University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides adapted from slides of Textbook, Dr. Ramirez's CS 0445, and Dr. Farnan's CS 1501)

Announcements

- Upcoming Deadlines:
  - Homework 9 and Lab 9
    - reopened until this Friday 12/2 @ 11:59 pm
  - Homework 10 (to be posted soon) and Lab 11
    - next Monday 12/5 @ 11:59 pm
  - Assignment 3 and 4: both due on Friday 12/9 @ 11:59 pm
    - very small amount of work!

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

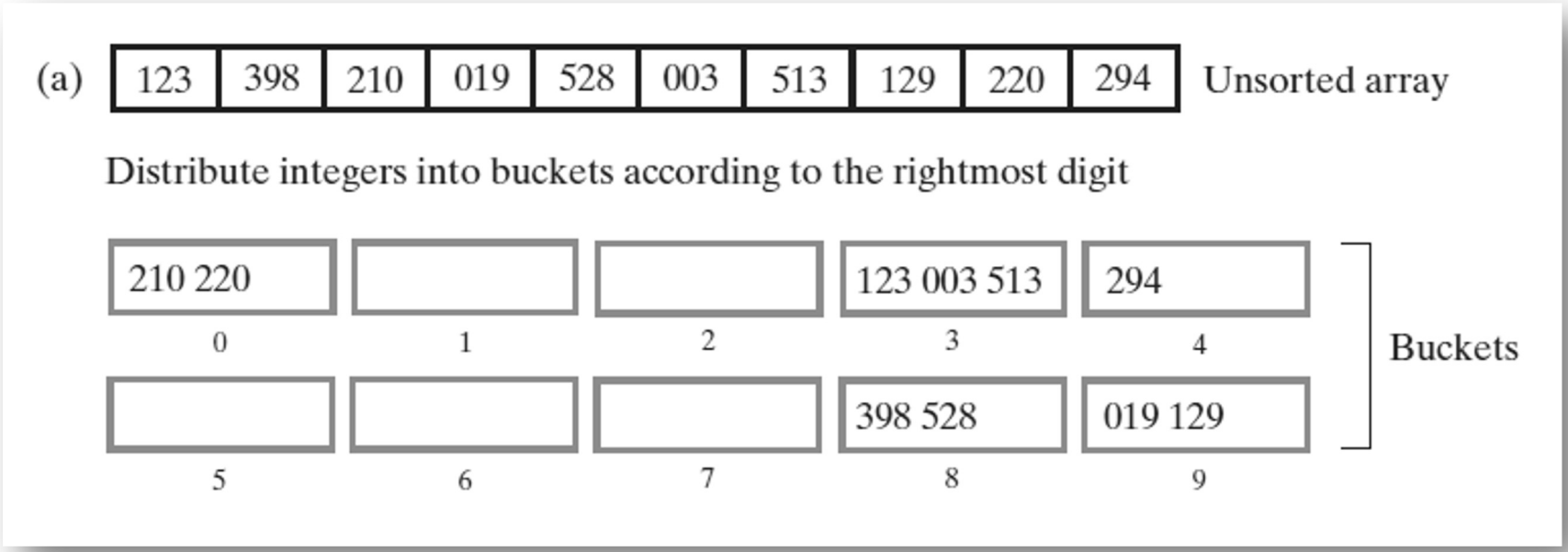
- **Q: Could you explain why the runtime of insertionsort is O(n^2)?**
- insertion Sort has two nested loops:
  - the outer loop has n-1 iterations
  - First iteration → 1 comparison in the worst case
  - Second iteration → 2 comparisons
  - Third iteration → 3 comparisons
  - …
  - iteration n-1 → n-1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

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.

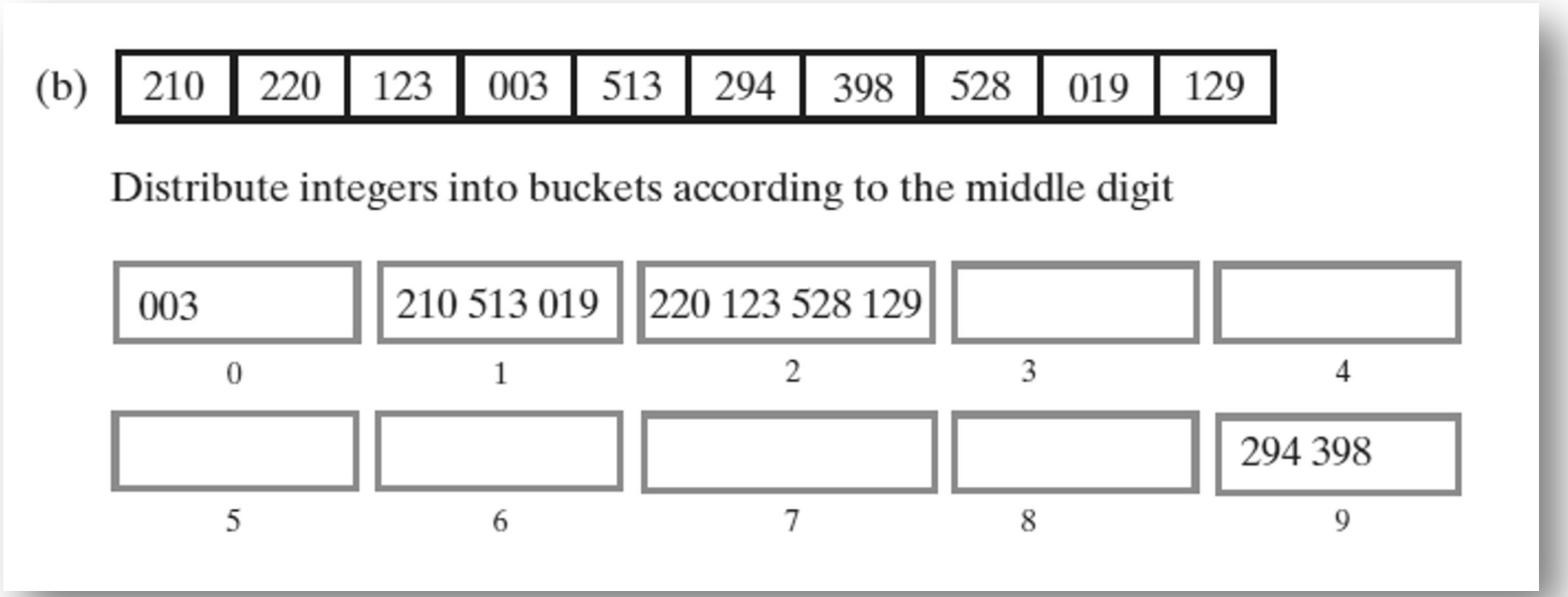
Radix Sort

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



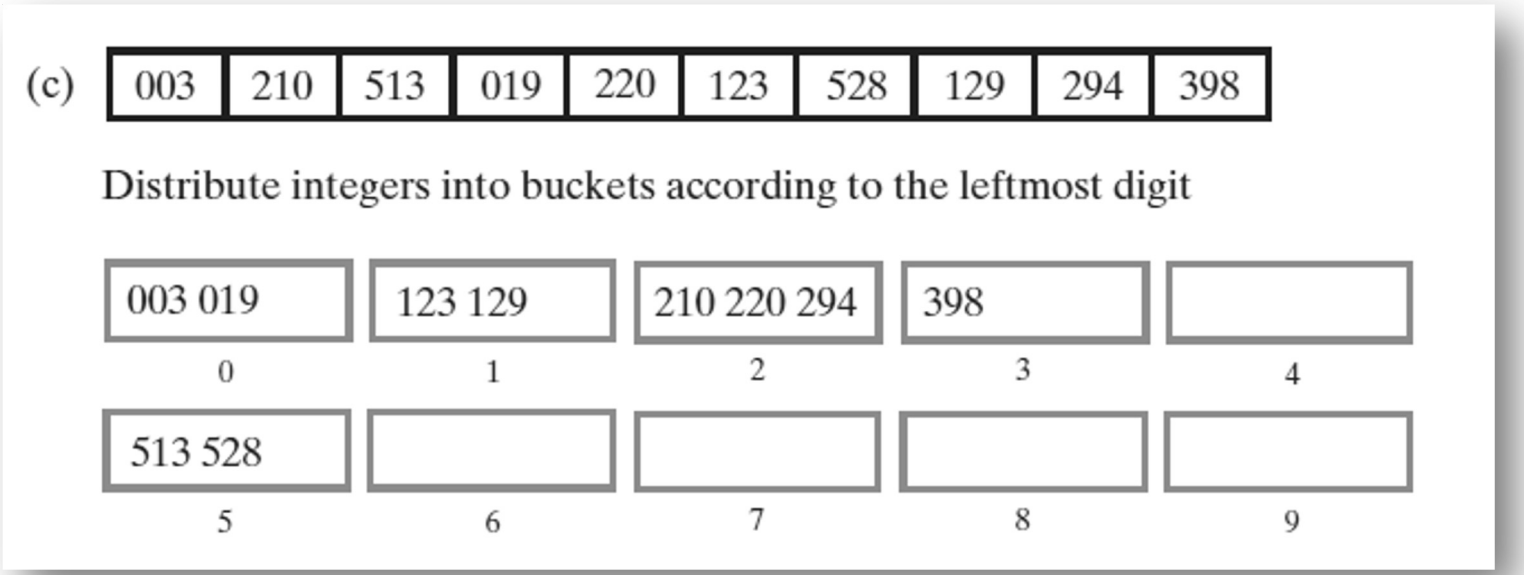
Radix Sort

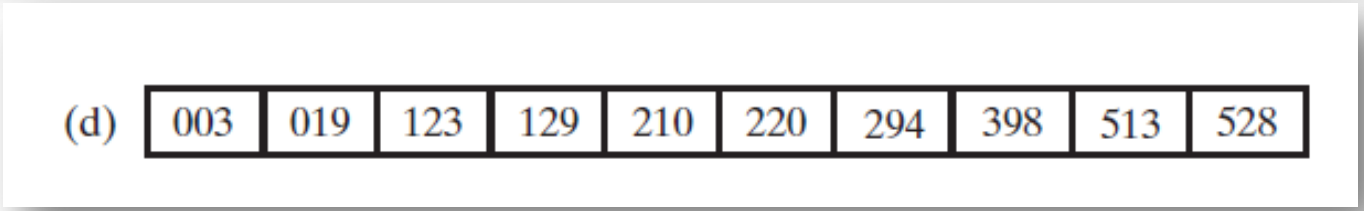
- Radix sort: (b) reordered array and buckets after second distribution;



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
- implemented as a distinct, usually nested, 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
```

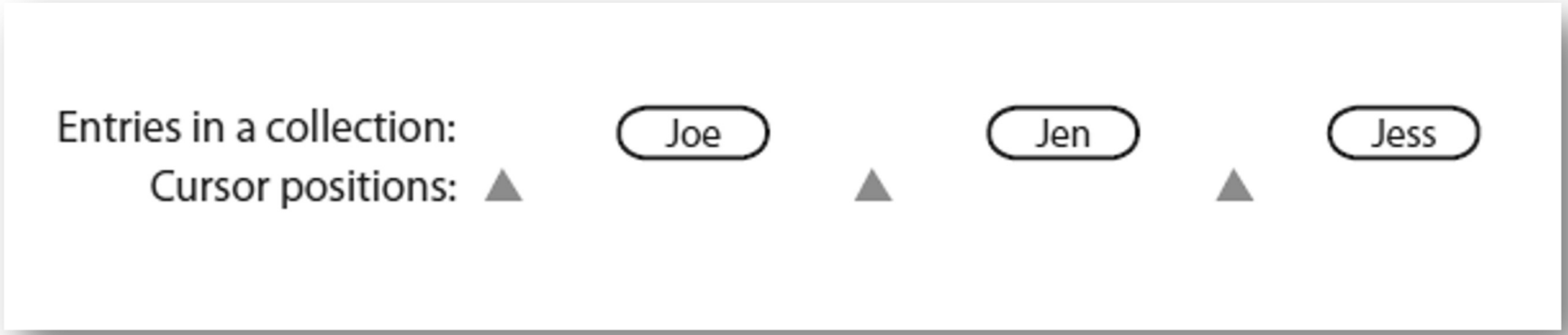
The Interface `Iterator`

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
```

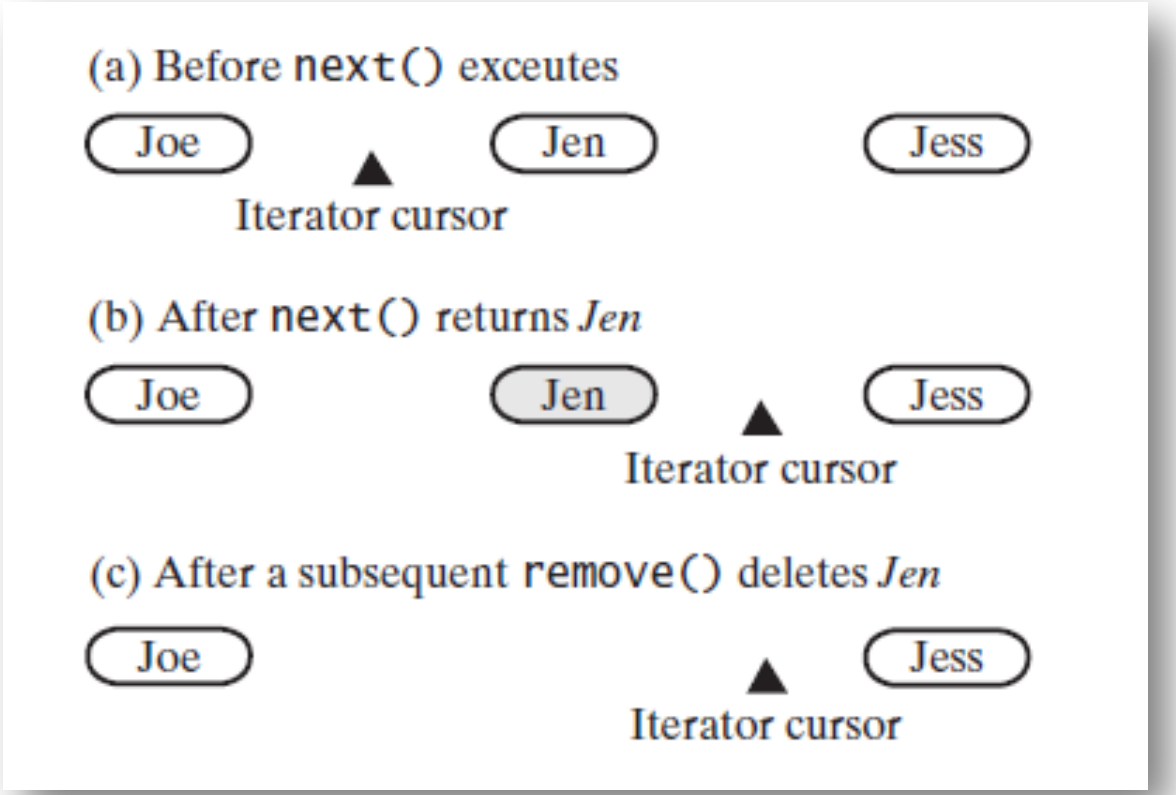
The Interface `Iterator`

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`**



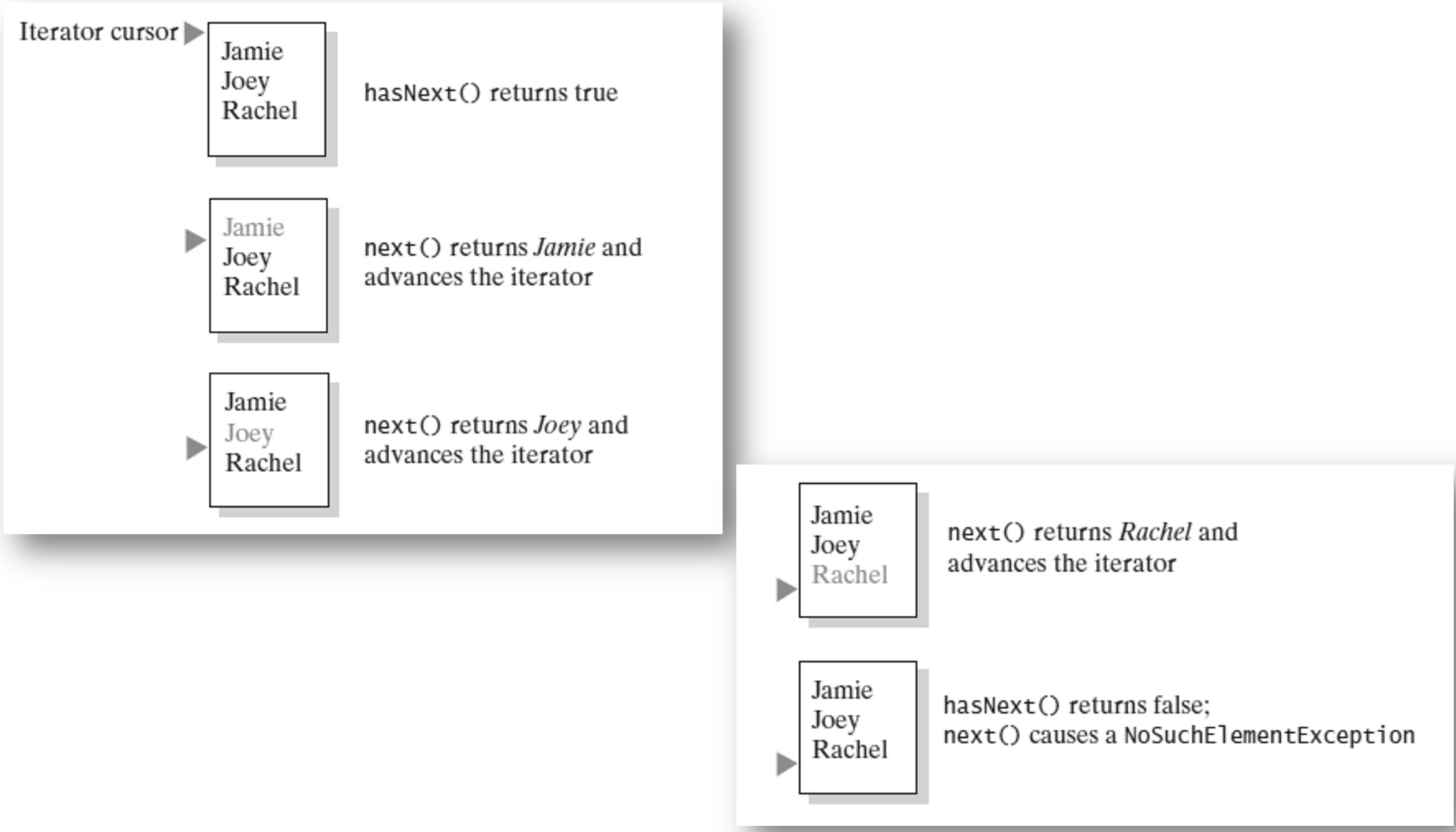
The Interface `Iterable`

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
```

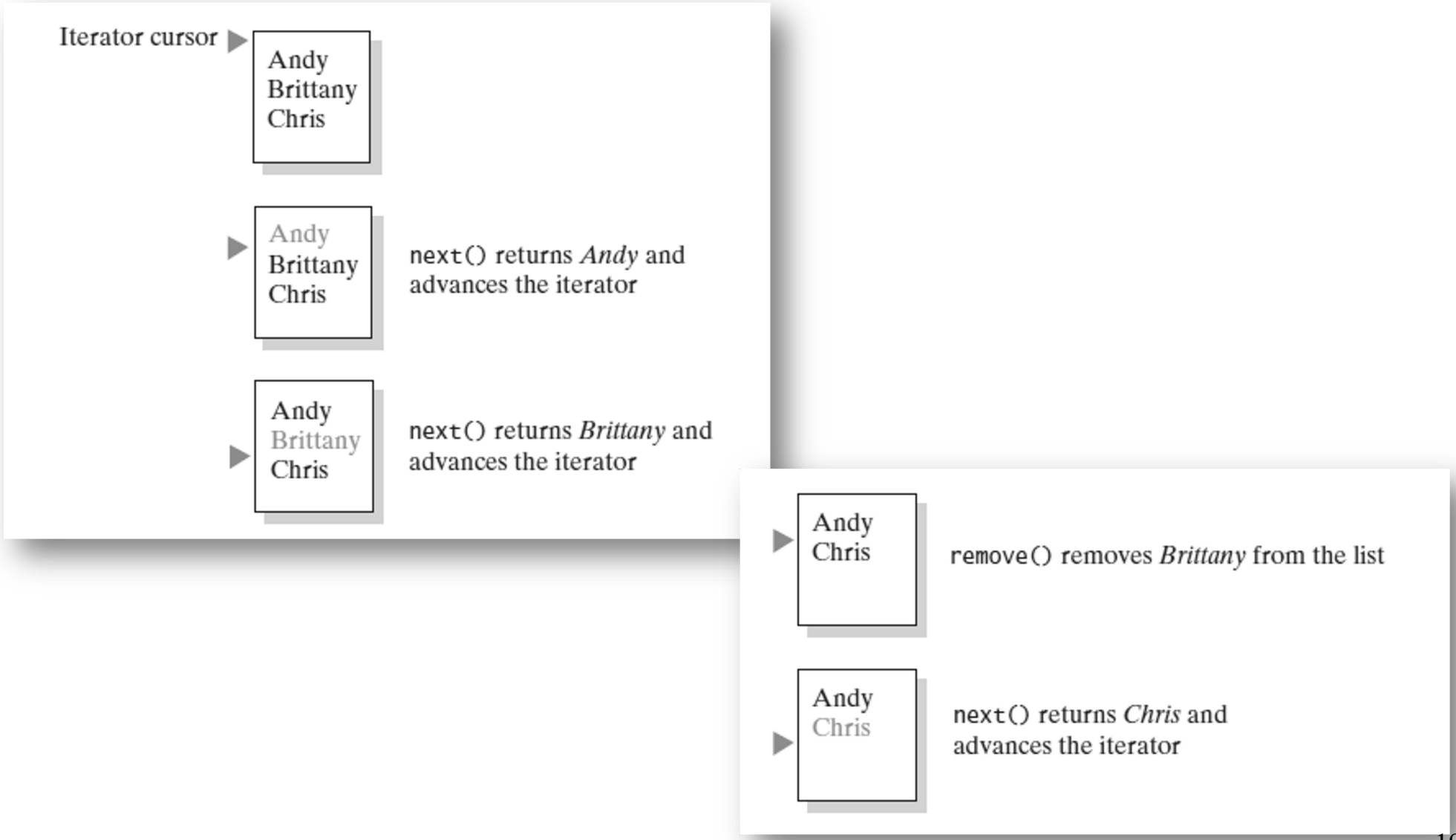
Using the Interface `Iterator`

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

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
```

Wouldn't it be wonderful if...

- Search through a collection could be accomplished in Θ(1) with relatively small memory needs?
- Lets try this:
  - Assume we have an array of length m (call it HT)
  - Assume we have a function h(x) that maps from our key space to {0, 1, 2, …, m-1}
    - E.g., $\mathbb{Z} \rightarrow \{0, 1, 2, \ldots, m-1\}$ for integer keys
    - Let's also assume h(x) is efficient to compute
- This is the basic premise of *hash tables*