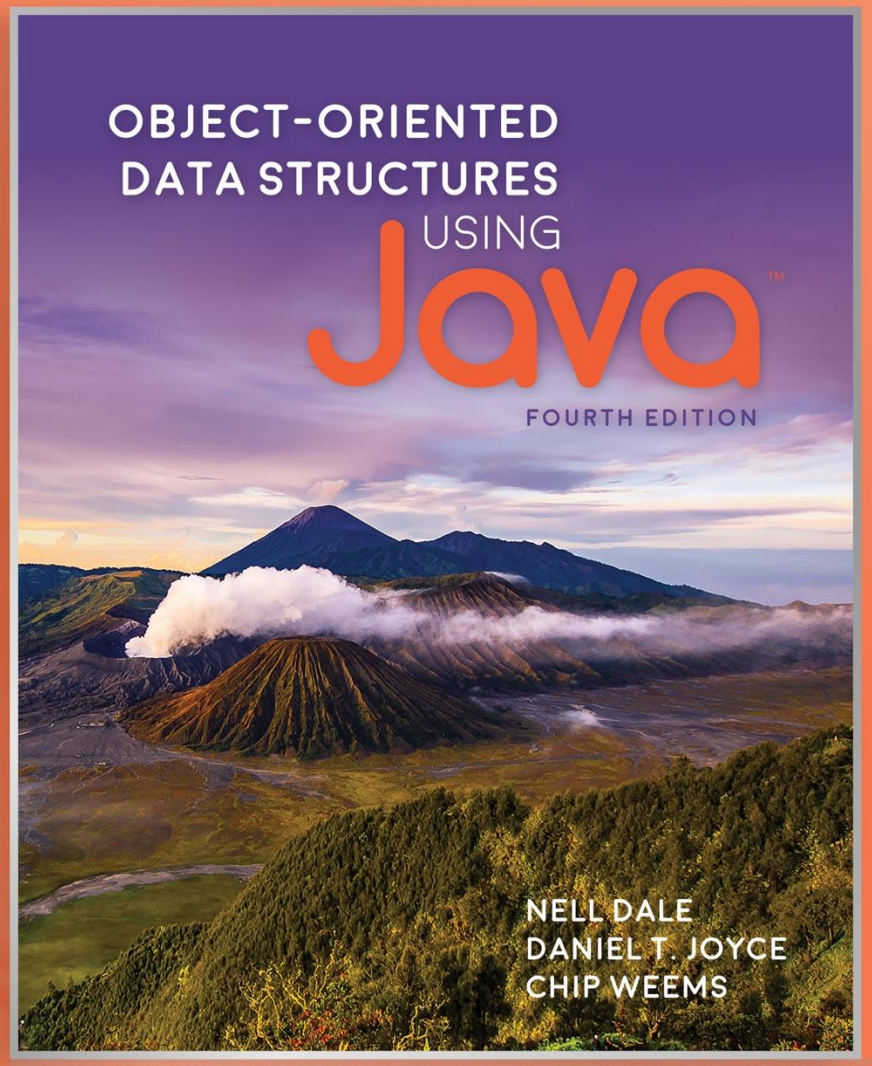
Chapter 6

The List ADT



Chapter 6: The List ADT

6.1 – The List Interface
6.2 – List Implementations
6.3 – Applications: Card Deck and Games
6.4 – Sorted Array-Based List Implementation
6.5 – List Variations
6.6 – Application: Large Integers

6.1 – The List Interface

- A list is a collection of elements, with a linear relationship existing among its elements.
- Each element on the list has a position on the list, its index.
- In addition to our lists supporting the standard collection operations add, get, contains, remove, isFull, isEmpty, and size, they support index-related operations and iteration.

Indexes

- The elements of a list are indexed sequentially, from zero to one less than the size of the list
- We define methods for adding, retrieving, changing, and removing an element at an indicated index, as well as a method for determining the index of an element.
- Each method that accepts an index as an argument throws an exception (`IndexOutOfBoundsException`) if the index is invalid

For example

```
void add(int index, T element);
// Throws IndexOutOfBoundsException if passed an index argument
// such that index < 0 or index > size().
// Otherwise, adds element to this list at position index; all current
// elements at that position or higher have 1 added to their index.
// Optional. Throws UnsupportedOperationException if not supported.

T set(int index, T newElement);
// Throws IndexOutOfBoundsException if passed an index argument
// such that index < 0 or index >= size().
// Otherwise, replaces element on this list at position index with
// newElement and returns the replaced element.
// Optional. Throws UnsupportedOperationException if not supported.
```

Optional Operations

- The `add` and `set` operations are optional.
- These operations allow the client to insert an element into a list at a specified index and for some list implementations, notably a sorted list implementation, this could invalidate the internal representation of the list.
- Our implementations will throw the Java library supplied `UnsupportedOperationException` in cases where an implementation does not support an operation.

Iteration

- Our lists implement the library's `Iterable` interface.
- `Iterable` requires a single method, `iterator`, that creates and returns an `Iterator` object.
- Methods that create and return objects are sometimes called Factory methods.
- `Iterator` objects provide three operations: `hasNext`, `next`, and `remove`.

Example use of an `Iterator`

Suppose `strings` is a List ADT object that contains the four strings “alpha,” “gamma,” “beta,” and “delta.” The following code would delete “gamma” from the list and display the other three strings.

```
Iterator<String> iter = strings.iterator();
String hold;
while (iter.hasNext())
{
  hold = iter.next();
  if (hold.equals("gamma"))
    iter.remove();
  else
    System.out.println(hold);
}
```

ListInterface (comments removed)

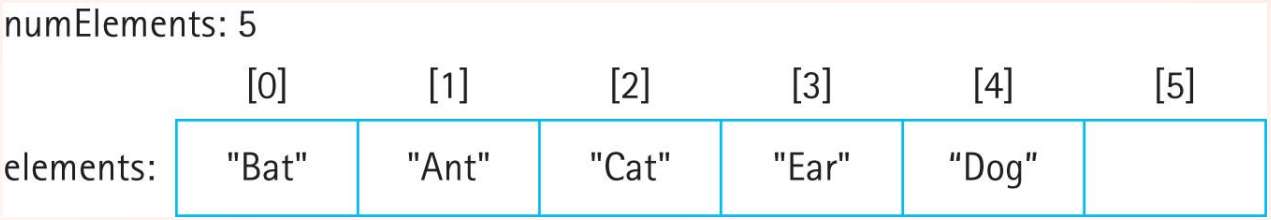
```
package ch06.lists;

import java.util.*;
import ch05.collections.CollectionInterface;

public interface ListInterface<T> extends CollectionInterface<T>,
                                          Iterable<T>
{
  void add(int index, T element);

  T set(int index, T newElement);

  T get(int index);

  int indexOf(T target);

  T remove(int index);
}
```

6.2 List Implementations

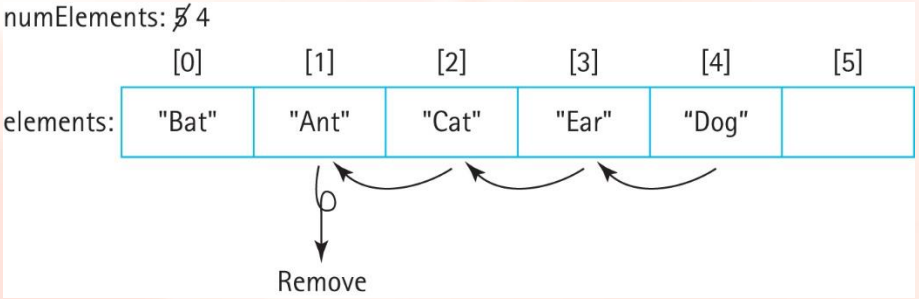
- In this section we develop an array-based and a link-based implementation of the List ADT.
- Because a list is a collection the mplementations share some design and code with their Collection ADT counterparts.
- Here we emphasize the new functionality—the indexing and the iteration

Array-Based Implementation

- Same approach for our array-based list:



- … except must maintain index "order" of elements during operations:



Index Related Operations

- The methods each follow the same pattern:
  - check the index argument
  - if it is outside the allowable range for that operation throw an exception
  - otherwise carry out the operation.
- Because of the close logical relationship between the internal representation, an array, and the ADT, an indexed list, the implementation of these operations is very straightforward.

For example, the `set` method

```
public T set(int index, T newElement)
// Throws IndexOutOfBoundsException if passed an index argument
// such that index < 0 or index >= size().
// Otherwise, replaces element on this list at position index with
// newElement and returns the replaced element.
{
  if ((index < 0) || (index >= size()))
    throw new IndexOutOfBoundsException("Illegal index of " + index +
                                        " passed to ABList set method.\n");

  T hold = elements[index];
  elements[index] = newElement;
  return hold;
}
```

Iteration

- We use an anonymous inner class approach
- Anonymous class has no name .. it is just instantiated where needed
- The behavior of an iterator is unspecified if the underlying representation is modified while the iteration is in progress in any way other than by calling the iterator's `remove` method

```
public Iterator<T> iterator()
{
   return new Iterator<T>()
   {
      private int previousPos = -1;

      public boolean hasNext()
      {
         return (previousPos < (size() - 1)) ;
      }

      public T next()
      {
         if (!hasNext())
            throw new IndexOutOfBoundsException("Illegal invocation of next " +
                                                " in LBList iterator.\n");
         previousPos++;
         return elements[previousPos];
      }

      public void remove()
      {
         for (int i = previousPos; i <= numElements - 2; i++)
            elements [i] = elements[i+1];
         elements [numElements - 1] = null;
         numElements--;
         previousPos--;
      }
   };
}
```

Link-Based Implementation

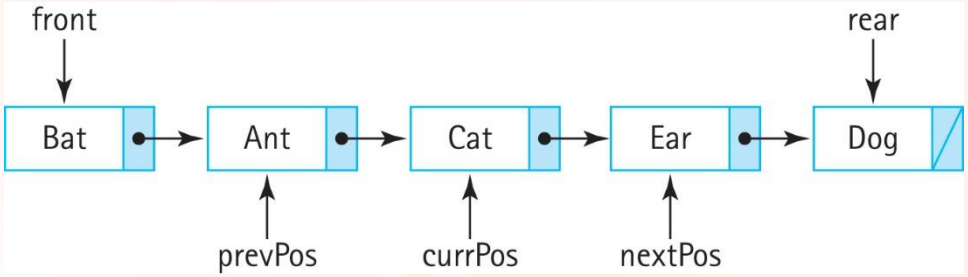
- Some of the link-based collection implementation design and code can be reused for the link-based list.
- To support the `add` method, which adds elements to the end of the list, we maintain a new reference `rear` to the end of the list.
- To support the `indexOf` method we include a new `targetIndex` variable, which the `find` method sets, in addition to setting `found`, `location`, and `previous`.

Example Index Related Operation

```
public T set(int index, T newElement
// Throws IndexOutOfBoundsException if passed an index argument
// such that index < 0 or index >= size().
// Otherwise, replaces element on this list at position index with
// newElement and returns the replaced element.
{
   if ((index < 0) || (index >= size()))
      throw new IndexOutOfBoundsException("Illegal index of " + index +
                                          " passed to LBList set method.\n");
   LLNode<T> node = front;
   for (int i = 0; i < index; i++)
      node = node.getLink();
   T hold = node.getInfo();
   node.setInfo(newElement);
   return hold;
}
```

Iteration

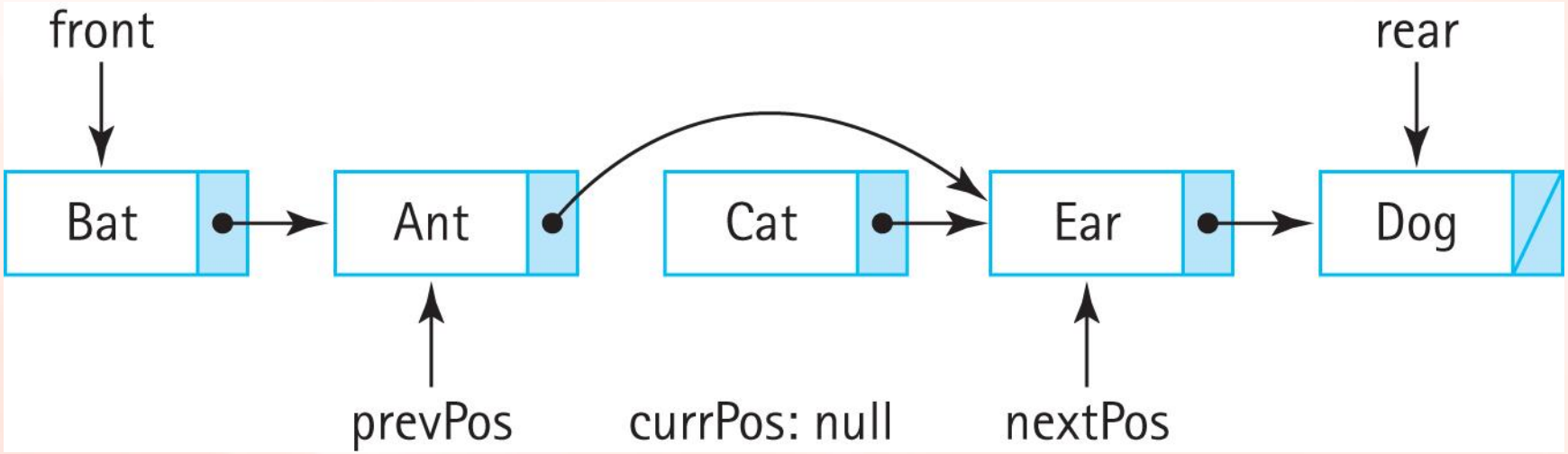
- Again use an anonymous inner class within the `iterator` method.
- The instantiated `Iterator` object keeps track of three instance variables to provide the iteration and to support the required remove operation:



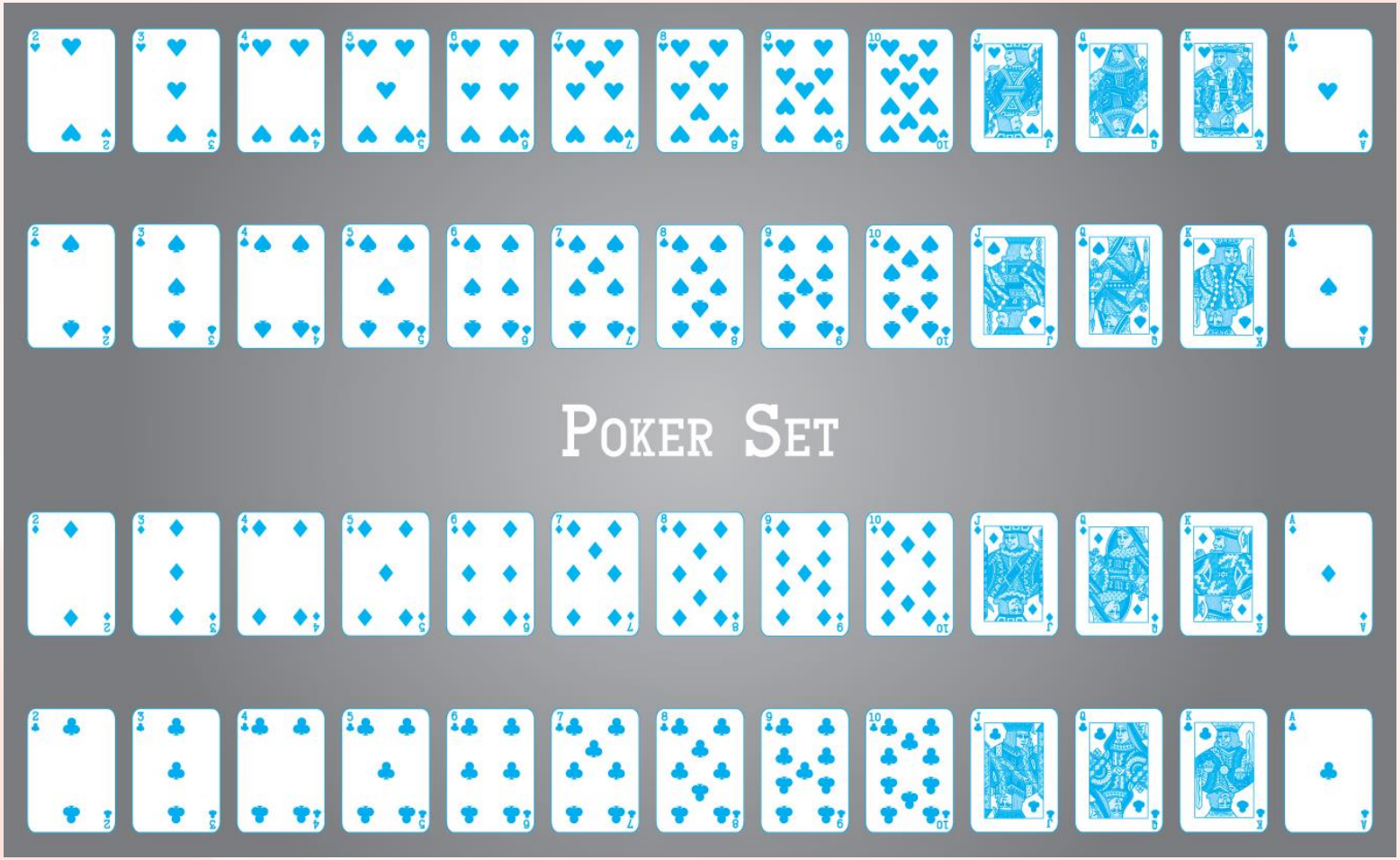
- The `next` method returns the element referenced by `nextPos` and updates the three references

Iteration

- If `remove` invoked in the middle of an iteration it removes the element that was just returned, the element referenced by `currPos`:



6.3 Applications: Card Deck and Games



The `Card` class

- Found in the `support.cards` package
- A card object has three attributes:
  - rank: the rank of the card e.g. Five or King
  - suit: the suit of the card e.g. Heart or Spade
  - image: an image icon associated with the card
- `rank` and `suit` are both represented by public `enum` classes provided by the `Card` class
- The image files used for the image icons are also located in the `support.cards` package
- Attribute getter methods are provided plus an `equals`, a `compareTo` and a `toString`

The `CardDeck` class

- Uses an `ABList` of `Card` objects named `deck` as its internal representation
- Another instance variable, `deal`, which holds an `Iterator<Card>` object, is used to deal cards
- `deal` is set to `deck.iterator()`
- Exports methods for shuffling the deck and iterating through the deck

Applications

- `CardHandCLI` – command line interface program which deals a 5 card hand from a card deck, allowing the user to arrange the cards
- `CardHandGUI` – graphical user interface program which deals a 5 card hand from a card deck, allowing the user to arrange the cards
- `HigherLower` – Predict whether the next card will be higher or lower
- `Pairs` – Analyzes the probability of being dealt a pair in a 5 card hand

6.4 Sorted Array-Based List Implementation

- Class `SortedABList` implements `ListInterface` and is found in the `ch06.lists` package
- Much of the design and code of the `SortedArrayCollection` from the `ch05.collections` package can be reused.
- We state as a general precondition of the class that the index-based `add` and `set` operations are not supported.

Code for the two unsupported methods

```
public void add(int index, T element)
// Throws UnsupportedOperationException.
{
  throw new UnsupportedOperationException("Unsupported index-based add …
}

public T set(int index, T newElement)
// Throws UnsupportedOperationException.
{
  throw new UnsupportedOperationException("Unsupported index-based set …
}
```

Comparator Interface

- We want to allow clients of our `SortedABList` to be able to specify for themselves how the elements should be sorted
- The Java `Comparator` interface defines two abstract methods:

```
public abstract int compare(T o1, T o2);
// Returns a negative integer, zero, or a positive integer to indicate that
// o1 is less than, equal to, or greater than o2

public abstract boolean equals(Object obj);
// Returns true if this Comparator equals obj; otherwise, false
```

- Using an approach based on the `Comparator` class allows for multiple sorting orders

Comparator Interface

- Using an approach based on the `Comparator` class allows for multiple sorting orders
- For example our FamousPerson class typically bases comparison on last name, first name but it could also define other approaches:

```
public static Comparator<FamousPerson> yearOfBirthComparator()
{
  return new Comparator<FamousPerson>()
  {
    public int compare(FamousPerson element1, FamousPerson element2)
    {
      return (element1.yearOfBirth - element2.yearOfBirth);
    }
  };
}
```

SortedABList Constructors

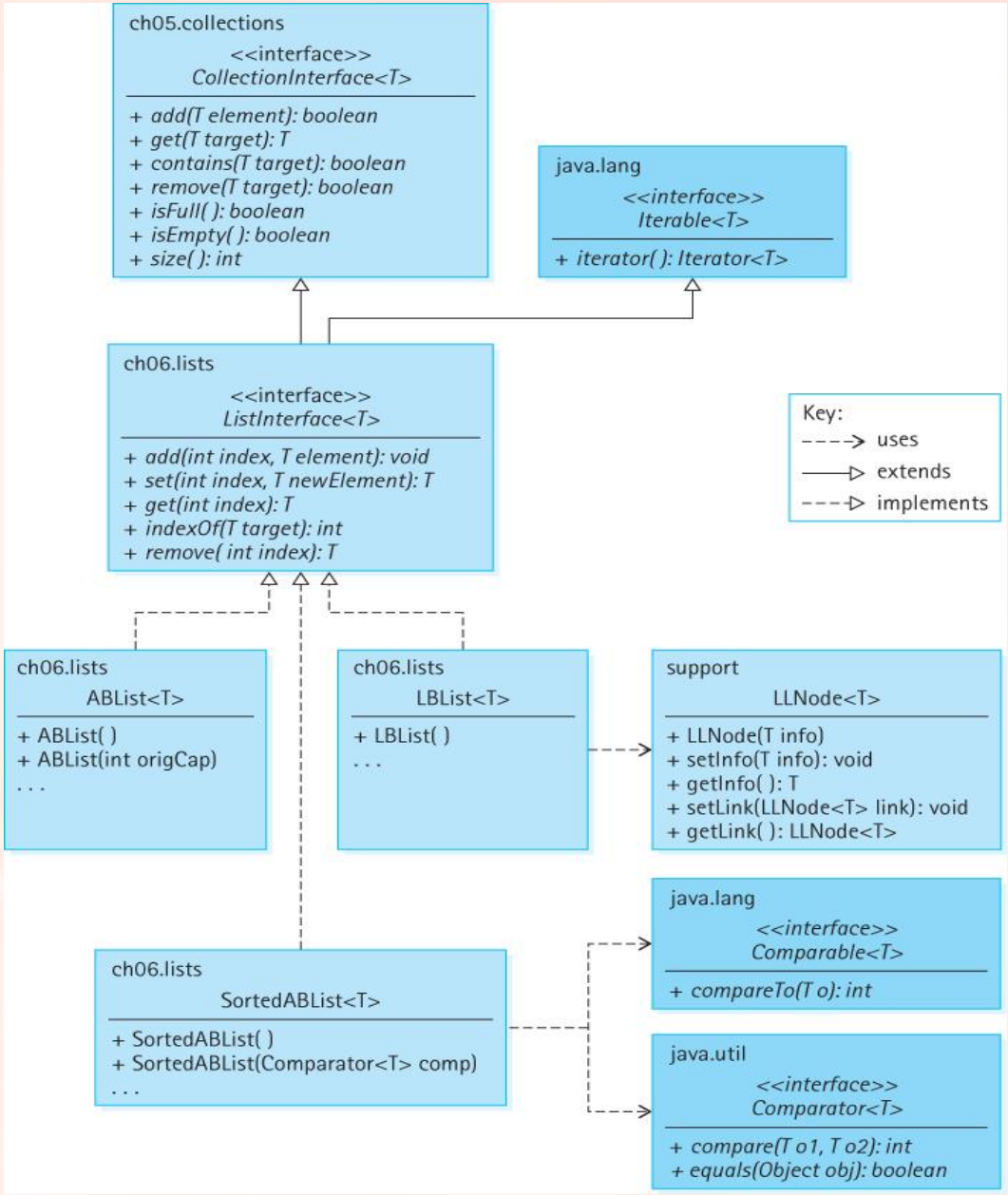
- There are two constructors
- One uses the “natural order” of the elements
- The other uses an order provided by the client who passes an appropriate `Comparator` object as an argument to the method
- A private variable `comp` of class `Comparator<T>` is used to make comparisons internally and is set by the invoked constructor

SortedABList Constructors

```
protected Comparator<T> comp;

public SortedABList()
// Precondition: T implements Comparable
{
  list = (T[]) new Object[DEFCAP];
  comp = new Comparator<T>()
  {
    public int compare(T element1, T element2)
    {
      return ((Comparable)element1).compareTo(element2);
    }
  };
}

public SortedABList(Comparator<T> comp)
{
  list = (T[]) new Object[DEFCAP];
  this.comp = comp;
}
```

See the `CSPeople` application in the `ch06.apps` package for an example demonstrating the use of these two constructors.

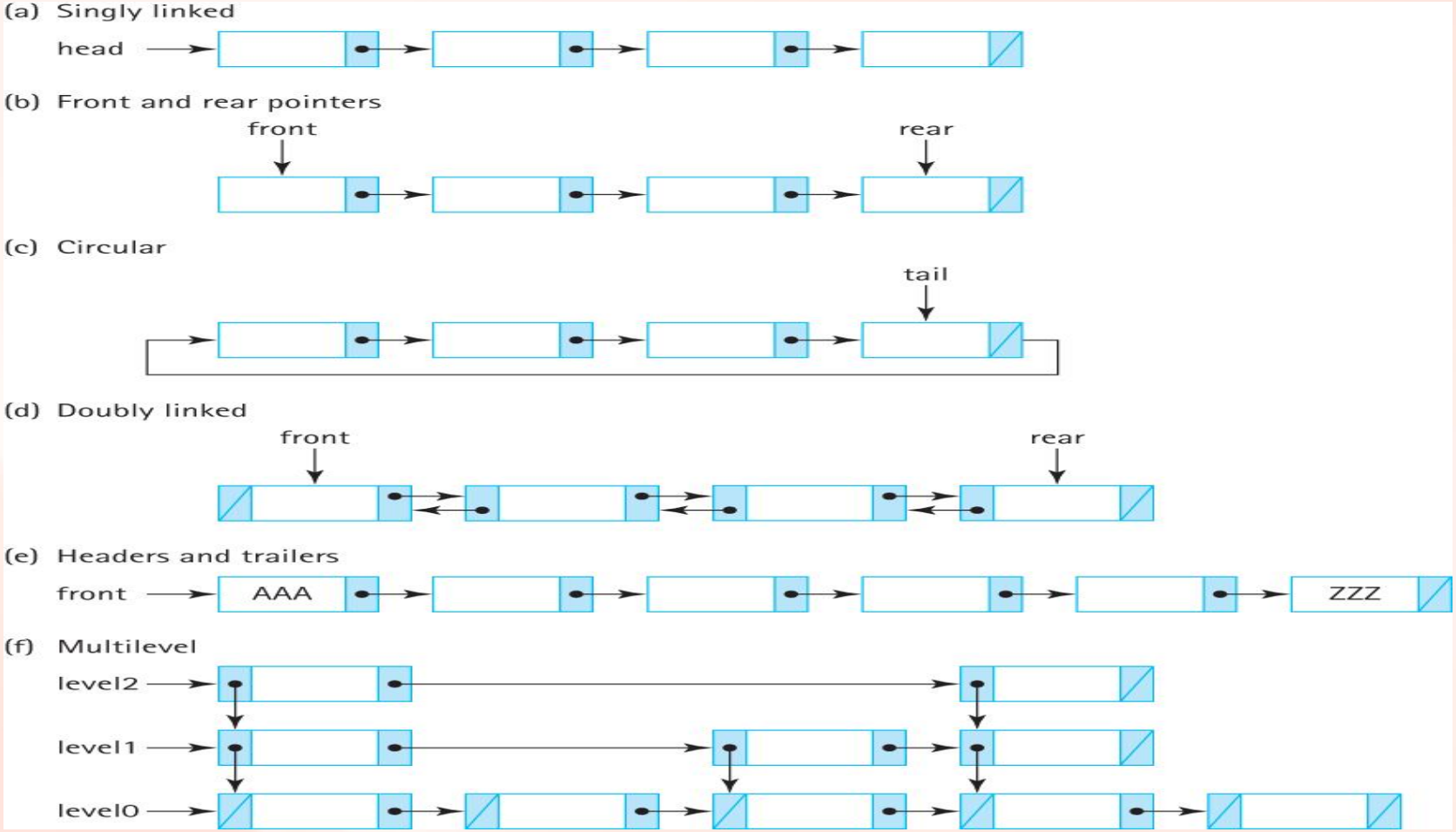
Our List ADT Architecture



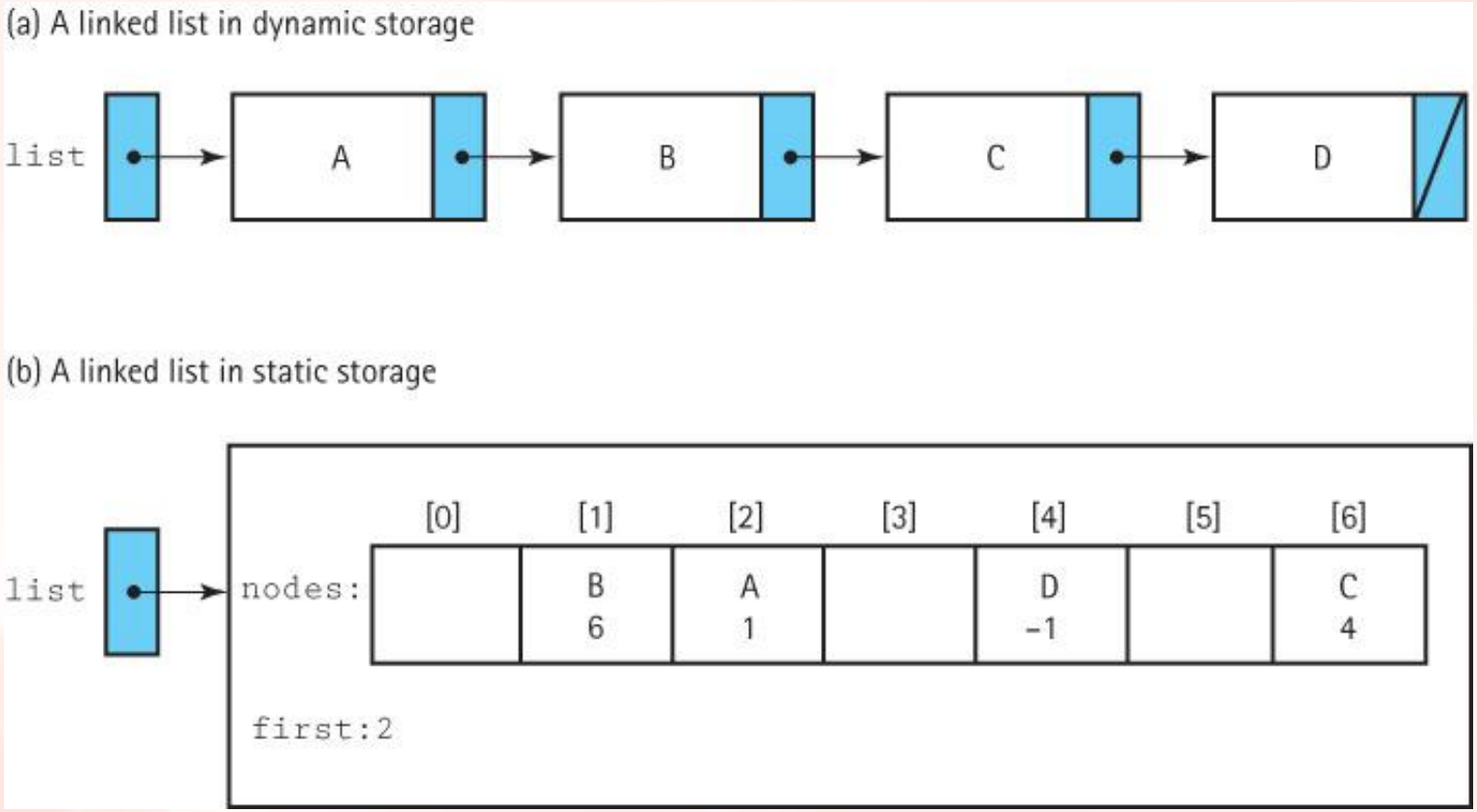
6.5 List Variations

- Java Library
  - The library provides a `List` interface that inherits from both the `Collection` and `Iterable` interfaces of the library.
  - The library's list interface is significantly more complex than ours, defining 28 abstract methods.
  - It is implemented by the following classes: `AbstractList`, `AbstractSequentialList`, `ArrayList`, `AttributeList`, `CopyOnWriteArrayList`, `LinkedList`, `RoleList`, `RoleUnresolvedList`, `Stack`, and `Vector`.

Linked List Variations



A Linked List as an Array of Nodes



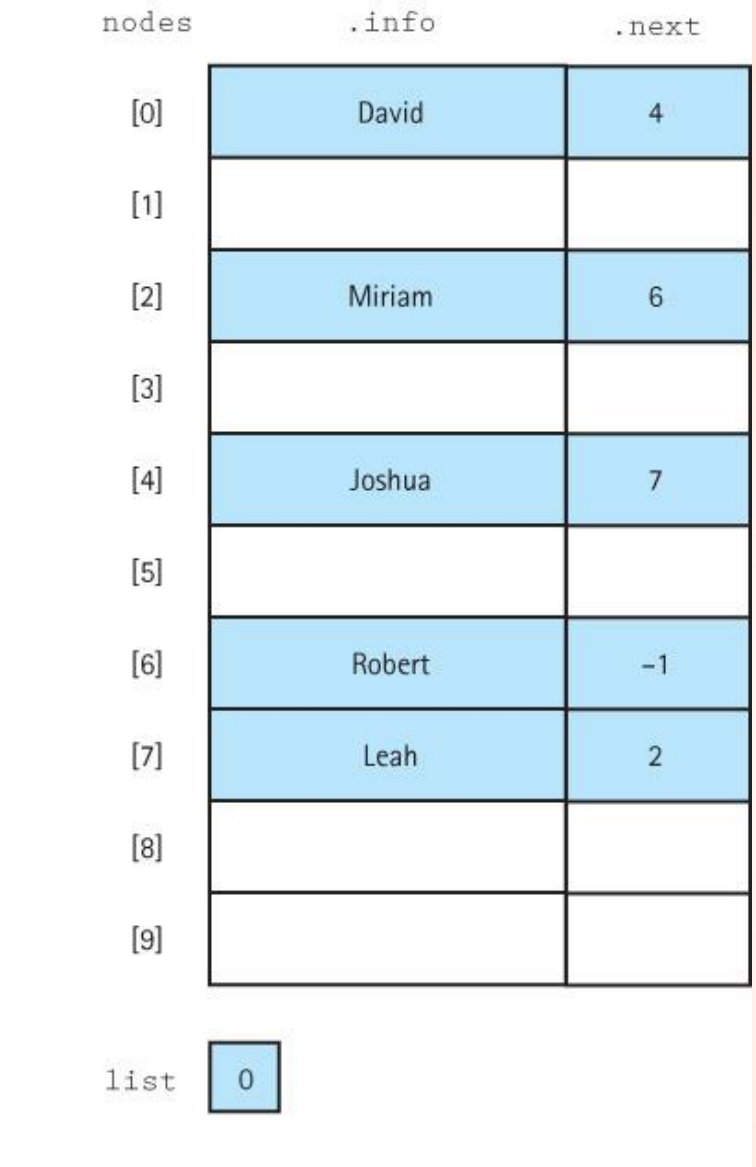
Why Use an Array?

- Sometimes managing the free space ourselves gives us greater flexibility
- There are programming languages that do not support dynamic allocation or reference types
- There are times when dynamic allocation of each node, one at a time, is too costly in terms of time

Boundedness

- A desire for static allocation is one of the primary motivations for the array-based linked approach
- We drop our assumption that our lists are of unlimited size in this section - our lists will not grow as needed.
- Applications should not add elements to a full list.

A sorted list



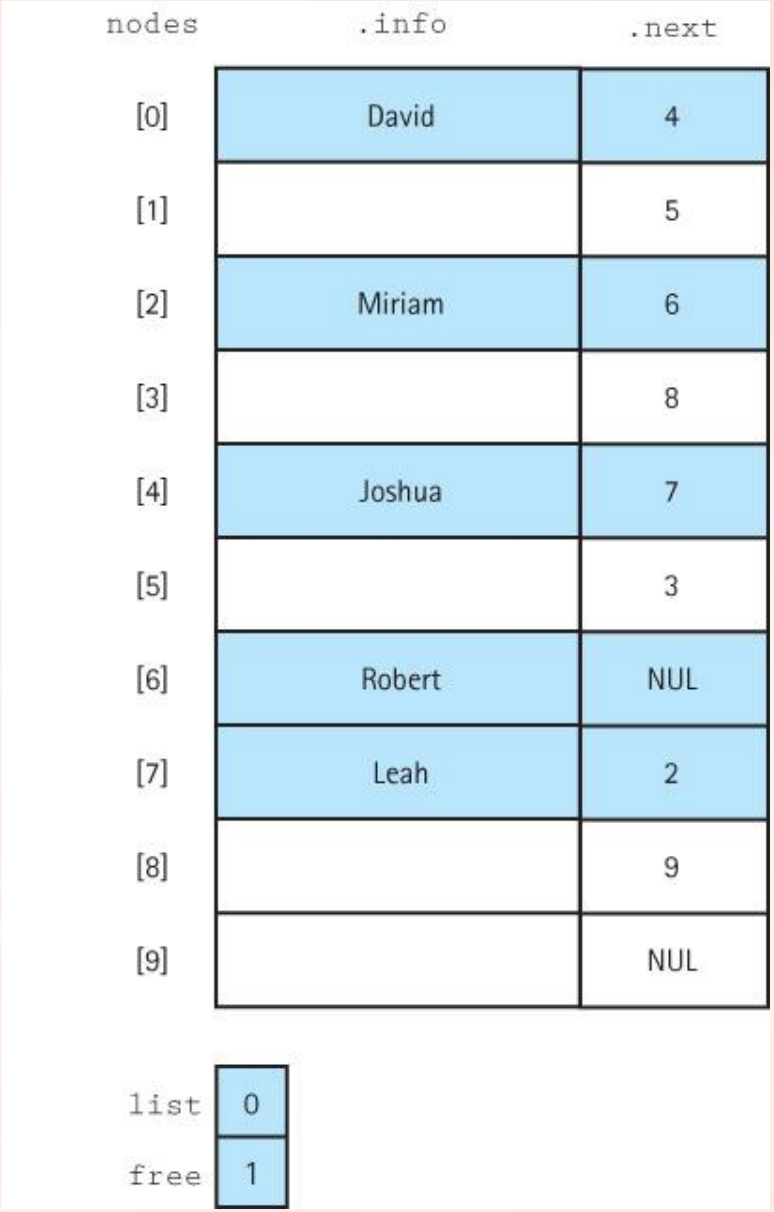
Implementation Issues

- We mark the end of the list with a “null” value
  - the “null” value must be an invalid address for a real list element
  - we use the value –1
  - we suggest using the identifier NUL and defining it to be -1

```
private static final int NUL = -1;
```

- One must directly manage the free space available for new list elements.
  - Link the collection of unused array elements together into a linked list of free nodes.
  - Write your own method to allocate nodes from the free space. We suggest calling this method `getNode`.
  - Write your own method, we suggest calling it `freeNode`, to put a node back into the pool of free space when it is de-allocated.

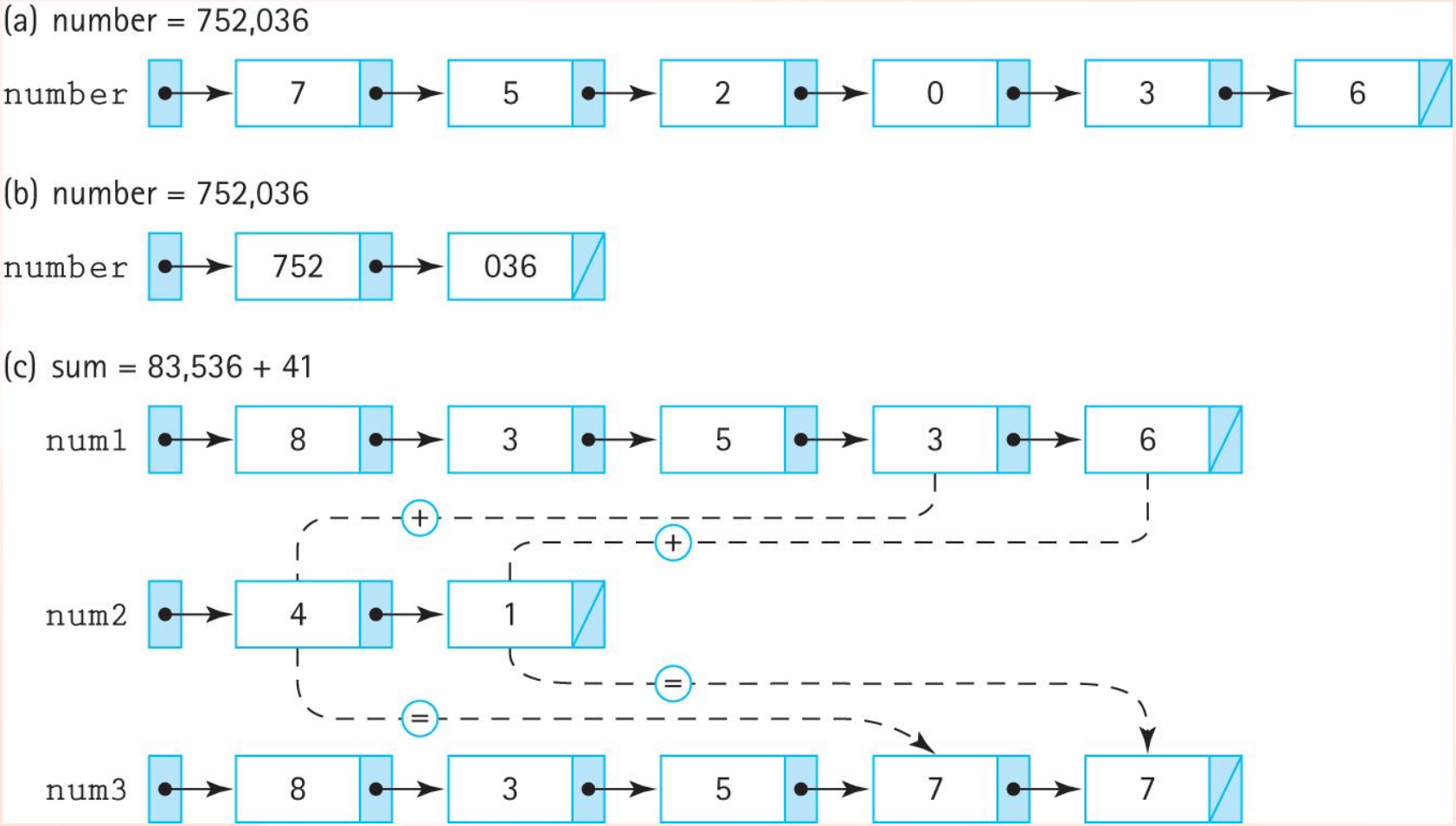
A linked list and free space



6.6 Application: Large Integers

- The largest Java integer type, `long`, can represent values between −9,223,372,036,854,775,808 and 9,223,372,036,854,775,807
- Believe it or not, for some applications that may not be sufficient
- A linked list of digits can grow to be any size, and thus can be used to represent integers of any size

Representing large integers with linked lists



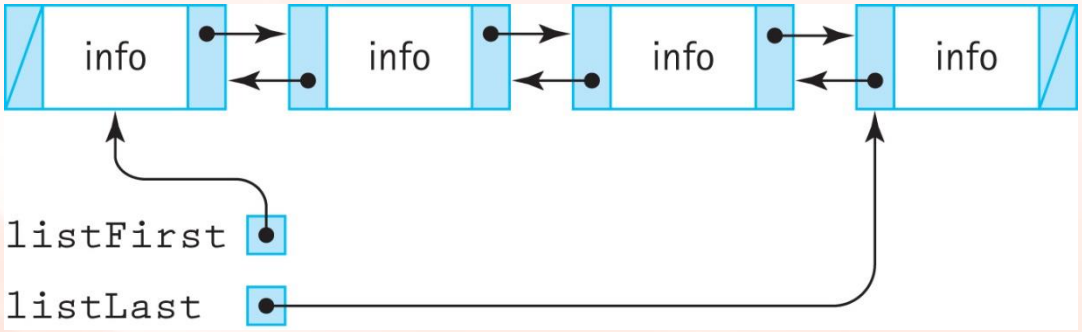
The `LargeInt` class

- Constructors – one that creates an "empty" integer and one that creates an integer based on a `String` argument
- `setNegative` – makes the large integer negative
- `toString` – returns string representation
- `add` – returns the sum of two large integers
- `subtract` – returns the difference of two large integers

- To support the creation and arithmetic manipulation of large integers we define a special list class …

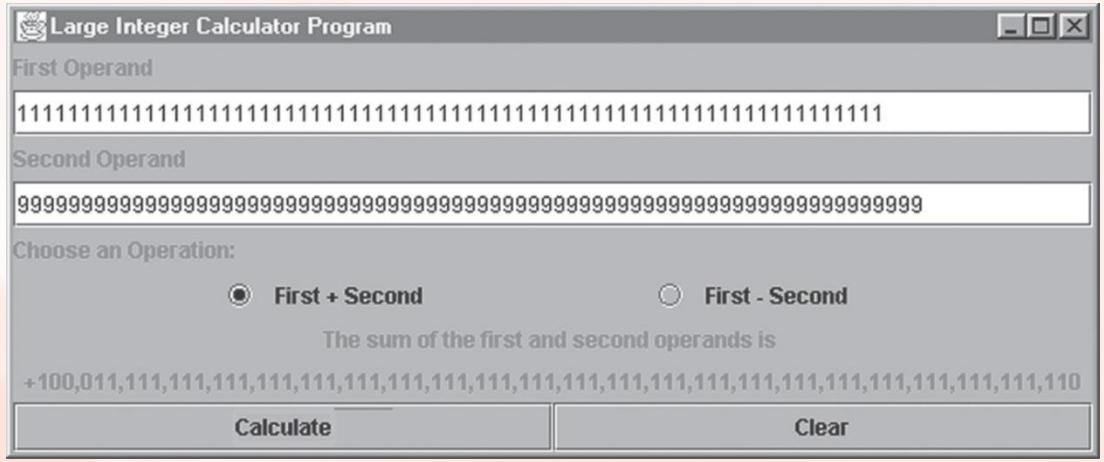
The `LargeIntList` class

- a list of `byte` (to hold digits)
- provide operations `size`, `addFront`, `addEnd`, and both `forward` and `reverse` iterators
- To support these requirements we use a reference-based doubly linked structure



Applications

- `LargeIntCLI` - in the `ch06.apps` package, allows the user to enter two large integers, performs the addition and subtraction of the two integers, and reports the results.
- `LargeIntGUI` - in the `ch06.apps` package:



Code and Demo

- Instructors can now review the algorithms, walk through the code for the classes, and demonstrate the running applications.

Important Concept Revisited: Abstraction Hierarchy

- Here we saw another example of an abstraction hierarchy.
- Applications use the LargeInt class, which uses the LargeIntList class, which uses the DLLNode class.
- When working at any level in this hierarchy as a programmer we need to know only how to use the next lower level – we do not need to know how it is implemented nor need we worry about the details of lower levels.
- Abstraction is indeed the key to conquering complexity.