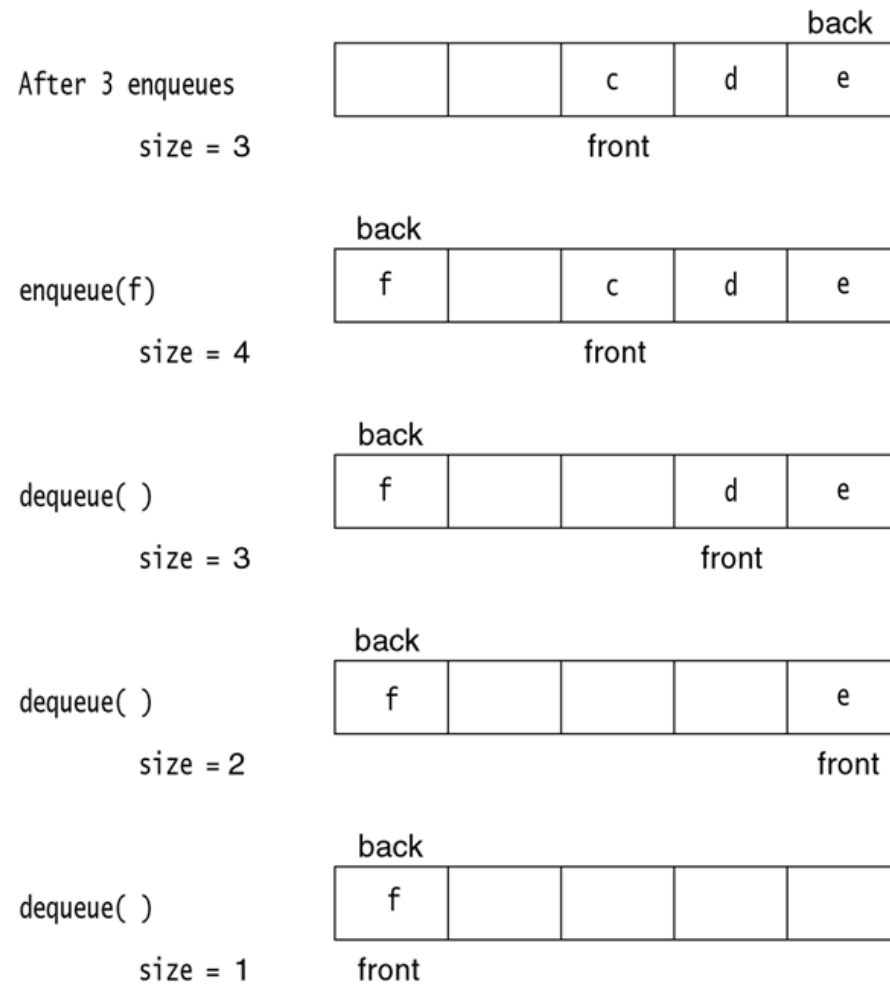size = 0

figure 16.9

Array implementation of the queue with wraparound

figure 16.10

Skeleton for the
array-based queue
class

```
 1  package weiss.nonstandard;
 2
 3  // ArrayQueue class
 4  //
 5  // CONSTRUCTION: with no initializer
 6  //
 7  // ******************PUBLIC OPERATIONS*********************
 8  // void enqueue( x )       --> Insert x
 9  // AnyType getFront( )     --> Return least recently inserted item
10  // AnyType dequeue( )      --> Return and remove least recent item
11  // boolean isEmpty( )      --> Return true if empty; else false
12  // void makeEmpty( )       --> Remove all items
13  // ******************ERRORS********************************
14  // getFront or dequeue on empty queue
15
16  public class ArrayQueue<AnyType>
17  {
18      public ArrayQueue( )
19        { /* Figure 16.12 */ }
20
21      public boolean isEmpty( )
22        { /* Figure 16.13 */ }
23      public void makeEmpty( )
24        { /* Figure 16.17 */ }
25      public AnyType dequeue( )
26        { /* Figure 16.16 */ }
27      public AnyType getFront( )
28        { /* Figure 16.16 */ }
29      public void enqueue( AnyType x )
30        { /* Figure 16.14 */ }
31
32      private int increment( int x )
33        { /* Figure 16.11 */ }
34      private void doubleQueue( )
35        { /* Figure 16.15 */ }
36
37      private AnyType [ ] theArray;
38      private int         currentSize;
39      private int         front;
40      private int         back;
41
42      private static final int DEFAULT_CAPACITY = 10;
43  }
```

```
 1      /**
 2       * Internal method to increment with wraparound.
 3       * @param x any index in theArray's range.
 4       * @return x+1, or 0 if x is at the end of theArray.
 5       */
 6      private int increment( int x )
 7      {
 8          if( ++x == theArray.length )
 9              x = 0;
10          return x;
11      }
```

figure 16.11

The wraparound
routine

```
1     /**
2      * Construct the queue.
3      */
4     public ArrayQueue( )
5     {
6         theArray = (AnyType []) new Object[ DEFAULT_CAPACITY ];
7         makeEmpty( );
8     }
```

figure 16.12

The constructor for
the ArrayQueue class

```
1    /**
2     * Test if the queue is logically empty.
3     * @return true if empty, false otherwise.
4     */
5    public boolean isEmpty( )
6    {
7        return currentSize == 0;
8    }
```

figure 16.13

The isEmpty routine
for the ArrayQueue
class

figure 16.14

The enqueue routine for the ArrayQueue class

```
1      /**
2       * Insert a new item into the queue.
3       * @param x the item to insert.
4       */
5      public void enqueue( AnyType x )
6      {
7          if( currentSize == theArray.length )
8              doubleQueue( );
9          back = increment( back );
10         theArray[ back ] = x;
11         currentSize++;
12     }
```

$O(1)$ amortized time

```
1       /**
2        * Internal method to expand theArray.
3        */
4       private void doubleQueue( )
5       {
6           AnyType [ ] newArray;
7
8           newArray = (AnyType []) new Object[ theArray.length * 2 ];
9
10              // Copy elements that are logically in the queue
11          for( int i = 0; i < currentSize; i++, front = increment( front ) )
12              newArray[ i ] = theArray[ front ];
13
14          theArray = newArray;
15          front = 0;
16          back = currentSize - 1;
17      }
```

figure 16.15

Dynamic expansion for the ArrayQueue class

```
1    /**
2     * Return and remove the least recently inserted item
3     * from the queue.
4     * @return the least recently inserted item in the queue.
5     * @throws UnderflowException if the queue is empty.
6     */
7    public AnyType dequeue( )
8    {
9        if( isEmpty( ) )
10           throw new UnderflowException( "ArrayQueue dequeue" );
11       currentSize--;
12
13       AnyType returnValue = theArray[ front ];
14       front = increment( front );
15       return returnValue;
16   }
17
18   /**
19    * Get the least recently inserted item in the queue.
20    * Does not alter the queue.
21    * @return the least recently inserted item in the queue.
22    * @throws UnderflowException if the queue is empty.
23    */
24   public AnyType getFront( )
25   {
26       if( isEmpty( ) )
27           throw new UnderflowException( "ArrayQueue getFront" );
28       return theArray[ front ];
29   }
```

figure 16.16

The dequeue and getFront routines for the ArrayQueue class

$O(1)$ time

$O(1)$ time

```
1      /**
2       * Make the queue logically empty.
3       */
4      public void makeEmpty( )
5      {
6          currentSize = 0;
7          front = 0;
8          back = -1;
9      }
```

figure 16.17

The makeEmpty routine
for the ArrayQueue
class

Interface `Queue` in `java.util`

Method Summary	
`E`	**`element()`** Retrieves, but does not remove, the head of this queue.
`boolean`	**`offer(E o)`** Inserts the specified element into this queue, if possible.
`E`	**`peek()`** Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
`E`	**`poll()`** Retrieves and removes the head of this queue, or `null` if this queue is empty.
`E`	**`remove()`** Retrieves and removes the head of this queue.

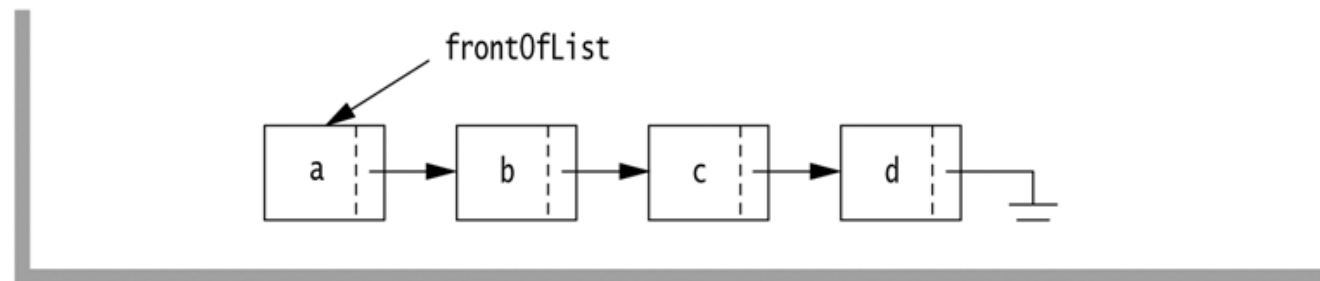Some implementing classes: `LinkedList`, `PriorityQueue`

Linked lists
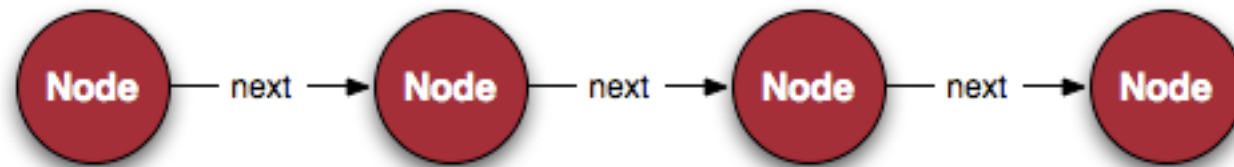
Disadvantages of storing a sequence of items as an array

(1) Insertion and removal of items take time proportional to the length of the array

(2) Arrays have a fixed length
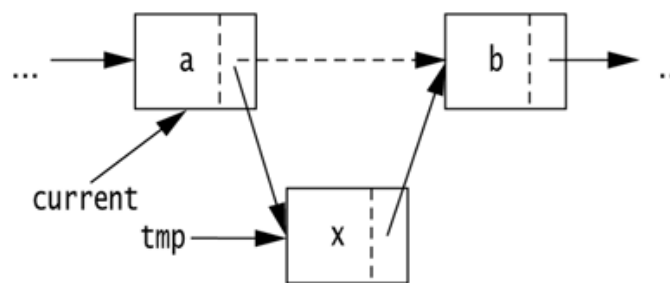
Singly linked list

figure 17.1

Basic linked list

Insertion

figure 17.2

Insertion in a linked list: Create new node (tmp), copy in x, set tmp's next link, and set current's next link.

```
current.next = new ListNode(x, current.next);
```
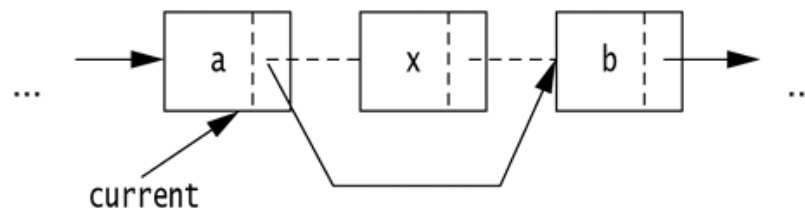
Deletion

figure 17.3

Deletion from a
linked list

```
current.next = current.next.next;
```

Using a header node for easy removal of the first element

figure 17.4

Using a header node for the linked list
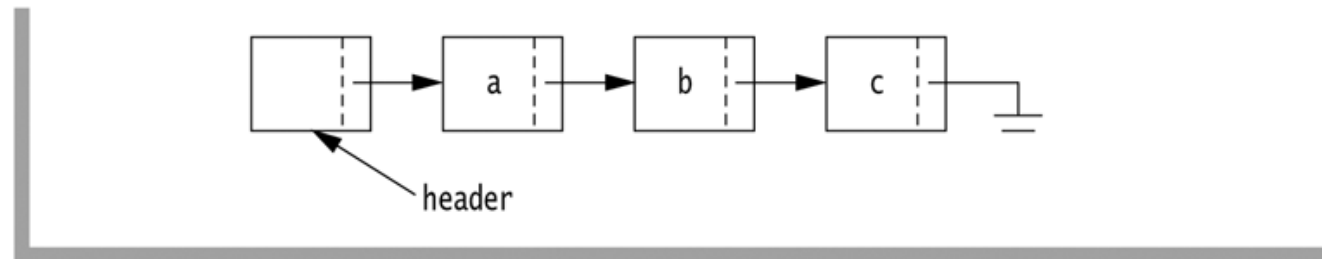
header

figure 17.5

Empty list when a header node is used

header
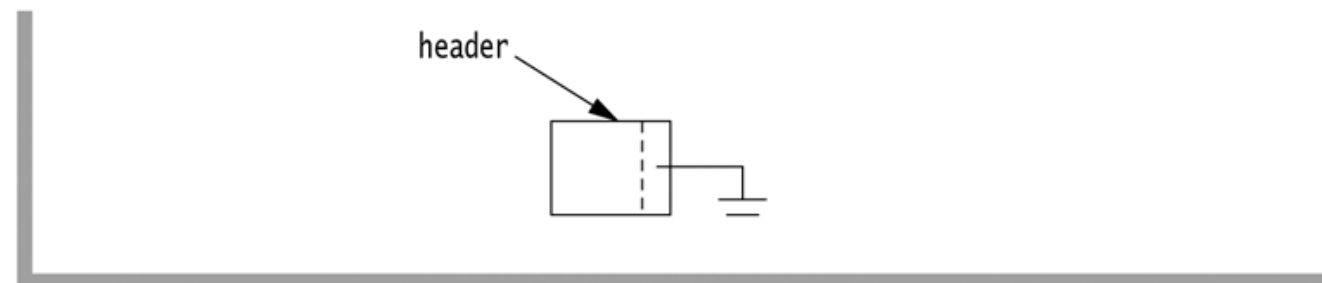
```
1      // In this routine, LinkedList and LinkedListIterator are the
2      // classes written in Section 17.2.
3      public static <AnyType> int listSize( LinkedList<AnyType> theList )
4      {
5          LinkedListIterator<AnyType> itr;
6          int size = 0;
7
8          for( itr = theList.first(); itr.isValid(); itr.advance() )
9              size++;
10
11         return size;
12     }
```

figure 17.6

A static method that returns the size of a list

non-standard

```java
 1  package weiss.nonstandard;
 2
 3  // LinkedListIterator class; maintains "current position"
 4  //
 5  // CONSTRUCTION: Package visible only, with a ListNode
 6  //
 7  // ******************PUBLIC OPERATIONS*********************
 8  // void advance( )         --> Advance
 9  // boolean isValid( )      --> True if at valid position in list
10  // AnyType retrieve        --> Return item in current position
11
12  public class LinkedListIterator<AnyType>
13  {
14      /**
15       * Construct the list iterator
16       * @param theNode any node in the linked list.
17       */
18      LinkedListIterator( ListNode<AnyType> theNode )
19        { current = theNode; }
20
21      /**
22       * Test if the current position is a valid position in the list.
23       * @return true if the current position is valid.
24       */
25      public boolean isValid( )
26        { return current != null; }
27
28      /**
29       * Return the item stored in the current position.
30       * @return the stored item or null if the current position
31       * is not in the list.
32       */
33      public AnyType retrieve( )
34        { return isValid( ) ? current.element : null; }
35
36      /**
37       * Advance the current position to the next node in the list.
38       * If the current position is null, then do nothing.
39       */
40      public void advance( )
41      {
42          if( isValid( ) )
43              current = current.next;
44      }
45
46      ListNode<AnyType> current;    // Current position
47  }
```

figure 17.7

The
LinkedListIterator
class

```
 1  package weiss.nonstandard;
 2
 3  // LinkedList class
 4  //
 5  // CONSTRUCTION: with no initializer
 6  // Access is via LinkedListIterator class
 7  //
 8  // *******************PUBLIC OPERATIONS*********************
 9  // boolean isEmpty( )       --> Return true if empty; else false
10  // void makeEmpty( )        --> Remove all items
11  // LinkedListIterator zeroth( )
12  //                          --> Return position to prior to first
13  // LinkedListIterator first( )
14  //                          --> Return first position
15  // void insert( x, p )      --> Insert x after current iterator position p
16  // void remove( x )         --> Remove x
17  // LinkedListIterator find( x )
18  //                          --> Return position that views x
19  // LinkedListIterator findPrevious( x )
20  //                          --> Return position prior to x
21  // *******************ERRORS*******************************
22  // No special errors
23
24  public class LinkedList<AnyType>
25  {
26      public LinkedList( )
27        { /* Figure 17.9 */ }
28
29      public boolean isEmpty( )
30        { /* Figure 17.9 */ }
31      public void makeEmpty( )
32        { /* Figure 17.9 */ }
33      public LinkedListIterator<AnyType> zeroth( )
34        { /* Figure 17.9 */ }
35      public LinkedListIterator<AnyType> first( )
36        { /* Figure 17.9 */ }
37      public void insert( AnyType x, LinkedListIterator<AnyType> p )
38        { /* Figure 17.14 */ }
39      public LinkedListIterator<AnyType> find( AnyType x )
40        { /* Figure 17.11 */ }
41      public LinkedListIterator<AnyType> findPrevious( AnyType x )
42        { /* Figure 17.13 */ }
43      public void remove( Object x )
44        { /* Figure 17.12 */ }
45
46      private ListNode<AnyType> header;
47  }
```

figure 17.8

The LinkedList class skeleton

```java
 1      /**
 2       * Construct the list
 3       */
 4      public LinkedList( )
 5      {
 6          header = new ListNode<AnyType>( null );
 7      }
 8
 9      /**
10       * Test if the list is logically empty.
11       * @return true if empty, false otherwise.
12       */
13      public boolean isEmpty( )
14      {
15          return header.next == null;
16      }
17
18      /**
19       * Make the list logically empty.
20       */
21      public void makeEmpty( )
22      {
23          header.next = null;
24      }
25
26      /**
27       * Return an iterator representing the header node.
28       */
29      public LinkedListIterator<AnyType> zeroth( )
30      {
31          return new LinkedListIterator<AnyType>( header );
32      }
33
34      /**
35       * Return an iterator representing the first node in the list.
36       * This operation is valid for empty lists.
37       */
38      public LinkedListIterator<AnyType> first( )
39      {
40          return new LinkedListIterator<AnyType>( header.next );
41      }
```

figure 17.9

Some LinkedList class one-liners

```java
1      // Simple print method
2      public static <AnyType> void printList( LinkedList<AnyType> theList )
3      {
4          if( theList.isEmpty( ) )
5              System.out.print( "Empty list" );
6          else
7          {
8              LinkedListIterator<AnyType> itr = theList.first( );
9              for( ; itr.isValid( ); itr.advance( ) )
10                 System.out.print( itr.retrieve( ) + " " );
11         }
12
13         System.out.println( );
14     }
```

figure 17.10

A method for printing the contents of a LinkedList

```
 1      /**
 2       * Return iterator corresponding to the first node containing x.
 3       * @param x the item to search for.
 4       * @return an iterator; iterator isPastEnd if item is not found.
 5       */
 6      public LinkedListIterator<AnyType> find( AnyType x )
 7      {
 8          ListNode<AnyType> itr = header.next;
 9
10          while( itr != null && !itr.element.equals( x ) )
11              itr = itr.next;
12
13          return new LinkedListIterator<AnyType>( itr );
14      }
```

figure 17.11

The find routine for the LinkedList class

```
1     /**
2      * Remove the first occurrence of an item.
3      * @param x the item to remove.
4      */
5     public void remove( AnyType x )
6     {
7         LinkedListIterator<AnyType> p = findPrevious( x );
8
9         if( p.current.next != null )
10            p.current.next = p.current.next.next;   // Bypass deleted node
11    }
```

figure 17.12

The remove routine for the LinkedList class

```
1      /**
2       * Return iterator prior to the first node containing an item.
3       * @param x the item to search for.
4       * @return appropriate iterator if the item is found. Otherwise, the
5       * iterator corresponding to the last element in the list is returned.
6       */
7      public LinkedListIterator<AnyType> findPrevious( AnyType x )
8      {
9          ListNode<AnyType> itr = header;
10
11         while( itr.next != null && !itr.next.element.equals( x ) )
12             itr = itr.next;
13
14         return new LinkedListIterator<AnyType>( itr );
15     }
```

figure 17.13

The findPrevious routine—similar to the find routine—for use with remove

```
 1      /**
 2       * Insert after p.
 3       * @param x the item to insert.
 4       * @param p the position prior to the newly inserted item.
 5       */
 6      public void insert( AnyType x, LinkedListIterator<AnyType> p )
 7      {
 8          if( p != null && p.current != null )
 9              p.current.next = new ListNode<AnyType>( x, p.current.next );
10      }
```

figure 17.14

The insertion routine for the LinkedList class
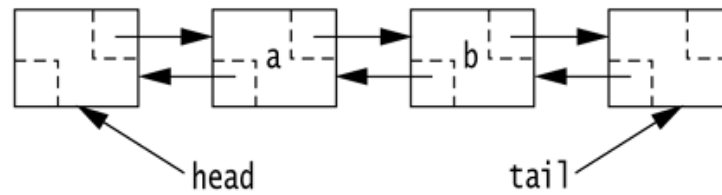
Doubly linked list

figure 17.15

A doubly linked list
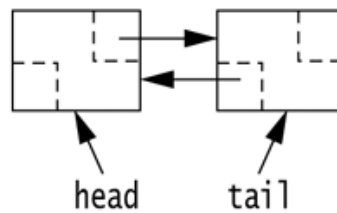
figure 17.16

An empty doubly linked list

Insertion

figure 17.17

Insertion in a doubly
linked list by getting
new node and then
changing pointers in
the order indicated
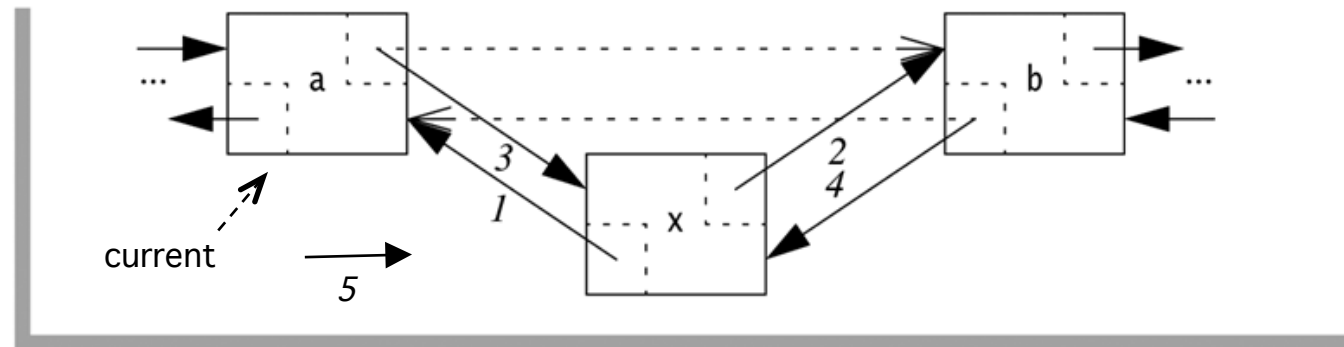
current

```
newNode = new DoublyLinkedListNode(x);
newNode.prev = current;                    // 1
newNode.next = current.next;               // 2
current.next = newNode;                     // 3
newNode.next.prev = newNode;               // 4
current = newNode;                          // 5
```

Deletion
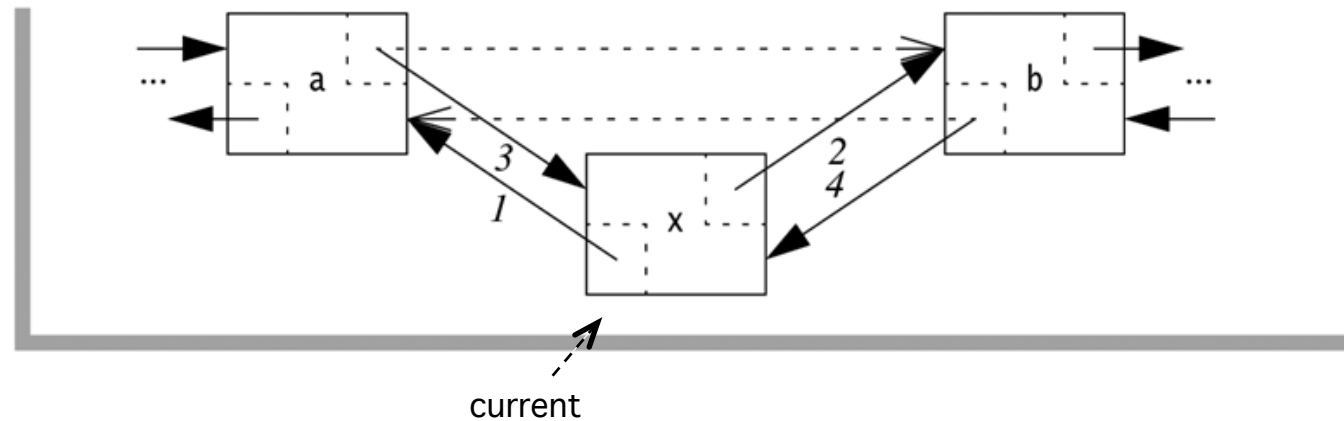
current

```
current.prev.next = current.next;    // set a's next link
current.next.prev = current.prev;    // set b's prev link
current.next = current.prev = null;
current = head;                      // So current is not stale
```

Circularly linked list

figure 17.18

A circularly and doubly linked list

```
 1 package weiss.nonstandard;
 2
 3 // SortedLinkedList class
 4 //
 5 // CONSTRUCTION: with no initializer
 6 // Access is via LinkedListIterator class
 7 //
 8 // ******************PUBLIC OPERATIONS*********************
 9 // void insert( x )          --> Insert x
10 // void insert( x, p )       --> Insert x (ignore p)
11 // All other LinkedList operations
12 // ******************ERRORS*******************************
13 // No special errors
14
15 public class SortedLinkedList<AnyType extends Comparable<? super AnyType>>
16                             extends LinkedList<AnyType>
17 {
18     /**
19      * Insert after p.
20      * @param x the item to insert.
21      * @param p this parameter is ignored.
22      */
23     public void insert( AnyType x, LinkedListIterator<AnyType> p )
24     {
25         insert( x );
26     }
27
28     /**
29      * Insert in sorted order.
30      * @param x the item to insert.
31      */
32     public void insert( AnyType x )
33     {
34         LinkedListIterator<AnyType> prev = zeroth( );
35         LinkedListIterator<AnyType> curr = first( );
36
37         while( curr.isValid( ) && x.compareTo( curr.retrieve( ) ) > 0 )
38         {
39             prev.advance( );
40             curr.advance( );
41         }
42
43         super.insert( x, prev );
44     }
45 }
```
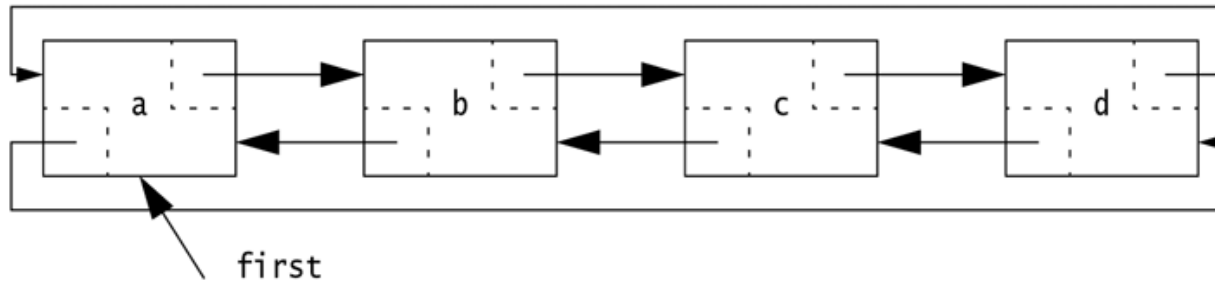
figure 17.19

The SortedLinkedList class, in which insertions are restricted to sorted order

76

```java
 1  package weiss.util;
 2  public class LinkedList<AnyType> extends AbstractCollection<AnyType>
 3                                    implements List<AnyType>, Queue<AnyType>
 4  {
 5      private static class Node<AnyType>
 6        { /* Figure 17.21 */ }
 7      private class LinkedListIterator<AnyType> implements ListIterator<AnyType>
 8        { /* Figure 17.30 */ }
 9
10      public LinkedList( )
11        { /* Figure 17.22 */ }
12      public LinkedList( Collection<AnyType> other )
13        { /* Figure 17.22 */ }
14
15      public int size( )
16        { /* Figure 17.23 */ }
17      public boolean contains( Object x )
18        { /* Figure 17.23 */ }
19      public boolean add( AnyType x )
20        { /* Figure 17.24 */ }
21      public void add( int idx, AnyType x )
22        { /* Figure 17.24 */ }
23      public void addFirst( AnyType x )
24        { /* Figure 17.24 */ }
25      public void addLast( AnyType x )
26        { /* Figure 17.24 */ }
27      public AnyType element( )
28        { /* Added in Java 5; same as getFirst */ }
29      public AnyType getFirst( )
30        { /* Figure 17.25 */ }
31      public AnyType getLast( )
32        { /* Figure 17.25 */ }
```

figure 17.20a

Class skeleton for standard LinkedList class (*continues*)

figure 17.20b

Class skeleton for
standard LinkedList
class (*continued*)

```
33      public AnyType remove( )
34        { /* Added in Java 5; same as removeFirst */ }
35      public AnyType removeFirst( )
36        { /* Figure 17.27 */ }
37      public AnyType removeLast( )
38        { /* Figure 17.27 */ }
39      public boolean remove( Object x )
40        { /* Figure 17.28 */ }
41      public AnyType get( int idx )
42        { /* Figure 17.25 */ }
43      public AnyType set( int idx, AnyType newVal )
44        { /* Figure 17.25 */ }
45      public AnyType remove( int idx )
46        { /* Figure 17.27 */ }
47      public void clear( )
48        { /* Figure 17.22 */ }
49      public Iterator<AnyType> iterator( )
50        { /* Figure 17.29 */ }
51      public ListIterator<AnyType> listIterator( int idx )
52        { /* Figure 17.29 */ }
53
54      private int theSize;
55      private Node<AnyType> beginMarker;
56      private Node<AnyType> endMarker;
57      private int modCount = 0;
58
59      private static final Node<AnyType> NOT_FOUND = null;
60      private Node<AnyType> findPos( Object x )
61        { /* Figure 17.23 */ }
62      private AnyType remove( Node<AnyType> p )
63        { /* Figure 17.27 */ }
64      private Node<AnyType> getNode( int idx )
65        { /* Figure 17.26 */}
66  }
```

```
1    /**
2     * This is the doubly linked list node.
3     */
4    private static class Node<AnyType>
5    {
6        public Node( AnyType d, Node<AnyType> p, Node<AnyType> n )
7        {
8            data = d; prev = p; next = n;
9        }
10
11       public AnyType       data;
12       public Node<AnyType> prev;
13       public Node<AnyType> next;
14   }
```

figure 17.21

Node nested class for standard LinkedList class

```java
 1      /**
 2       * Construct an empty LinkedList.
 3       */
 4      public LinkedList( )
 5      {
 6          clear( );
 7      }
 8
 9      /**
10       * Construct a LinkedList with same items as another Collection.
11       */
12      public LinkedList( Collection<AnyType> other )
13      {
14          clear( );
15          for( AnyType val : other )
16              add( val );
17      }
18
19      /**
20       * Change the size of this collection to zero.
21       */
22      public void clear( )
23      {
24          beginMarker = new Node<AnyType>( null, null, null );
25          endMarker = new Node<AnyType>( null, beginMarker, null );
26          beginMarker.next = endMarker;
27
28          theSize = 0;
29          modCount++;
30      }
```

figure 17.22

Constructors and clear method for standard LinkedList class

```java
1     /**
2      * Returns the number of items in this collection.
3      * @return the number of items in this collection.
4      */
5     public int size( )
6     {
7         return theSize;
8     }
9
10    /**
11     * Tests if some item is in this collection.
12     * @param x any object.
13     * @return true if this collection contains an item equal to x.
14     */
15    public boolean contains( Object x )
16    {
17        return findPos( x ) != NOT_FOUND;
18    }
19
20    /**
21     * Returns the position of first item matching x
22     * in this collection, or NOT_FOUND if not found.
23     * @param x any object.
24     * @return the position of first item matching x
25     * in this collection, or NOT_FOUND if not found.
26     */
27    private Node<AnyType> findPos( Object x )
28    {
29        for( Node<AnyType> p = beginMarker.next; p != endMarker; p = p.next )
30            if( x == null )
31            {
32                if( p.data == null )
33                    return p;
34            }
35            else if( x.equals( p.data ) )
36                return p;
37
38        return NOT_FOUND;
39    }
```

figure 17.23

size and contains for standard LinkedList class

figure 17.24

add methods for
standard LinkedList
class

```
1    /**
2     * Adds an item to this collection, at the end.
3     * @param x any object.
4     * @return true.
5     */
6    public boolean add( AnyType x )
7    {
8        addLast( x );
9        return true;
10   }
11
12   /**
13    * Adds an item to this collection, at the front.
14    * Other items are slid one position higher.
15    * @param x any object.
16    */
17   public void addFirst( AnyType x )
18   {
19       add( 0, x );
20   }
21
22   /**
23    * Adds an item to this collection, at the end.
24    * @param x any object.
25    */
26   public void addLast( AnyType x )
27   {
28       add( size( ), x );
29   }
30
31   /**
32    * Adds an item to this collection, at a specified position.
33    * Items at or after that position are slid one position higher.
34    * @param x any object.
35    * @param idx position to add at.
36    * @throws IndexOutOfBoundsException if idx is not
37    *         between 0 and size(), inclusive.
38    */
39   public void add( int idx, AnyType x )
40   {
41       Node<AnyType> p = getNode( idx );
42       Node<AnyType> newNode = new Node<AnyType>( x, p.prev, p );
43       newNode.prev.next = newNode;
44       p.prev = newNode;
45       theSize++;
46       modCount++;
47   }
```

82

figure 17.25

get and set methods
for standard
LinkedList class

```
1    /**
2     * Returns the first item in the list.
3     * @throws NoSuchElementException if the list is empty.
4     */
5    public AnyType getFirst( )
6    {
7        if( isEmpty( ) )
8            throw new NoSuchElementException( );
9        return getNode( 0 ).data;
10   }
11
12   /**
13    * Returns the last item in the list.
14    * @throws NoSuchElementException if the list is empty.
15    */
16   public AnyType getLast( )
17   {
18       if( isEmpty( ) )
19           throw new NoSuchElementException( );
20       return getNode( size( ) - 1 ).data;
21   }
22
23   /**
24    * Returns the item at position idx.
25    * @param idx the index to search in.
26    * @throws IndexOutOfBoundsException if index is out of range.
27    */
28   public AnyType get( int idx )
29   {
30       return getNode( idx ).data;
31   }
32
33   /**
34    * Changes the item at position idx.
35    * @param idx the index to change.
36    * @param newVal the new value.
37    * @return the old value.
38    * @throws IndexOutOfBoundsException if index is out of range.
39    */
40   public AnyType set( int idx, AnyType newVal )
41   {
42       Node<AnyType> p = getNode( idx );
43       AnyType oldVal = p.data;
44
45       p.data = newVal;
46       return oldVal;
47   }
```

```
1      /**
2       * Gets the Node at position idx, which must range from 0 to size( ).
3       * @param idx index to search at.
4       * @return internal node corresponding to idx.
5       * @throws IndexOutOfBoundsException if idx is not
6       *            between 0 and size(), inclusive.
7       */
8      private Node<AnyType> getNode( int idx )
9      {
10         Node<AnyType> p;
11
12         if( idx < 0 || idx > size( ) )
13             throw new IndexOutOfBoundsException( );
14
15         if( idx < size( ) / 2 )
16         {
17             p = beginMarker.next;
18             for( int i = 0; i < idx; i++ )
19                 p = p.next;
20         }
21         else
22         {
23             p = endMarker;
24             for( int i = size( ); i > idx; i-- )
25                 p = p.prev;
26         }
27
28         return p;
29     }
```

figure 17.26

Private getNode for standard LinkedList class

figure 17.27

remove methods for standard `LinkedList` class

```
1    /**
2     * Removes the first item in the list.
3     * @return the item was removed from the collection.
4     * @throws NoSuchElementException if the list is empty.
5     */
6    public AnyType removeFirst( )
7    {
8        if( isEmpty( ) )
9            throw new NoSuchElementException( );
10       return remove( getNode( 0 ) );
11   }
12
13   /**
14    * Removes the last item in the list.
15    * @return the item was removed from the collection.
16    * @throws NoSuchElementException if the list is empty.
17    */
18   public AnyType removeLast( )
19   {
20       if( isEmpty( ) )
21           throw new NoSuchElementException( );
22       return remove( getNode( size( ) - 1 ) );
23   }
24
25   /**
26    * Removes an item from this collection.
27    * @param idx the index of the object.
28    * @return the item that was removed from the collection.
29    */
30   public AnyType remove( int idx )
31   {
32       return remove( getNode( idx ) );
33   }
34
35   /**
36    * Removes the object contained in Node p.
37    * @param p the Node containing the object.
38    * @return the item that was removed from the collection.
39    */
40   private AnyType remove( Node<AnyType> p )
41   {
42       p.next.prev = p.prev;
43       p.prev.next = p.next;
44       theSize--;
45       modCount++;
46
47       return p.data;
48   }
```

85

```java
1    /**
2     * Removes an item from this collection.
3     * @param x any object.
4     * @return true if this item was removed from the collection.
5     */
6    public boolean remove( Object x )
7    {
8        Node<AnyType> pos = findPos( x );
9
10       if( pos == NOT_FOUND )
11           return false;
12       else
13       {
14           remove( pos );
15           return true;
16       }
17   }
```

figure 17.28

Additional remove method for standard LinkedList class

```java
1      /**
2       * Obtains an Iterator object used to traverse the collection.
3       * @return an iterator positioned prior to the first element.
4       */
5      public Iterator<AnyType> iterator( )
6      {
7          return new LinkedListIterator( 0 );
8      }
9
10     /**
11      * Obtains a ListIterator object used to traverse the
12      * collection bidirectionally.
13      * @return an iterator positioned prior to the requested element.
14      * @param idx the index to start the iterator. Use size() to do
15      * complete reverse traversal. Use 0 to do complete forward traversal.
16      * @throws IndexOutOfBoundsException if idx is not
17      *          between 0 and size(), inclusive.
18      */
19     public ListIterator<AnyType> listIterator( int idx )
20     {
21         return new LinkedListIterator( idx );
22     }
```

figure 17.29

Iterator factory methods for standard LinkedList class

```
1     /**
2      * This is the implementation of the LinkedListIterator.
3      * It maintains a notion of a current position and of
4      * course the implicit reference to the LinkedList.
5      */
6     private class LinkedListIterator implements ListIterator<AnyType>
7     {
8         private Node<AnyType> current;
9         private Node<AnyType> lastVisited = null;
10        private boolean lastMoveWasPrev = false;
11        private int expectedModCount = modCount;
12
13        public LinkedListIterator( int idx )
14        {
15            current = getNode( idx );
16        }
17
18        public boolean hasNext( )
19        {
20            if( expectedModCount != modCount )
21                throw new ConcurrentModificationException( );
22            return current != endMarker;
23        }
24
25        public AnyType next( )
26        {
27            if( !hasNext( ) )
28                throw new NoSuchElementException( );
29
30            AnyType nextItem = current.data;
31            lastVisited = current;
32            current = current.next;
33            lastMoveWasPrev = false;
34            return nextItem;
35        }
```

figure 17.30a

Iterator inner class implementation for standard LinkedList class (*continues*)

```java
36          public void remove( )
37          {
38              if( expectedModCount != modCount )
39                  throw new ConcurrentModificationException( );
40              if( lastVisited == null )
41                  throw new IllegalStateException( );
42
43              LinkedList.this.remove( lastVisited );
44              lastVisited = null;
45              if( lastMoveWasPrev )
46                  current = current.next;
47              expectedModCount++;
48          }
49
50          public boolean hasPrevious( )
51          {
52              if( expectedModCount != modCount )
53                  throw new ConcurrentModificationException( );
54              return current != beginMarker.next;
55          }
56
57          public AnyType previous( )
58          {
59              if( expectedModCount != modCount )
60                  throw new ConcurrentModificationException( );
61              if( !hasPrevious( ) )
62                  throw new NoSuchElementException( );
63
64              current = current.prev;
65              lastVisited = current;
66              lastMoveWasPrev = true;
67              return current.data;
68          }
69      }
```

figure 17.30b

Iterator inner class implementation for standard LinkedList class (*continued*)