University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Textbook slides and Dr. Ramirez's CS 0445 slides)

Announcements

- Upcoming Deadlines:
  - Lab 9: next Monday 11/21 @ 11:59 pm
  - Homework 9 (to be posted soon): next Monday 11/21 @ 11:59 pm

Sorting Algorithms

- $O(n^2)$
  - Selection Sort
  - Insertion Sort
  - Shell Sort
- $O(n \log n)$
  - Merge Sort
  - Quick Sort
- O(n) Sorting
  - Radix Sort

Muddiest Points

- **Q: Could you explain why the runtime of insertionsort is O(n^2)?**
- insertion Sort has two nested loops:
  - the outer loop has n-1 iterations
  - First iteration → 1 comparison in the worst case
  - Second iteration → 2 comparisons
  - Third iteration → 3 comparisons
  - …
  - iteration n-1 → n-1 comparisons
- the inner loop has *i* iterations in the worst case, where i is the outer loop counter
- Total # comparisons = 1 + 2 + 3 + … + n-1 = $O(n^2)$

Muddiest Points

- **Q: why merge sort is not tail-recursive**
- Because after the second recursive call, the merge method is called

Muddiest Points

- **Q: When should merge sort be used instead of quick sort?**
- When a stable sort is required

Muddiest Points

- **Q:could you show us the average runtime of ShellSort and MergeSort?**
- Shell Sort is $O(n^{1.5})$ on average
- Merge Sort is O(n log n) on average
- Proofs are outside the scope of this course

Muddiest Points

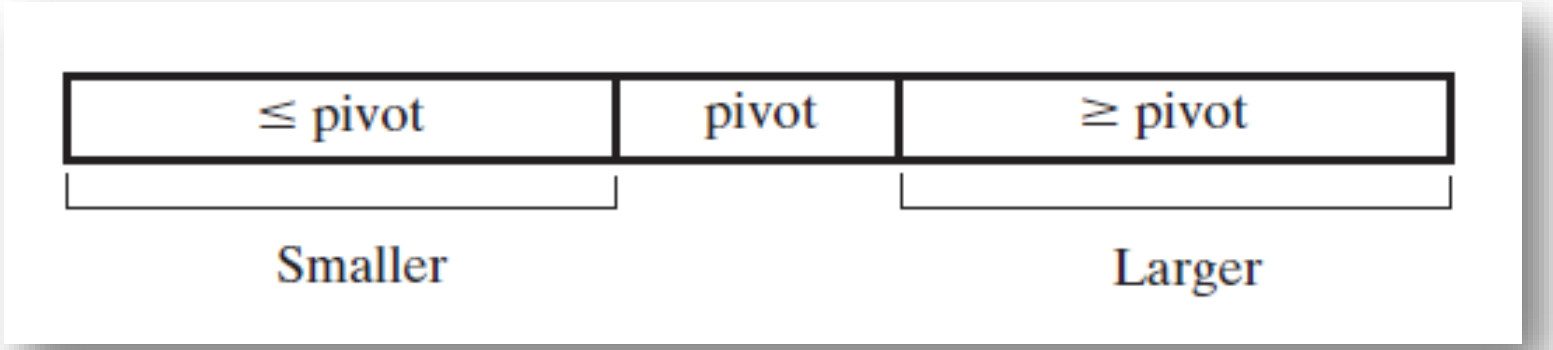
- **Q: I don't get how you can make an array of Comparable<?>[]**
- Comparable<?> is the upper bound on the type parameter T
- Comparable<?>[] is an array of references to objects that implement the Comparable interface

Muddiest Points

- **Q: how do you determine the runtime of merge sort?**
- We will do that today!

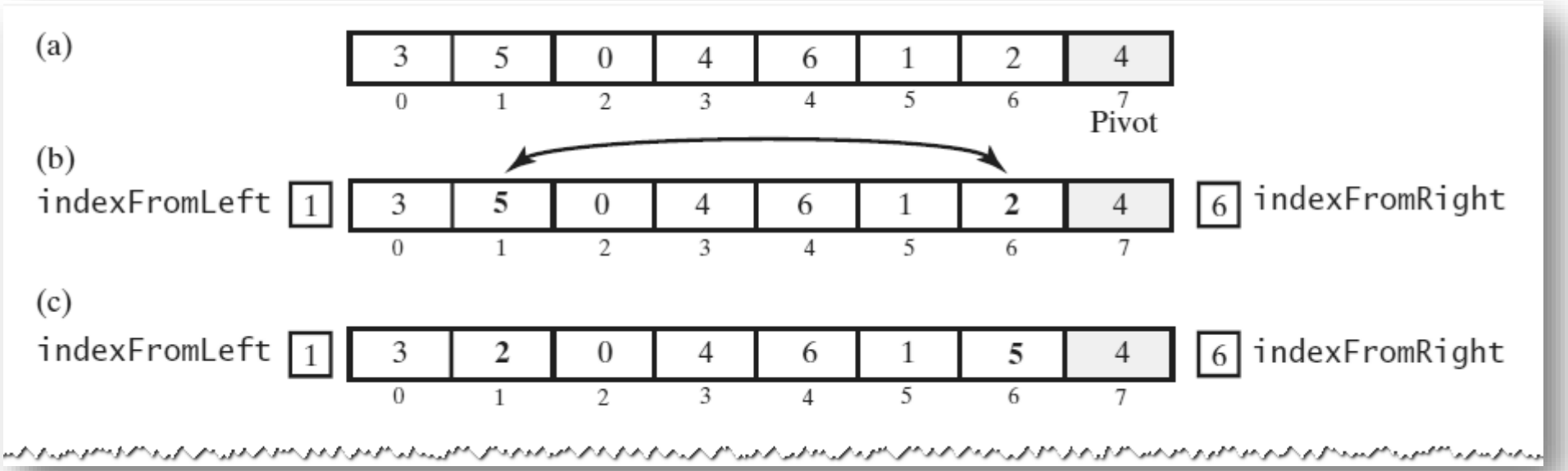
Quick Sort

- A partition of an array during a quick sort



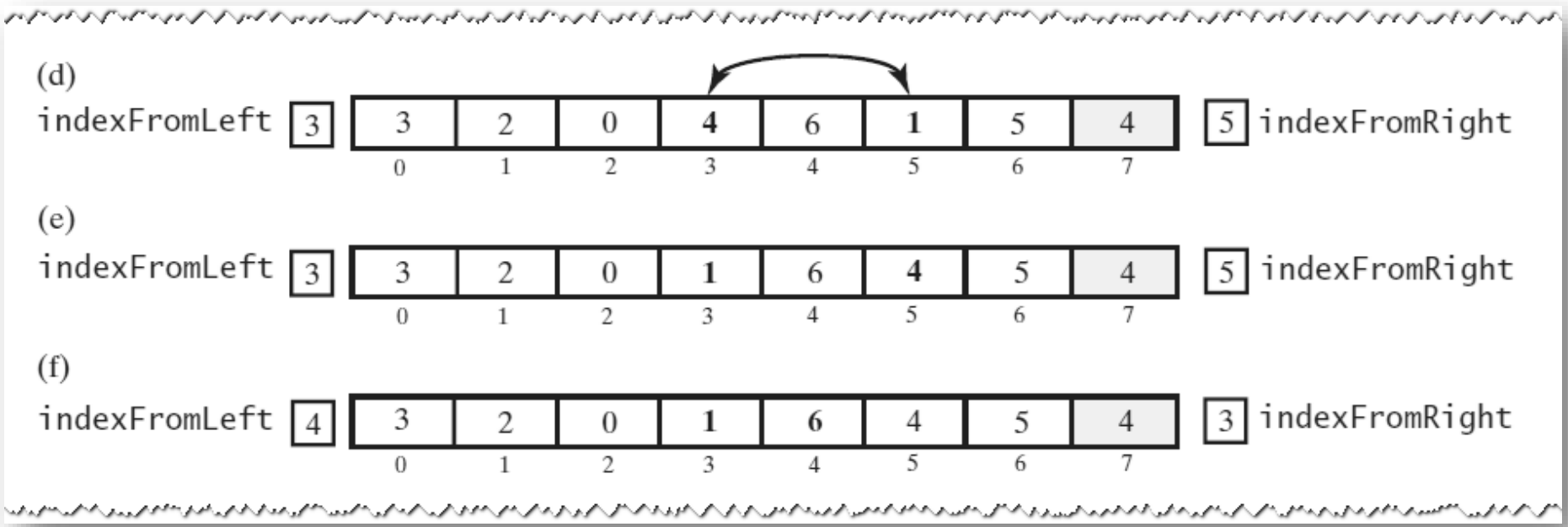
Creating the Partition

- A partitioning strategy for quick sort



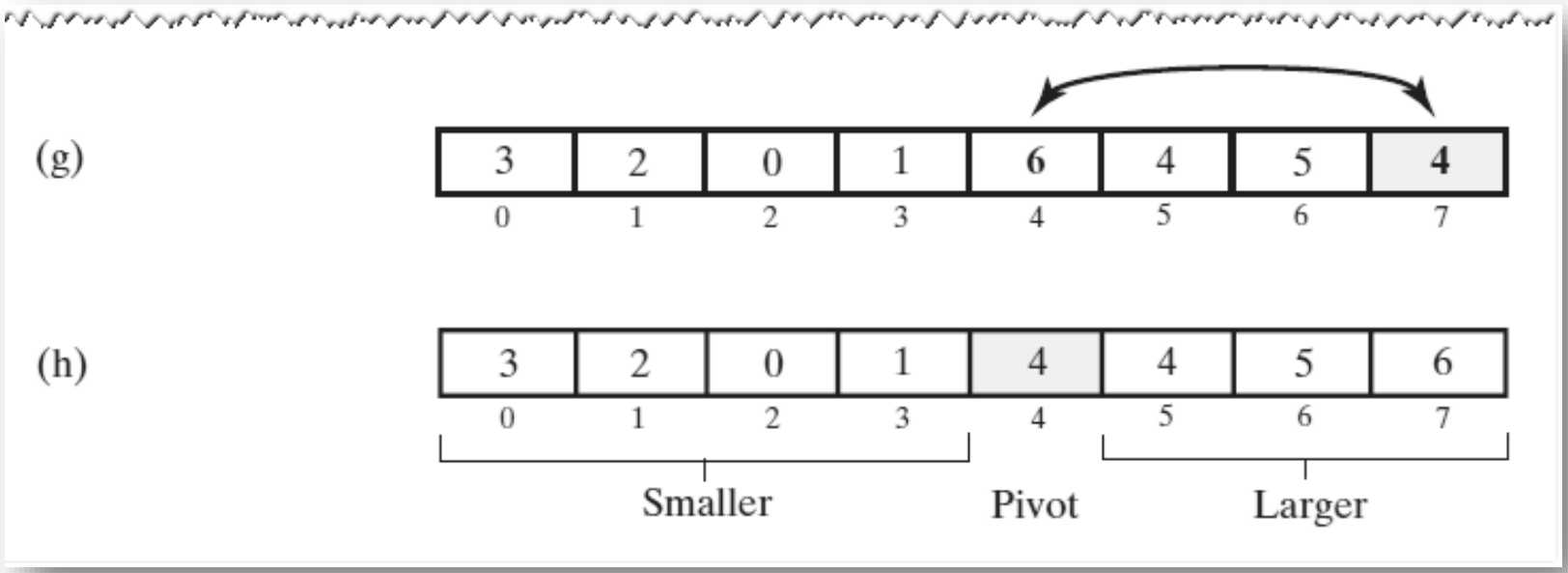
Creating the Partition

- A partitioning strategy for quick sort



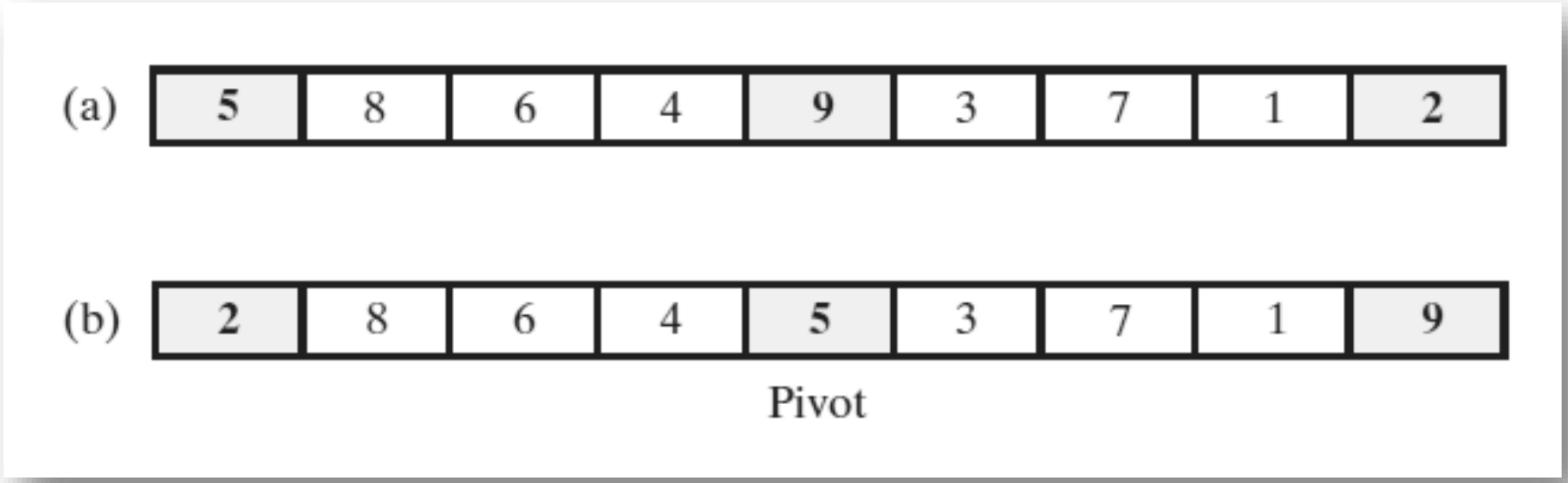
Creating the Partition

- A partitioning strategy for quick sort



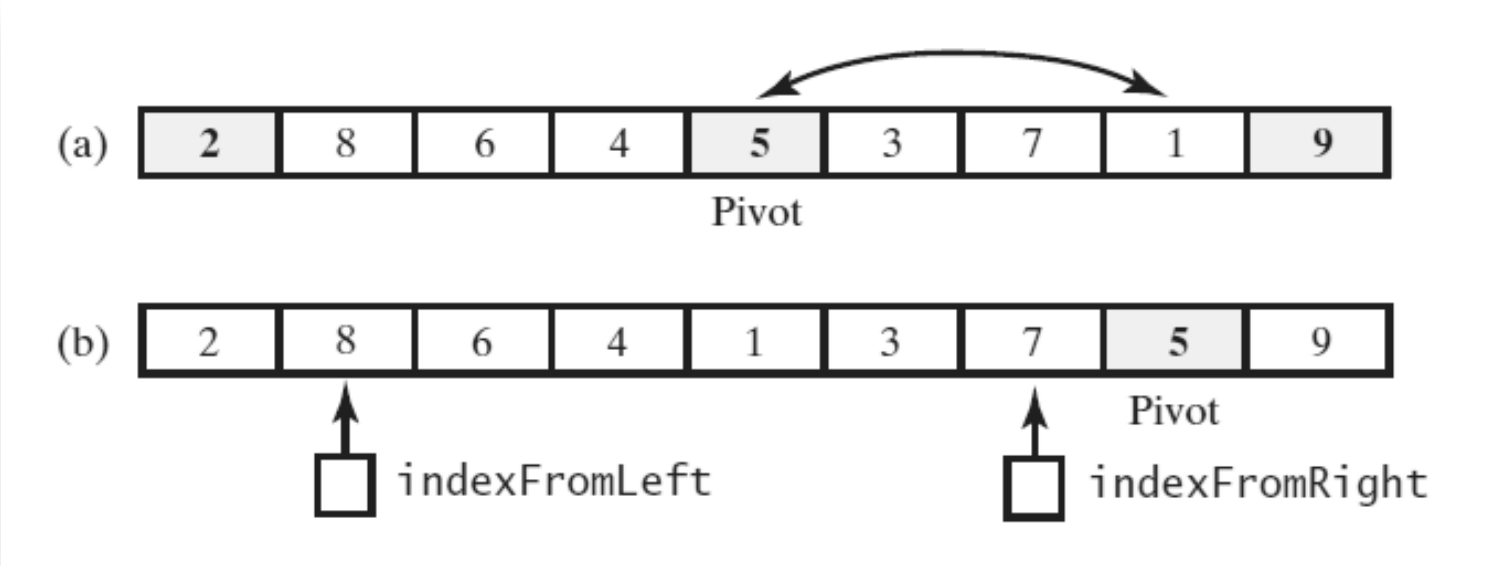
Creating the Partition

- Median-of-three pivot selection: (a) The original array; (b) the array with its first, middle, and last entries sorted



Adjusting the Partition Algorithm

- The array with its first, middle, and last entries sorted; (b) the array after positioning the pivot and just before partitioning



Adjusting the Partition Algorithm

```
Algorithm partition(a, first, last)
// Partitions an array a[first..last] as part of quick sort into two subarrays named
// Smaller and Larger that are separated by a single entry—the pivot— named pivotValue.
// Entries in Smaller are <= pivotValue and appear before pivotValue in the array.
// Entries in Larger are >= pivotValue and appear after pivotValue in the array.
// first >= 0; first < a.length; last - first >= 3; last < a.length.
// Returns the index of the pivot.

   mid = index of the array's middle entry
   sortFirstMiddleLast(a, first, mid, last)
   // Assertion: a[mid] is the pivot, that is, pivotValue;
   // a[first] <= pivotValue and a[last] >= pivotValue, so do not compare these two
   // array entries with pivotValue.

   // Move pivotValue to next-to-last position in array
```

Adjusting the Partition Algorithm

```
// Move pivotValue to next-to-last position in array
Exchange a[mid] and a[last - 1]
pivotIndex = last - 1
pivotValue = a[pivotIndex]

// Determine two subarrays:
//    Smaller = a[first..endSmaller] and
//    Larger  = a[endSmaller+1..last-1]
// such that entries in Smaller are <= pivotValue and
// entries in Larger are >= pivotValue.
// Initially, these subarrays are empty.
indexFromLeft = first + 1
indexFromRight = last - 2
done = false
while (!done)
```

Adjusting the Partition Algorithm

```
while (!done)
{
   // Starting at the beginning of the array, leave entries that are < pivotValue and
   // locate the first entry that is >= pivotValue. You will find one, since the last
   // entry is >= pivotValue.
   while (a[indexFromLeft] < pivotValue)
      indexFromLeft++

   // Starting at the end of the array, leave entries that are > pivotValue and
   // locate the first entry that is <= pivotValue. You will find one, since the first
   // entry is <= pivotValue.
   while (a[indexFromRight] > pivotValue)
      indexFromRight--

   // Assertion: a[indexFromLeft] >= pivotValue and
   //            a[indexFromRight] <= pivotValue
   if (indexFromLeft < indexFromRight)
```

Adjusting the Partition Algorithm

```
    //          a[indexFromRight] <= pivotValue
    if (indexFromLeft < indexFromRight)
    {
       Exchange a[indexFromLeft] and a[indexFromRight]
       indexFromLeft++
       indexFromRight--

    }
    else
       done = true
 }

Exchange a[pivotIndex] and a[indexFromLeft]
pivotIndex = indexFromLeft

// Assertion: Smaller = a[first..pivotIndex-1]
//            pivotValue = a[pivotIndex]
//            Larger = a[pivotIndex+1..last]
return pivotIndex
```

The Quick Sort Method

- Above method implements quick sort.

```
/** Sorts an array into ascending order. Uses quick sort with
    median-of-three pivot selection for arrays of at least
    MIN_SIZE entries, and uses insertion sort for smaller arrays. */
public static <T extends Comparable<? super T>>
       void quickSort(T[] a, int first, int last)
{
   if (last - first + 1 < MIN_SIZE)
   {
      insertionSort(a, first, last);
   }
   else
   {
       // Create the partition: Smaller | Pivot | Larger
       int pivotIndex = partition(a, first, last);

       // Sort subarrays Smaller and Larger
       quickSort(a, first, pivotIndex - 1);
       quickSort(a, pivotIndex + 1, last);
   } // end if
} // end quickSort
```

QuickSort in the Java Class Library

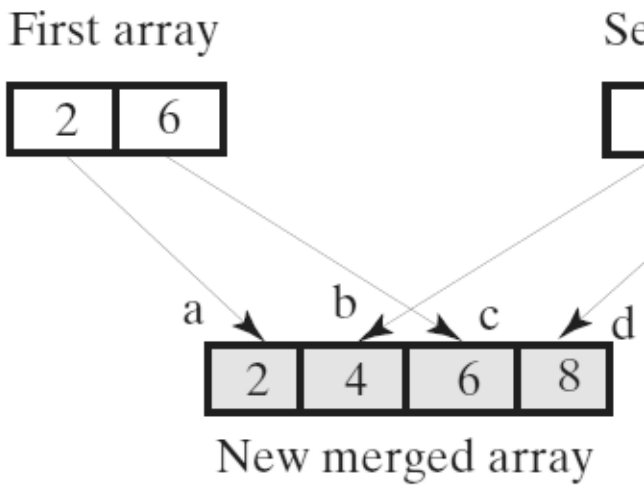
- Class **`Arrays`** in the package **`java.util`** uses a quick sort to sort arrays of primitive types into ascending order

```
public static void sort(type[] a)

public static void sort(type[] a, int first, int after)
```

Efficiency of Merge Sort

- A worst-case merge of two sorted arrays.
- Efficiency is $O(n \log n)$.



a. $2 < 4$, so copy 2 to new array

b. $6 > 4$, so copy 4 to new array

c. $6 < 8$, so copy 6 to new array

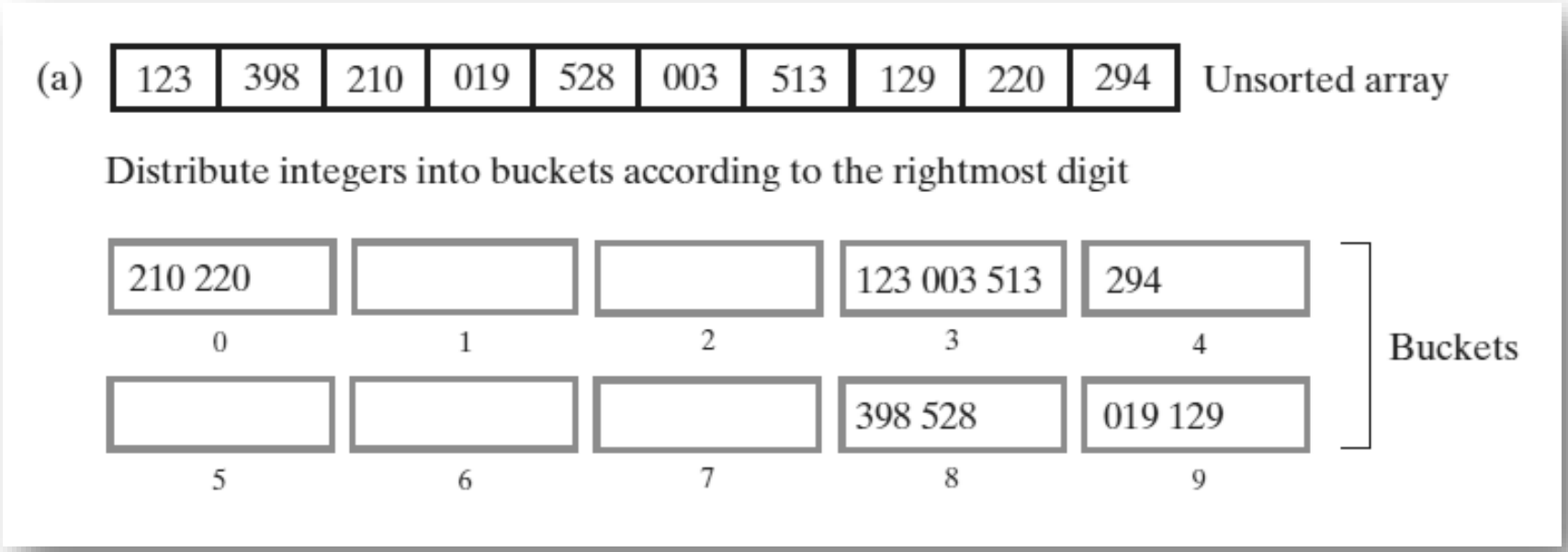
d. Copy 8 to new array

Radix Sort

- Does not use comparison
- Treats array entries as if they were strings that have the same length.
  - Group integers according to their rightmost character (digit) into “buckets”
  - Repeat with next character (digit), etc.

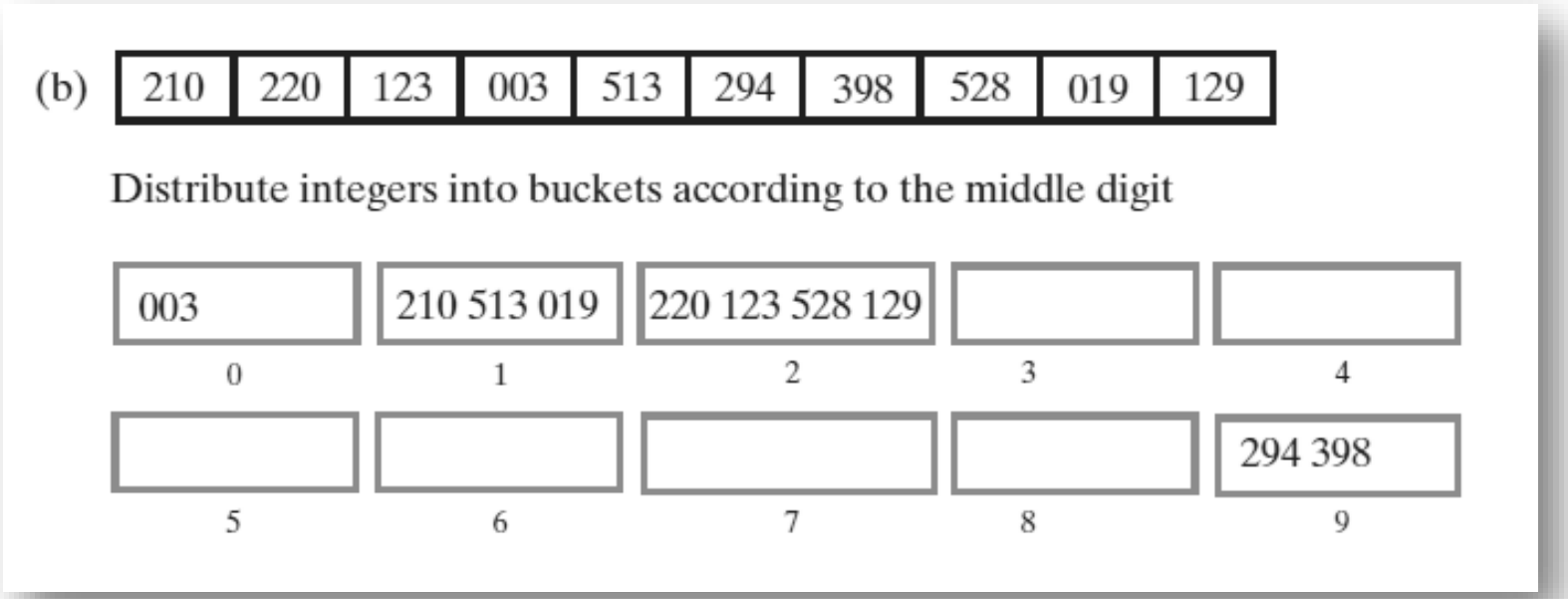
Radix Sort

- Radix sort: (a) Original array and buckets after first distribution;



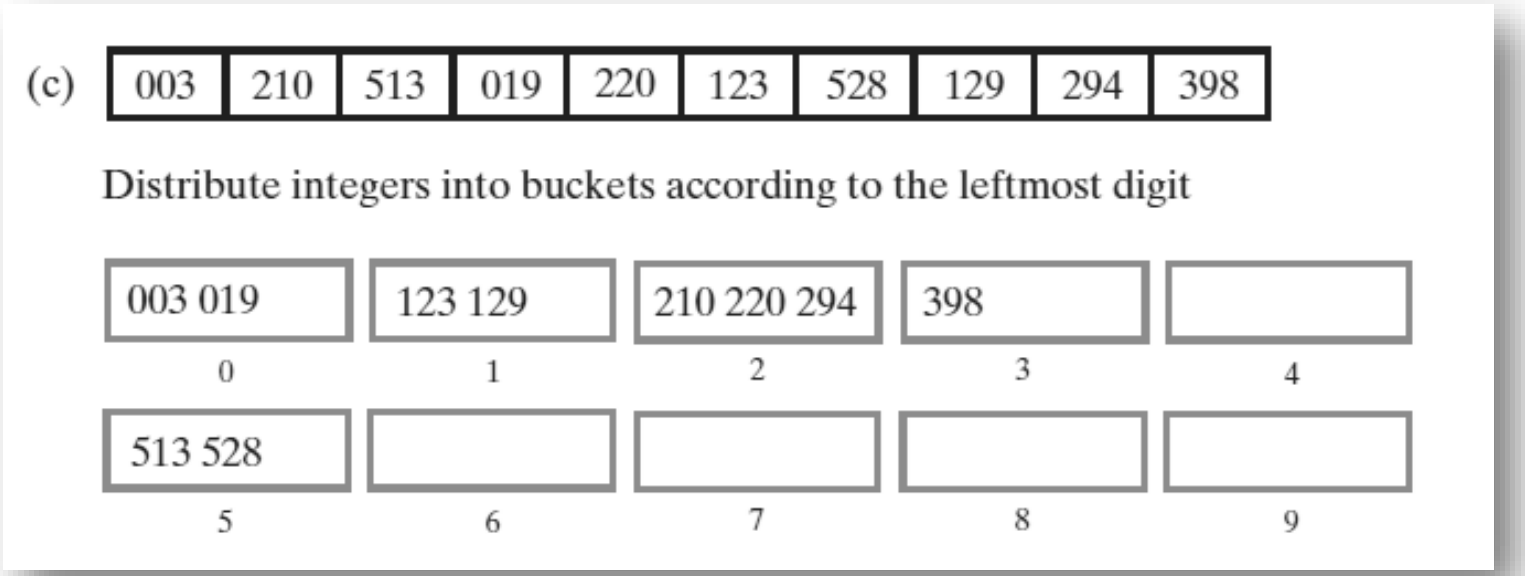
Radix Sort

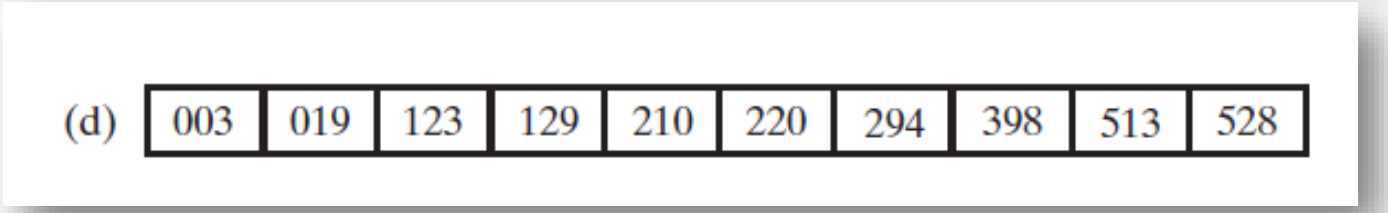
- Radix sort: (b) reordered array and buckets after second distribution;



Radix Sort

- Radix sort: (c) reordered array and buckets after third distribution; (d) sorted array





Pseudocode for Radix Sort

- Radix sort is an O(n) algorithm for certain data, it is not appropriate for all data

```
Algorithm radixSort(a, first, last, maxDigits)
// Sorts the array of positive decimal integers a[first..last] into ascending order;
// maxDigits is the number of digits in the longest integer.

for (i = 0 to maxDigits - 1)
{
    Clear bucket[0], bucket[1], . . . , bucket[9]
    for (index = first to last)
    {
        digit = digit i of a[index]
        Place a[index] at end of bucket[digit]
    }
    Place contents of bucket[0], bucket[1], . . . , bucket[9] into the array a
}
```

Comparing the Algorithms

- The time efficiency of various sorting algorithms, expressed in Big Oh notation

	Average Case	Best Case	Worst Case
Radix sort	$O(n)$	$O(n)$	$O(n)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Shell sort	$O(n^{1.5})$	$O(n)$	$O(n^2)$ or $O(n^{1.5})$
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Iterators

What Is an Iterator?

- An object that traverses a collection of data
- During iteration, each data item is considered once
  - Possible to modify item as accessed
- Should implement as a distinct class that interacts with the ADT

The Interface `Iterator`

Java's interface **`java.util.Iterator`**

```
1  package java.util;
2  public interface Iterator<T>
3  {
4     /** Detects whether this iterator has completed its traversal
5         and gone beyond the last entry in the collection of data.
6         @return  True if the iterator has another entry to return. */
7     public boolean hasNext();
8
9     /** Retrieves the next entry in the collection and
10        advances this iterator by one position.
11        @return  A reference to the next entry in the iteration,
12                 if one exists.
13        @throws  NoSuchElementException if the iterator had reached the
14                 end already, that is, if hasNext() is false. */
15    public T next();
16
17    /** Removes from the collection of data the last entry that
18        next() returned. A subsequent call to next() will behave
19        as it would have before the removal.
20        Precondition: next() has been called, and remove() has not
```

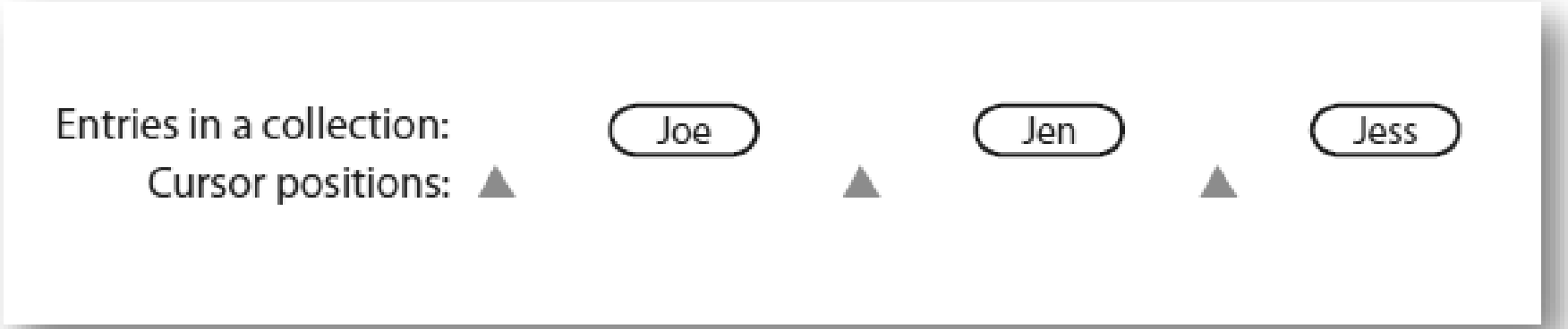
The Interface `Iterator`

Java's interface **`java.util.Iterator`**

```
12                  if one exists.
13          @throws  NoSuchElementException if the iterator had reached the
14                   end already, that is, if hasNext() is false. */
15      public T next();
16
17      /** Removes from the collection of data the last entry that
18          next() returned. A subsequent call to next() will behave
19          as it would have before the removal.
20          Precondition: next() has been called, and remove() has not
21          been called since then. The collection has not been altered
22          during the iteration except by calls to this method.
23          @throws  IllegalStateException if next() has not been called, or
24                   if remove() was called already after the last call to next().
25          @throws  UnsupportedOperationException if the iterator does
26                   not permit a remove operation. */
27      public void remove(); // Optional method
28  } // end Iterator
```

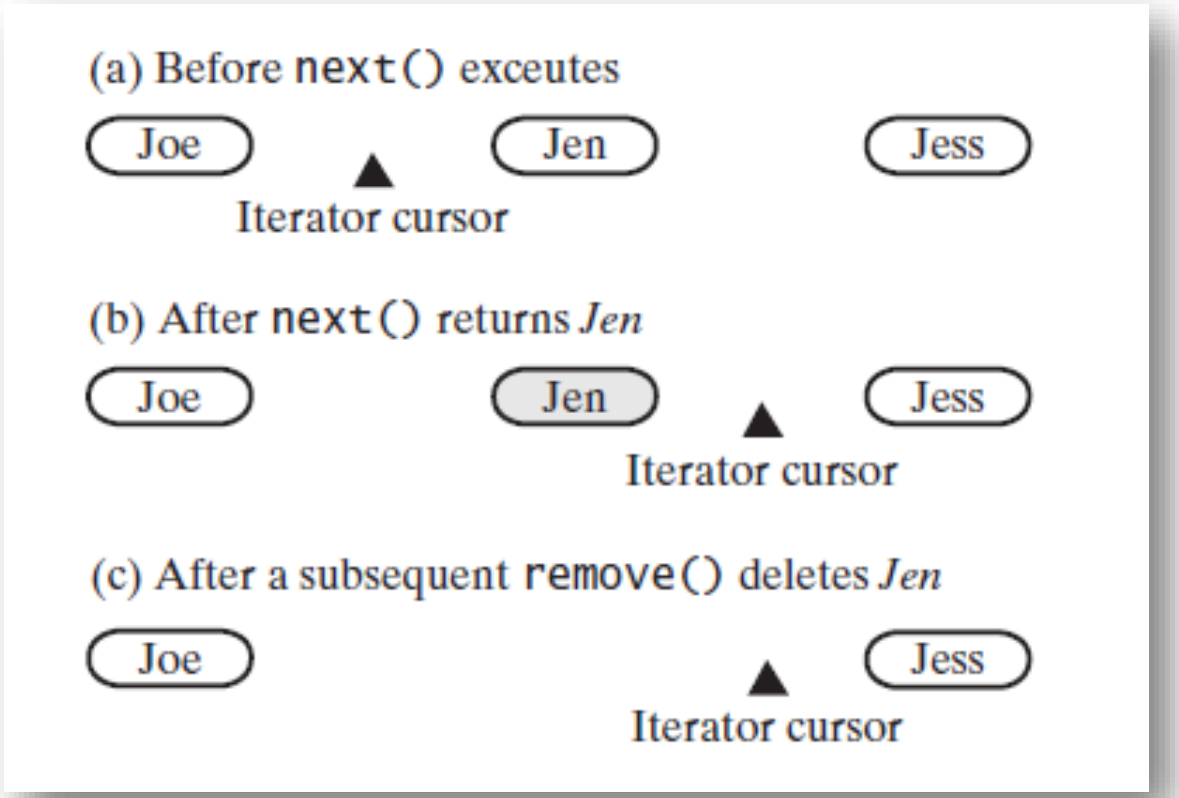
The Interface `Iterator`

Possible positions of an iterator's cursor within a collection



The Interface `Iterator`

The effect on a collections iterator by a call to **`next`** and a subsequent call to **`remove`**



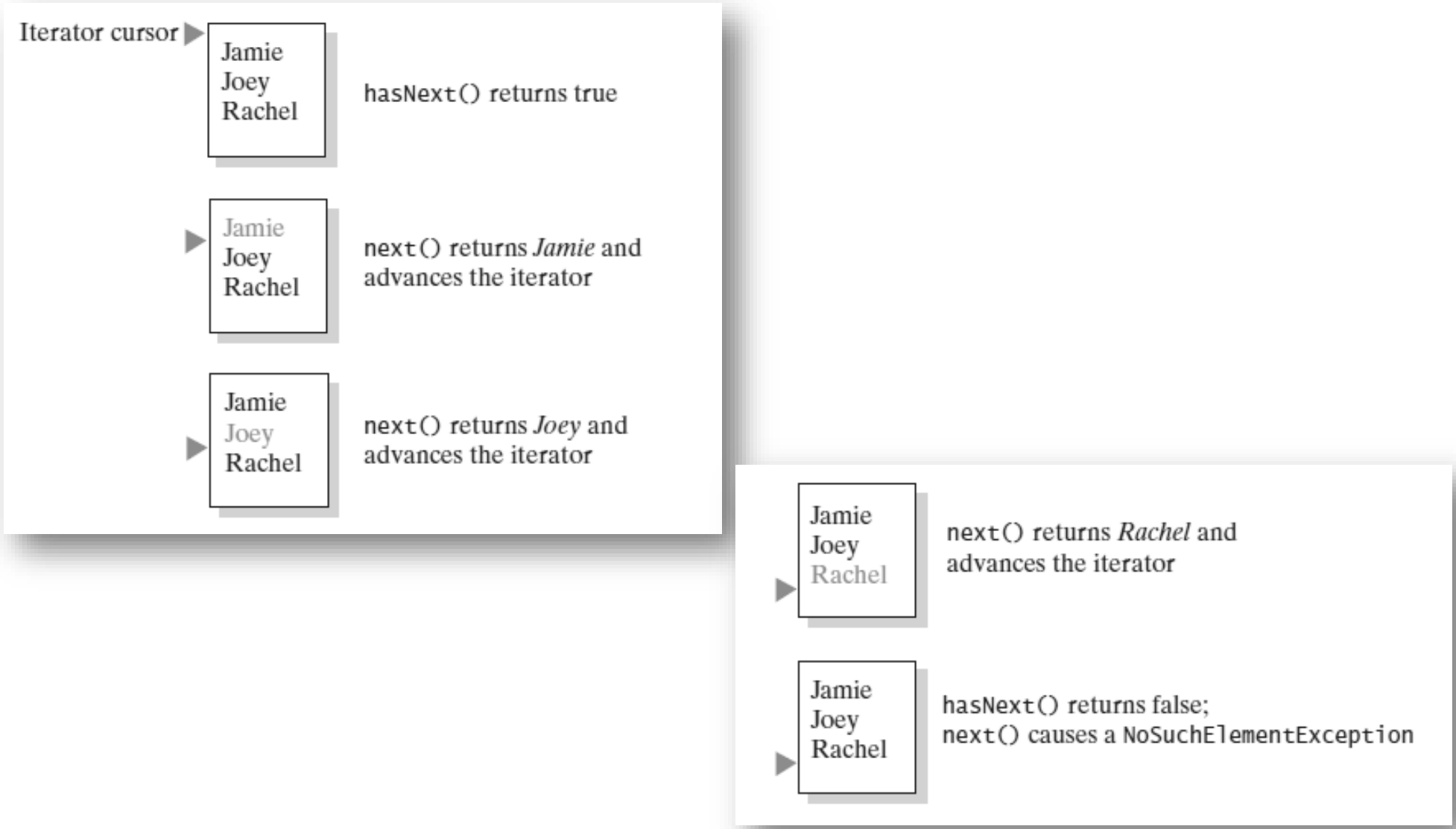
The Interface `Iterable`

The interface `java.lang.Iterable`

```
1 package java.lang;
2 public interface Iterable<T>
3 {
4    /** @return  An iterator for a collection of objects of type T. */
5    Iterator<T> iterator();
6 } // end Iterable
```

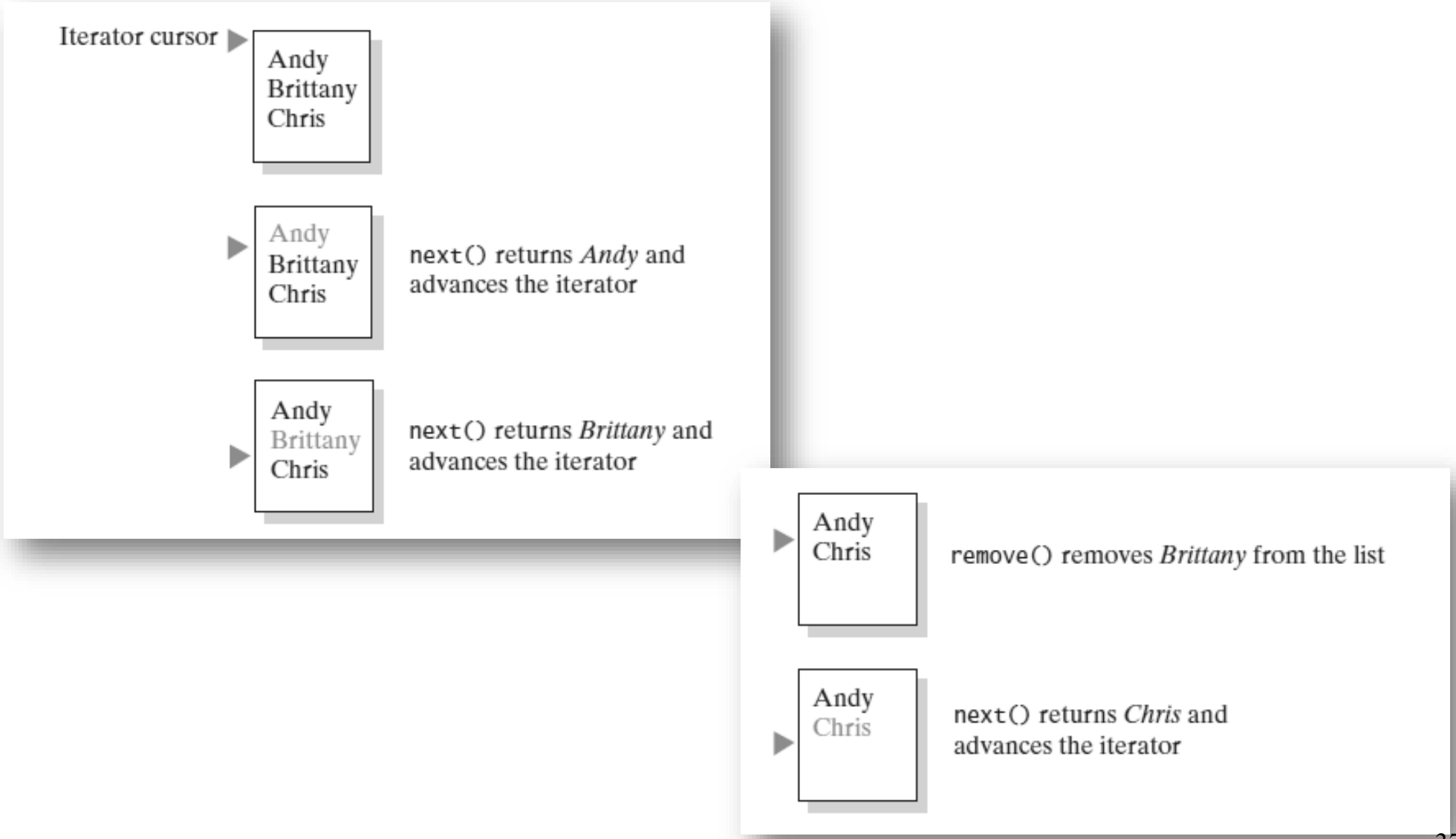
Using the Interface `Iterator`

The effect of the iterator methods **`hasNext`** and **`next`** on a list



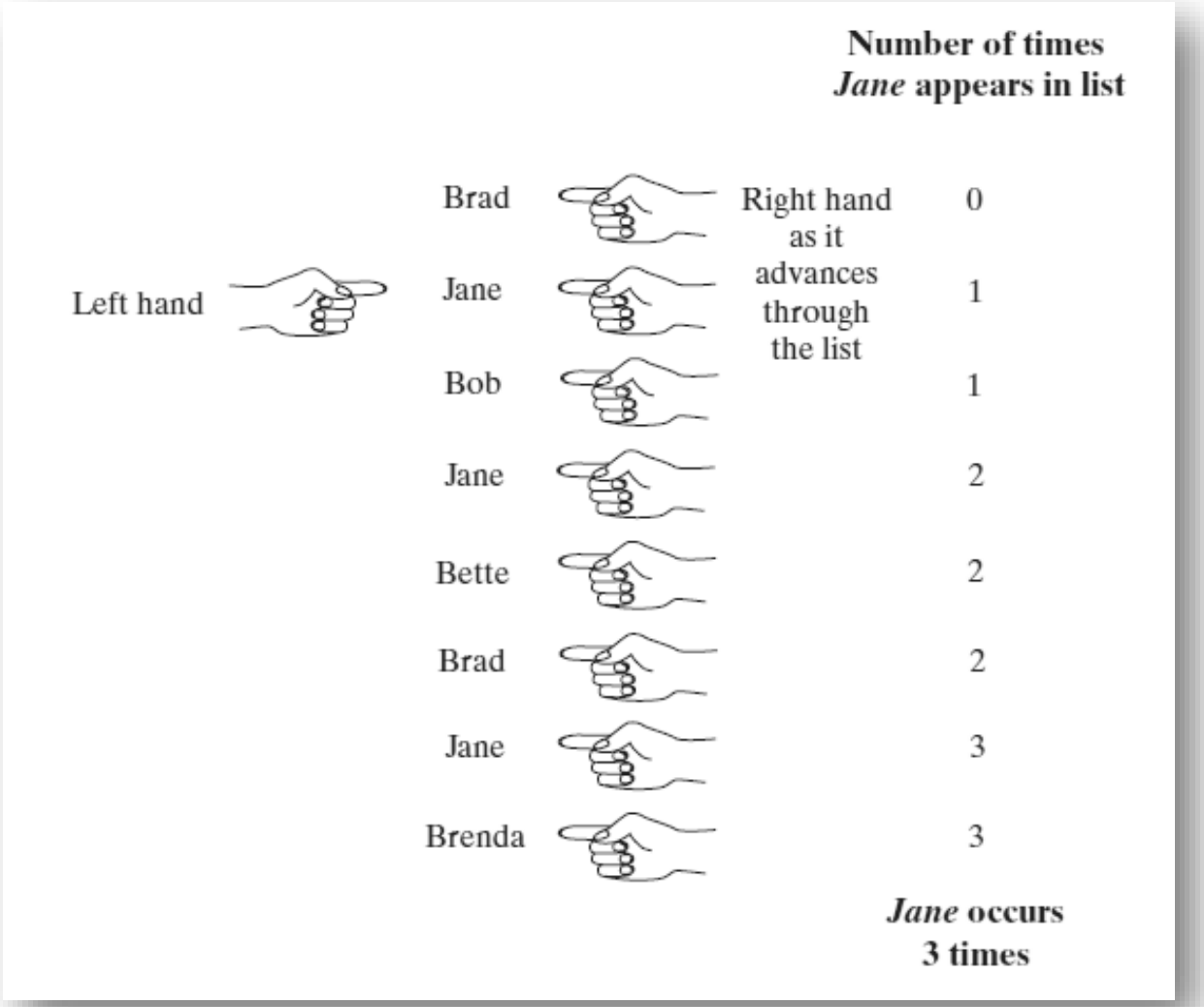
Using the Interface `Iterator`

The effect of the iterator methods `next` and `remove` on a list



Multiple Iterators

Counting the number of times
that *Jane* appears in a list of names



Multiple Iterators

Code that counts the occurrences of each name

```
Iterator<String> nameIterator = namelist.iterator();
while (nameIterator.hasNext())
{
   String currentName = nameIterator.next();
   int nameCount = 0;
   Iterator<String> countingIterator = namelist.iterator();
   while (countingIterator.hasNext())
   {
      String nextName = countingIterator.next();
      if (currentName.equals(nextName))
         nameCount++;
   } // end while
   System.out.println(currentName + " occurs " + nameCount + " times.");
} // end while
```

The Interface `ListIterator`

Java's interface **`java.util.ListIterator`**

```
1  package java.util;
2  public interface ListIterator<T> extends Iterator<T>
3  {
4     /** Detects whether this iterator has gone beyond the last
5         entry in the list.
6         @return  True if the iterator has another entry to return when
7                  traversing the list forward; otherwise returns false. */
8     public boolean hasNext();
9
10    /** Retrieves the next entry in the list and
11        advances this iterator by one position.
12        @return  A reference to the next entry in the iteration,
13                 if one exists.
14        @throws  NoSuchElementException if the iterator had reached the
15                 end already, that is, if hasNext() is false. */
16    public T next();
17
18    /** Removes from the list the last entry that either next()
19        or previous() has returned.
20        Precondition: next() or previous() has been called, but the
21        iterator's remove() or add() method has not been called
```

The Interface `ListIterator`

Java's interface **`java.util.ListIterator`**

```
18    /** Removes from the list the last entry that either next()
19        or previous() has returned.
20        Precondition: next() or previous() has been called, but the
21        iterator's remove() or add() method has not been called
22        since then. That is, you can call remove only once per
23        call to next() or previous(). The list has not been altered
24        during the iteration except by calls to the iterator's
25        remove(), add(), or set() methods.
26        @throws  IllegalStateException if next() or previous() has not
27                 been called, or if remove() or add() has been called
28                 already after the last call to next() or previous().
29        @throws  UnsupportedOperationException if the iterator does not
30                 permit a remove operation. */
31    public void remove(); // Optional method
32
33  // The previous three methods are in the interface Iterator; they are
34  // duplicated here for reference and to show new behavior for remove.
```

The Interface ListIterator

Java's interface **java.util.ListIterator**

```
35
36    /** Detects whether this iterator has gone before the first
37        entry in the list.
38        @return  True if the iterator has another entry to visit when
39                 traversing the list backward; otherwise returns false. */
40    public boolean hasPrevious();
41
42    /** Retrieves the previous entry in the list and moves this
43        iterator back by one position.
44        @return  A reference to the previous entry in the iteration, if
45                 one exists.
46        @throws  NoSuchElementException if the iterator has no previous
47                 entry, that is, if hasPrevious() is false. */
48    public T previous();
49
50    /** Gets the index of the next entry.
51        @return  The index of the list entry that a subsequent call to
52                 next() would return. If next() would not return an entry
53                 because the iterator is at the end of the list, returns
54                 the size of the list. Note that the iterator numbers
55                 the list entries from 0 instead of 1. */
56    public int nextIndex();
```

The Interface ListIterator

Java's interface **java.util.ListIterator**

```
58    /** Gets the index of the previous entry.
59        @return  The index of the list entry that a subsequent call to
60                 previous() would return. If previous() would not return
61                 an entry because the iterator is at the beginning of the
62                 list, returns -1. Note that the iterator numbers the
63                 list entries from 0 instead of 1. */
64    public int previousIndex();
65
66    /** Adds an entry to the list just before the entry, if any,
67        that next() would have returned before the addition. This
68        addition is just after the entry, if any, that previous()
69        would have returned. After the addition, a call to
70        previous() will return the new entry, but a call to next()
71        will behave as it would have before the addition.
72        Further, the addition increases by 1 the values that
73        nextIndex() and previousIndex() will return.
74        @param newEntry  An object to be added to the list.
75        @throws  ClassCastException if the class of newEntry prevents the
76                 addition to the list.
77        @throws  IllegalArgumentException if some other aspect of
78                 newEntry prevents the addition to the list.
79        @throws  UnsupportedOperationException if the iterator does not
80                 permit an add operation. */
81    public void add(T newEntry); // Optional method
82
```

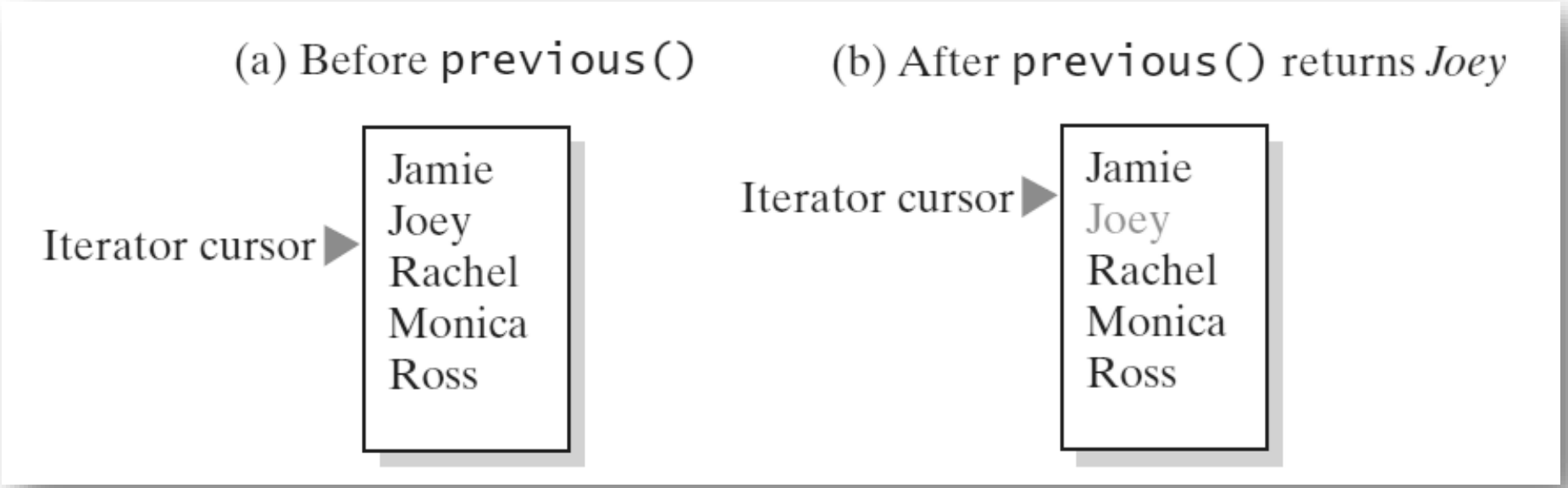
The Interface `ListIterator`

Java's interface **`java.util.ListIterator`**

```
83    /** Replaces the last entry in the list that either next()
84        or previous() has returned.
85        Precondition: next() or previous() has been called, but the
86        iterator's remove() or add() method has not been called since then.
87        @param newEntry  An object that is the replacement entry.
88        @throws  ClassCastException if the class of newEntry prevents the
89                 addition to the list.
90        @throws  IllegalArgumentException if some other aspect of newEntry
91                 prevents the addition to the list.
92        @throws  IllegalStateException if next() or previous() has not
93                 been called, or if remove() or add() has been called
94                 already after the last call to next() or previous().
95        @throws  UnsupportedOperationException if the iterator does not
96                 permit a set operation. */
97    public void set(T newEntry); // Optional method
98 } // end ListIterator
```

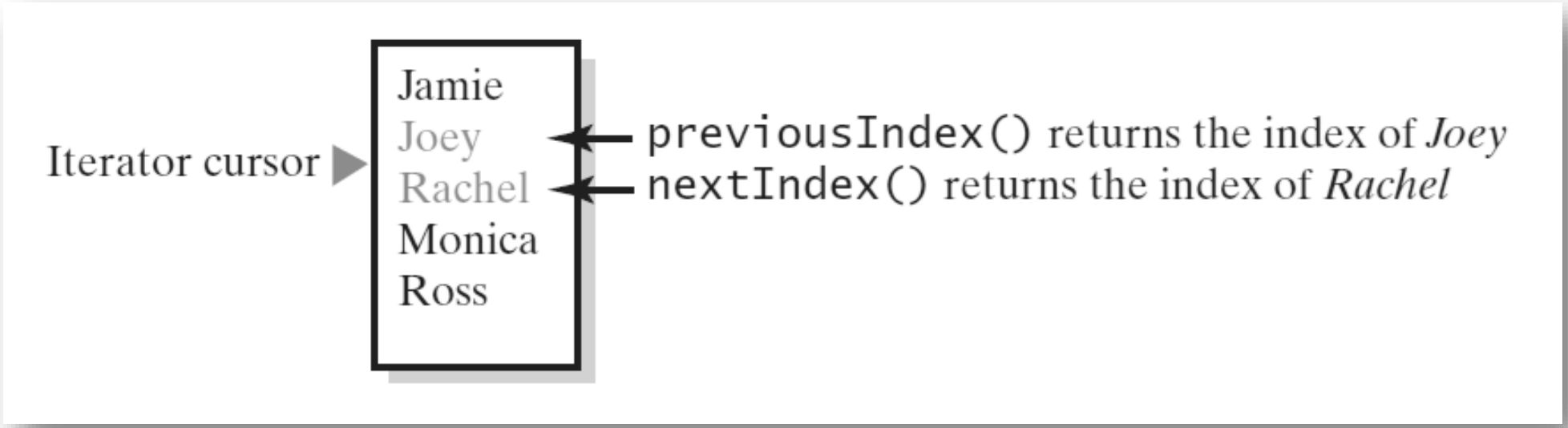
The Interface `ListIterator`

The effect of a call to **`previous`** on a list



The Interface `ListIterator`

The indices returned by the methods **`nextIndex`** and **`previousIndex`**



The Interface `List` Revisited

- Method **`set`** replaces entry that either **`next`** or **`previous`** just returned.
- Method **`add`** inserts an entry into list just before iterator's current position
- Method **`remove`** removes list entry that last call to either **`next`** or **`previous`** returned