
CS/COE 0445 - Data Structures

# Week 11: Searching and Iterators

Sherif Khattab

http://www.cs.pitt.edu/~skhattab/cs0445

# Searching

- Searching is an everyday occurrence

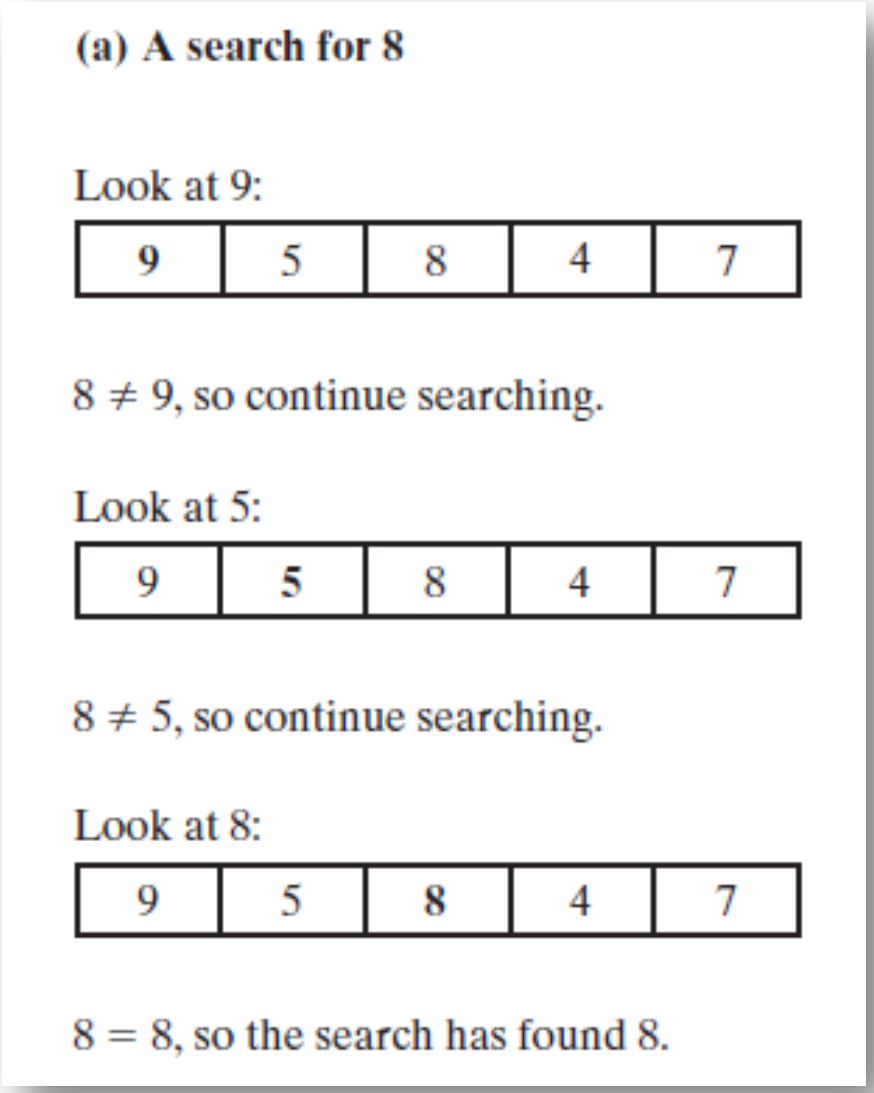
# Iterative Sequential Search of an Unsorted Array

- Using a loop to search for a specific valued entry.

```
public static <T> boolean inArray(T[] anArray, T anEntry)
{
   boolean found = false;
   int index = 0;
   while (!found && (index < anArray.length))
   {
      if (anEntry.equals(anArray[index]))
         found = true;
      index++;
   } // end while

   return found;
} // end inArray
```

# Iterative Sequential Search of an Unsorted Array

An iterative sequential search of an array that finds its target



**(a) A search for 8**

Look at 9:

| | | | | |
|---|---|---|---|---|
| **9** | 5 | 8 | 4 | 7 |

$8 \neq 9$, so continue searching.

Look at 5:

| | | | | |
|---|---|---|---|---|
| 9 | **5** | 8 | 4 | 7 |

$8 \neq 5$, so continue searching.

Look at 8:

| | | | | |
|---|---|---|---|---|
| 9 | 5 | **8** | 4 | 7 |

$8 = 8$, so the search has found 8.

# Iterative Sequential Search of an Unsorted Array

**(b) A search for 6**

Look at 9:

| | | | | |
|---|---|---|---|---|
| **9** | 5 | 8 | 4 | 7 |

$6 \neq 9$, so continue searching.

Look at 5:

| | | | | |
|---|---|---|---|---|
| 9 | **5** | 8 | 4 | 7 |

$6 \neq 5$, so continue searching.

Look at 8:

| | | | | |
|---|---|---|---|---|
| 9 | 5 | **8** | 4 | 7 |

$6 \neq 8$, so continue searching.

Look at 4:

| | | | | |
|---|---|---|---|---|
| 9 | 5 | 8 | **4** | 7 |

$6 \neq 4$, so continue searching.

Look at 7:

| | | | | |
|---|---|---|---|---|
| 9 | 5 | 8 | 4 | **7** |

$6 \neq 7$, so continue searching.

No entries are left to consider, so the search ends. 6 is not in the array.

An iterative sequential search of an array that does not find its target

# Recursive Sequential Search of an Unsorted Array

Pseudocode of the logic of our recursive algorithm.

```
Algorithm to search a[first] through a[last] for desiredItem
if (there are no elements to search)
   return false
else if (desiredItem equals a[first])
   return true
else
   return the result of searching a[first + 1] through a[last]
```

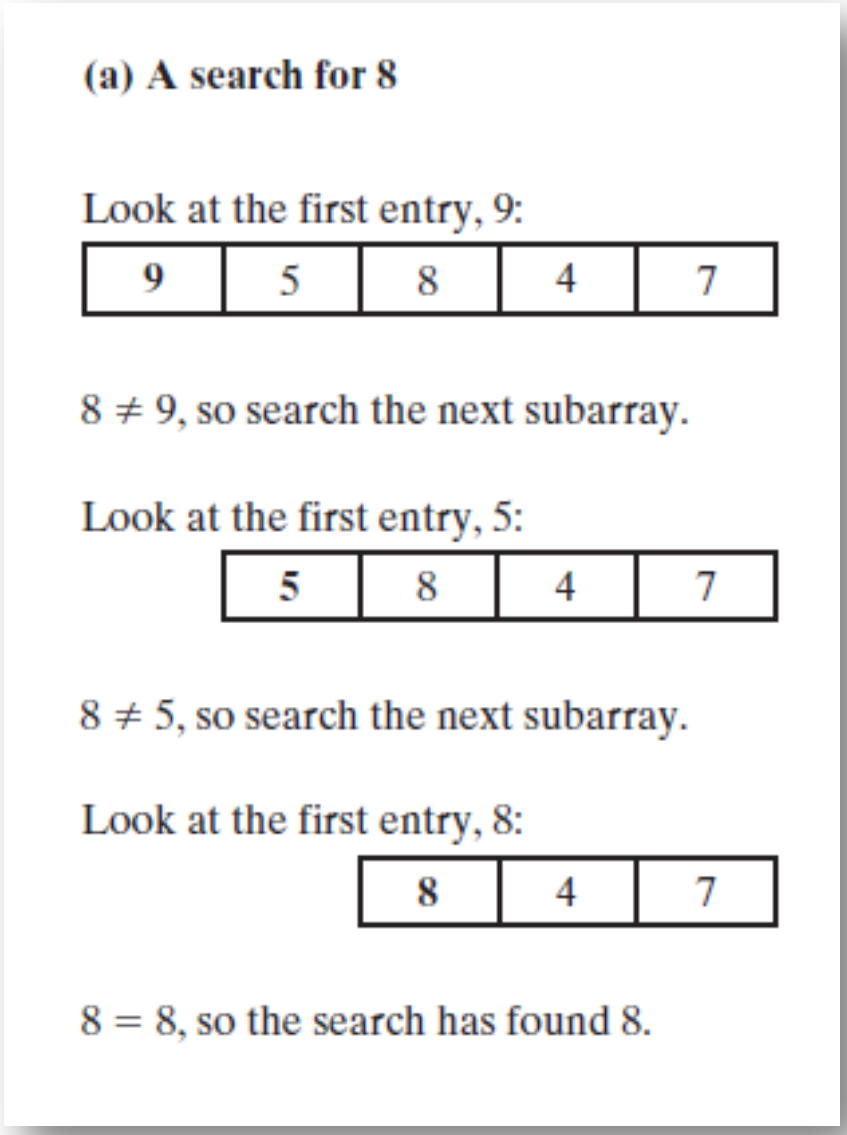
# Recursive Sequential Search of an Unsorted Array

- Method that implements this algorithm will need parameters **`first`** and **`last`**.

```
/** Searches an array for anEntry. */
public static <T> boolean inArray(T[] anArray, T anEntry)
{
   return search(anArray, 0, anArray.length - 1, anEntry);
} // end inArray

// Searches anArray[first] through anArray[last] for desiredItem.
// first >= 0 and < anArray.length.
// last >= 0 and < anArray.length.
private static <T> boolean search(T[] anArray, int first, int last,
T desiredItem)
{
   boolean found;

   if (first > last)
      found = false; // No elements to search
   else if (desiredItem.equals(anArray[first]))
      found = true;
   else
      found = search(anArray, first + 1, last, desiredItem);

   return found;
} // end search
```

# Recursive Sequential Search of an Unsorted Array

A recursive sequential search of an array that finds its target



**(a) A search for 8**

Look at the first entry, 9:

| **9** | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$8 \neq 9$, so search the next subarray.

Look at the first entry, 5:

| **5** | 8 | 4 | 7 |
|---|---|---|---|

$8 \neq 5$, so search the next subarray.

Look at the first entry, 8:

| **8** | 4 | 7 |
|---|---|---|

$8 = 8$, so the search has found 8.

# Recursive Sequential Search of an Unsorted Array

A recursive sequential search of an array that does not find its target

**(b) A search for 6**

Look at the first entry, 9:

| **9** | 5 | 8 | 4 | 7 |
|---|---|---|---|---|

$6 \neq 9$, so search the next subarray.

Look at the first entry, 5:

| **5** | 8 | 4 | 7 |
|---|---|---|---|

$6 \neq 5$, so search the next subarray.

Look at the first entry, 8:

# Recursive Sequential Search of an Unsorted Array

A recursive sequential search of an array that does not find its target

$6 \neq 5$, so search the next subarray.

Look at the first entry, 8:

| **8** | 4 | 7 |
|---|---|---|

$6 \neq 8$, so search the next subarray.

Look at the first entry, 4:

| **4** | 7 |
|---|---|

$6 \neq 4$, so search the next subarray.

Look at the first entry, 7:

| **7** |
|---|

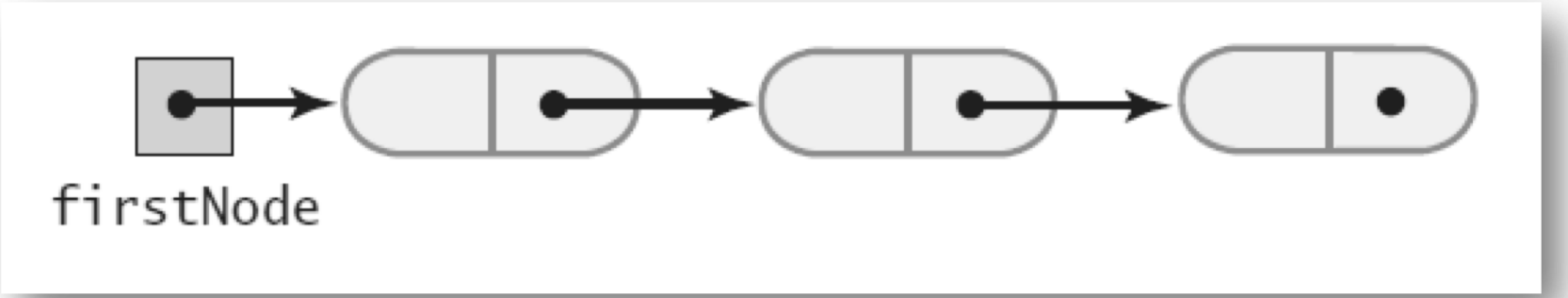
# Efficiency of a Sequential Search of an Array

The time efficiency of a sequential search of an array.

- Best case O(1)
- Worst case: O(*n*)
- Average case: O(*n*)

# Iterative Sequential Search of an Unsorted Chain

A chain of linked nodes that contain the entries in a list



# Iterative Sequential Search of an Unsorted Chain

## Implementation of iterative search

```
public boolean contains(T anEntry)
{
   boolean found = false;
   Node currentNode = firstNode;

   while (!found && (currentNode != null))
   {
      if (anEntry.equals(currentNode.getData()))
         found = true;
      else
         currentNode = currentNode.getNextNode();
   } // end while

   return found;
} // end contains
```

# Recursive Sequential Search of an Unsorted Chain

## Implementation of the method `search`

```
// Recursively searches a chain of nodes for desiredItem,
// beginning with the node that currentNode references.
private boolean search(Node currentNode, T desiredItem)
{
   boolean found;

   if (currentNode == null)
      found = false;
   else if (desiredItem.equals(currentNode.getData()))
      found = true;
   else
      found = search(currentNode.getNextNode(), desiredItem);

   return found;
} // end search
```

# Efficiency of a Sequential Search of a Chain

The time efficiency of a sequential search of a chain of linked nodes

- Best case: O(1)
- Worst case: O(*n*)
- Average case: O(*n*)

# Average-case Analysis of Sequential Search

- To do this we need to make an assumption about the index where the target exists
  - Let's assume that all index values are equally likely
    - If this is not the case, we can still do the analysis, if we know the actual probability distribution for the index
  - Our assumption means that, given N choices for an index, the probability of stopping at a given index, i, (which we will call P(i)) is
    - 1/N for any i
  - Let's define our key operation to be "looking at" an entry in the list
    - So for a given index i, we will require i operations
    - Let's call this value Ops(i)

# Average-case Analysis of Sequential Search

- Now we can define the average number of operations to be:

  Avg Ops = Sum_over_i (Ops(i) * P(i))

  = Sum_over_i (i * 1/N)

  = 1/N * Sum_over_i (i)

  = 1/N * [N * (N+1)]/2

  = (N+1)/2

- This is for success case (target found):
  - overall: successful search probability * (N+1)/2 +

    failed search probability * N
- In an absolute sense, this is better than the worst case, but asymptotically it is the same (why?)
- So in this case the worst and average cases are the same

# Amortized Analysis

- Average over a sequence of operations
- **add(newEntry)**
  - Recall that this version of the method adds to the end of the list
  - Runtime for Resizable Array ?

    O(1): We can go directly to the last location and insert there

  - What about the time to resize?

    The answer above is a bit deceptive

    Some adds take significantly more time, since we have to first allocate a new array and copy all of the data into it – O(N) time

    So we have O(N) + O(1) → O(N) total

# Amortized Analysis

- So we have an operation that sometimes takes O(1) and sometimes takes O(N)
- How do we handle this issue?

- Amortized Time (see http://en.wikipedia.org/wiki/Amortized_analysis )
  - Average time required over a sequence of operations
  - Individual operations may vary in their run-time, but we can get a consistent time for the overall sequence
  - Let's stick with the add() method for resizable bag and consider 2 different options for resizing:
    1) Increase the array size by 1 each time we resize
    2) Double the array size each time we resize (which is the way the authors actually did it)

# Amortized Analysis

1) Increase the array size by 1 each time we resize
   - Note that with this approach, once we resize we will have to do it with every add
   - Thus rather than O(1) our add() is now O(N) all the time
   - Specifically, assume the initial array size is 1
     - On insert 1 we just add the item (1 assignment)
     - On insert 2 we allocate and assign 2 items
     - On insert 3 we allocate and assign 3 items
     - …
     - Overall for N add() ops look at the total number of assignments we have to make:

       **1 + 2 + 3 + … + N =** $N(N+1)/2 \rightarrow O(N^2)$

# Amortized Analysis

## 2) Double the array size each time we resize

| Insert # | # of assignments | End array size |
|---|---|---|
| 1 | 1 | 1 |
| 2 | $2 = 1 + 2^0$ | 2 |
| 3 | $3 = 1 + 2^1$ | 4 |
| 4 | 1 | 4 |
| 5 | $5 = 1 + 2^2$ | 8 |
| … | 1 | 8 |
| 9 | $9 = 1 + 2^3$ | 16 |
| … | 1 | 16 |
| 17 | $17 = 1 + 2^4$ | 32 |
| … | 1 | 32 |
| 32 | 1 | 32 |

# Amortized Analysis

- Note that every row has 1 assignment (blue)
- Rows that are $2^K + 1$ for some K have an additional $2^K$ assignments (red) to copy data
- So for N **adds**, we have a total of
  - N assignments for the actual add
  - $2^0 + 2^1 + \ldots + 2^x$ for the copying
  - We stop when $1+2^x <= N < 1+2^{x+1}$
  - What is x?

    $\text{floor}(\lg_2(N-1))$

  - This gives us the geometric series
  - $\sum_{i=0}^{lg_2(N-1)} 2^i = O\left(2^{lg_2(N-1)}\right) = O(N)$

# Amortized Analysis

- Total is N + (N-1) = 2N-1 → O(N)
- Since we did N add() operations overall, our amortized time is O(N)/N = O(1), a constant
- Recall that when increasing by 1 we had $O(N^2)$ overall for the sequence, which gives us O(N) in amortized time
  - Note how much better our performance is when we double the array size
- Ok, that one was a bit complicated
  - Had a good deal of math in it
  - But that is what algorithm analysis is all about
  - If you can do some math you can save yourself some programming!

# Sequential Search of a Sorted Array

- Coins sorted by their mint dates
- Note: sequential search can be
- more efficient if the data is sorted


# Binary Search of a Sorted Array

Ignoring one half of the data
when the data is sorted

# Binary Search of a Sorted Array

First draft of an algorithm for a binary search of an array

```
Algorithm to search a[0] through a[n - 1] for desiredItem
mid = approximate midpoint between 0 and n - 1
if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return the result of searching a[0] through a[mid - 1]
else if (desiredItem > a[mid])
   return the result of searching a[mid + 1] through a[n - 1]
```

# Binary Search of a Sorted Array

Use parameters and make recursive calls look more like Java

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = approximate midpoint between first and last
if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else if (desiredItem > a[mid])
   return binarySearch(a, mid + 1, last, desiredItem)
```

# Binary Search of a Sorted Array

Refine the logic a bit, get a more complete algorithm

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last) / 2 // Approximate midpoint
if (first > last)
   return false
else if (desiredItem equals a[mid])
   return true
else if (desiredItem < a[mid])
   return binarySearch(a, first, mid - 1, desiredItem)
else // desiredItem > a[mid]
   return binarySearch(a, mid + 1, last, desiredItem)
```

# Binary Search of a Sorted Array

A recursive binary search of a sorted array that finds its target

**(a) A search for 8**

Look at the middle entry, 10:

| 2 | 4 | 5 | 7 | 8 | **10** | 12 | 15 | 18 | 21 | 24 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

| 2 | 4 | **5** | 7 | 8 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

| **7** | 8 |
|---|---|
| 3 | 4 |

$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:

| **8** |
|---|
| 4 |

$8 = 8$, so the search ends. 8 is in the array.

# Binary Search of a Sorted Array

A recursive binary search of a sorted array that (b) does not find its target

**(b) A search for 16**

Look at the middle entry, 10:

| 2 | 4 | 5 | 7 | 8 | **10** | 12 | 15 | 18 | 21 | 24 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

16 > 10, so search the right half of the array.

Look at the middle entry, 18:

| 12 | 15 | **18** | 21 | 24 | 26 |
|---|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 10 | 11 |

16 < 18, so search the left half of the array.

Look at the middle entry, 12:

# Binary Search of a Sorted Array

A recursive binary search of a sorted array that does not find its target

Look at the middle entry, 12:

| 12 | 15 |
|---|---|
| 6 | 7 |

16 > 12, so search the right half of the array.

Look at the middle entry, 15:

| 15 |
|---|
| 7 |

16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

# Binary Search of a Sorted Array

## Implementation of the method `binarySearch`

```
private static <T extends Comparable<? super T>> boolean binarySearch(T[]
anArray, int first, int last, T desiredItem)
{
   boolean found;
   int mid = first + (last - first) / 2;

   if (first > last)
      found = false;
   else if (desiredItem.equals(anArray[mid]))
      found = true;
   else if (desiredItem.compareTo(anArray[mid]) < 0)
      found = binarySearch(anArray, first, mid - 1, desiredItem);
   else
      found = binarySearch(anArray, mid + 1, last, desiredItem);

   return found;
} // end binarySearch
```

# Time complexity of Binary Search

- To simplify calculations we'll cheat a bit:
  1) Assume that the array is cut exactly in half with each iteration
     - In reality it may vary by one element either way
  2) Assume that the initial size of the array, N, is an exact power of 2, or $2^K$ for some K
     - In reality it can be any value
     - However, it will not affect our results
- Ok, so we have the following:
  - Initially: $N_0 = 2^K$
  - At iteration 1, $N_1 = N_0/2 = 2^{K-1}$ (in terms of K)
  - ...
  - Last iteration is when N = 1 = $2^0$ (in terms of K)

# Time complexity of Binary Search

- We do one comparison (test) per iteration
- Thus we have a total of K+1 comparisons maximum
  - But $N = 2^K$
  - So $K = \lg_2 N$
  - Which makes $K + 1 = \lg_2 N + 1$
- This leads to our final answer of $O(\lg_2 N)$

# Java Class Library

static method **`binarySearch`** with the specification:

```
/** Searches an entire array for a given item.
    @param array        An array sorted in ascending order.
    @param desiredItem  The item to be found in the array.
    @return  Index of the array entry that equals desiredItem;
             otherwise returns -belongsAt - 1, where belongsAt is
             the index of the array element that should contain
             desiredItem. */
public static int binarySearch(type[] array, type desiredItem);
```

# Efficiency of a Binary Search of an Array

The time efficiency of a binary search of an array

- Best case: O(1)
- Worst case: O(log *n*)
- Average case: O(log *n*)

# Searching a Sorted Chain

- Similar to sequentially searching a sorted array.
- Implementation of **`contains`**.

```
public boolean contains(T anEntry)
{
   Node currentNode = firstNode;

   while ( (currentNode != null) &&
           (anEntry.compareTo(currentNode.getData()) > 0) )
   {
      currentNode = currentNode.getNextNode();
   } // end while

   return (currentNode != null) &&
          anEntry.equals(currentNode.getData());
} // end contains
```

# Binary Search of a Sorted Chain

- To find the middle of the chain you must traverse the whole chain
- Then must traverse one of the halves to find the middle of that half

- Hard to implement
- Less efficient than sequential search!

# Choosing between Sequential Search and Binary Search

The time efficiency of searching, expressed in Big Oh notation

| | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Sequential search (unsorted data) | O(1) | O($n$) | O($n$) |
| Sequential search (sorted data) | O(1) | O($n$) | O($n$) |
| Binary search (sorted array) | O(1) | O(log $n$) | O(log $n$) |

# Choosing between Iterative Search and Recursive Search

- Can save some time and space by using iterative version of a search
- Using recursion will not require much additional space for the recursive calls
- Coding binary search recursively is somewhat easier

# 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
```

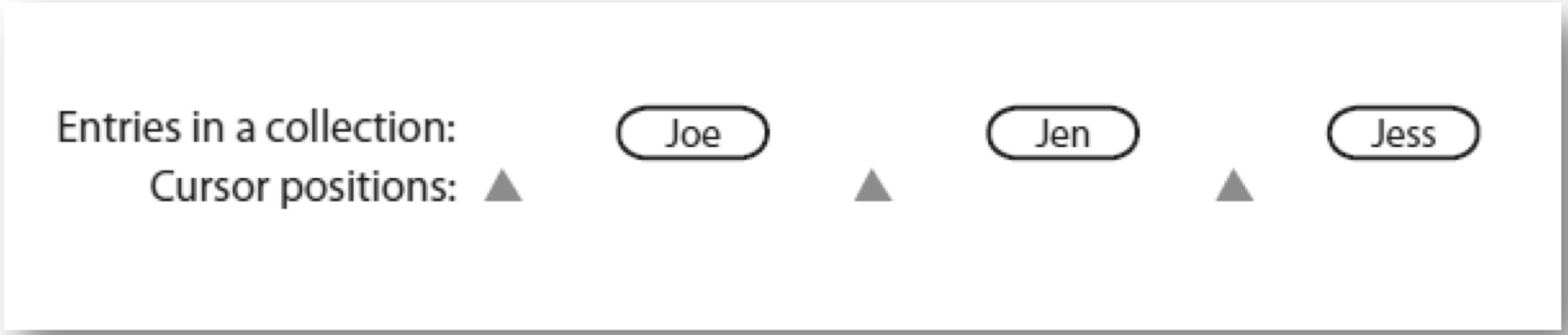
# The Interface `Iterator`

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
```

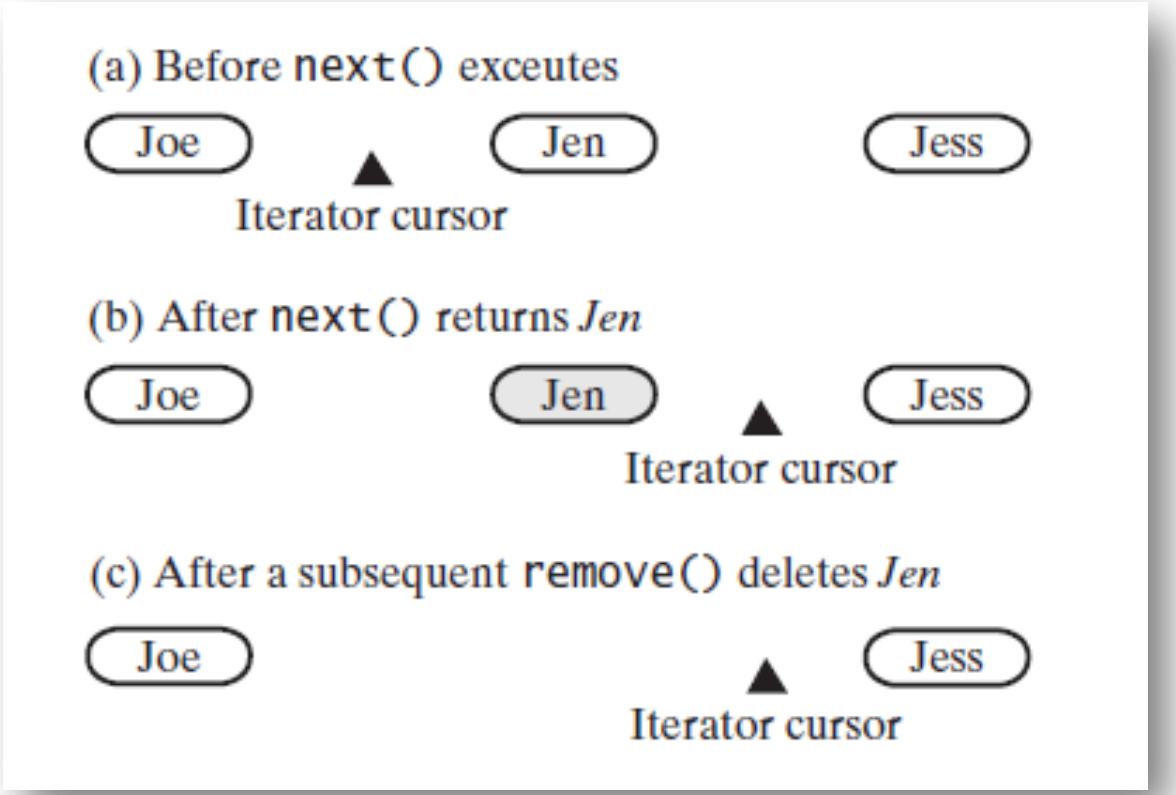
# The Interface `Iterator`

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`



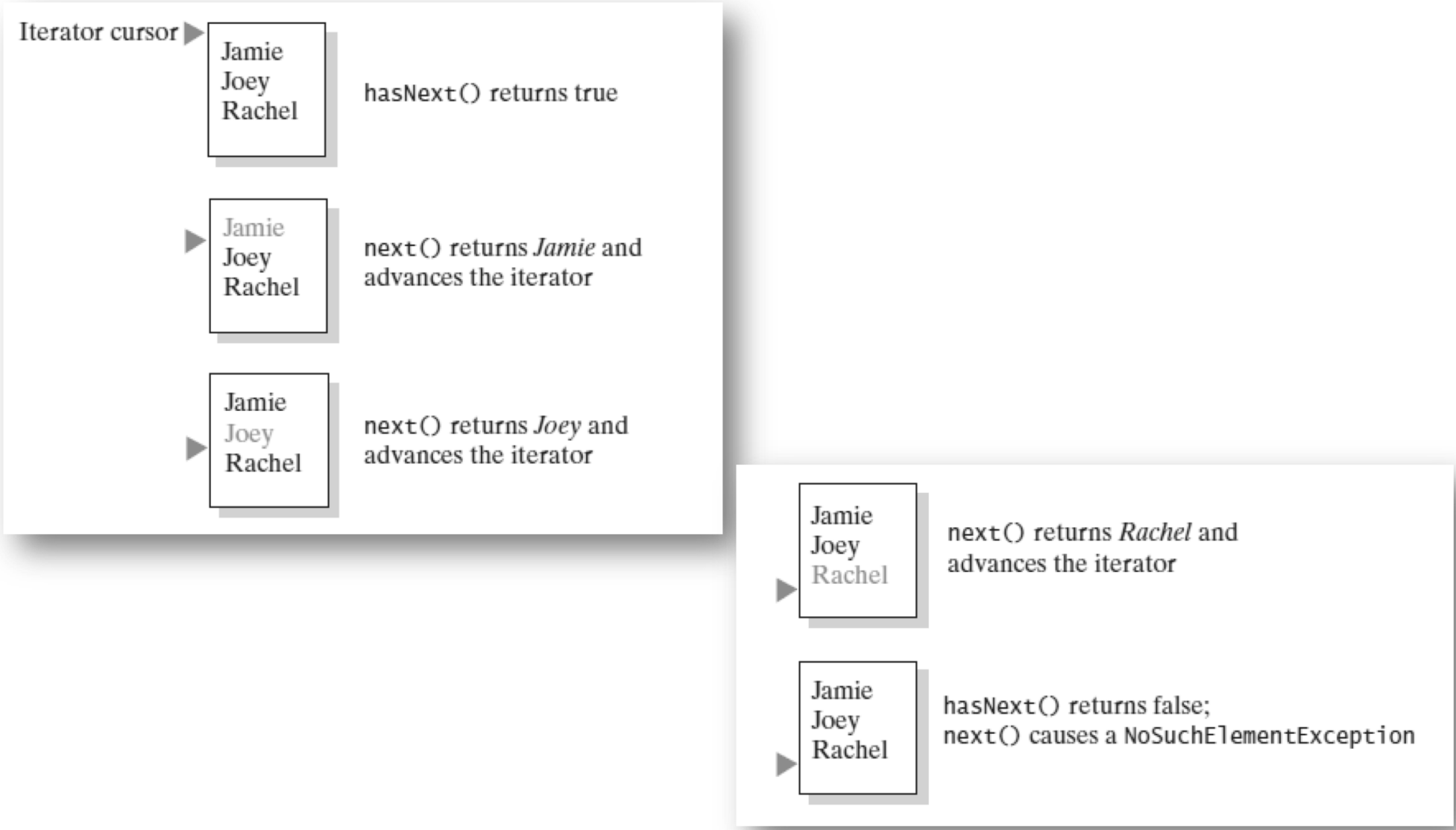
# The Interface `Iterable`

## 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
```

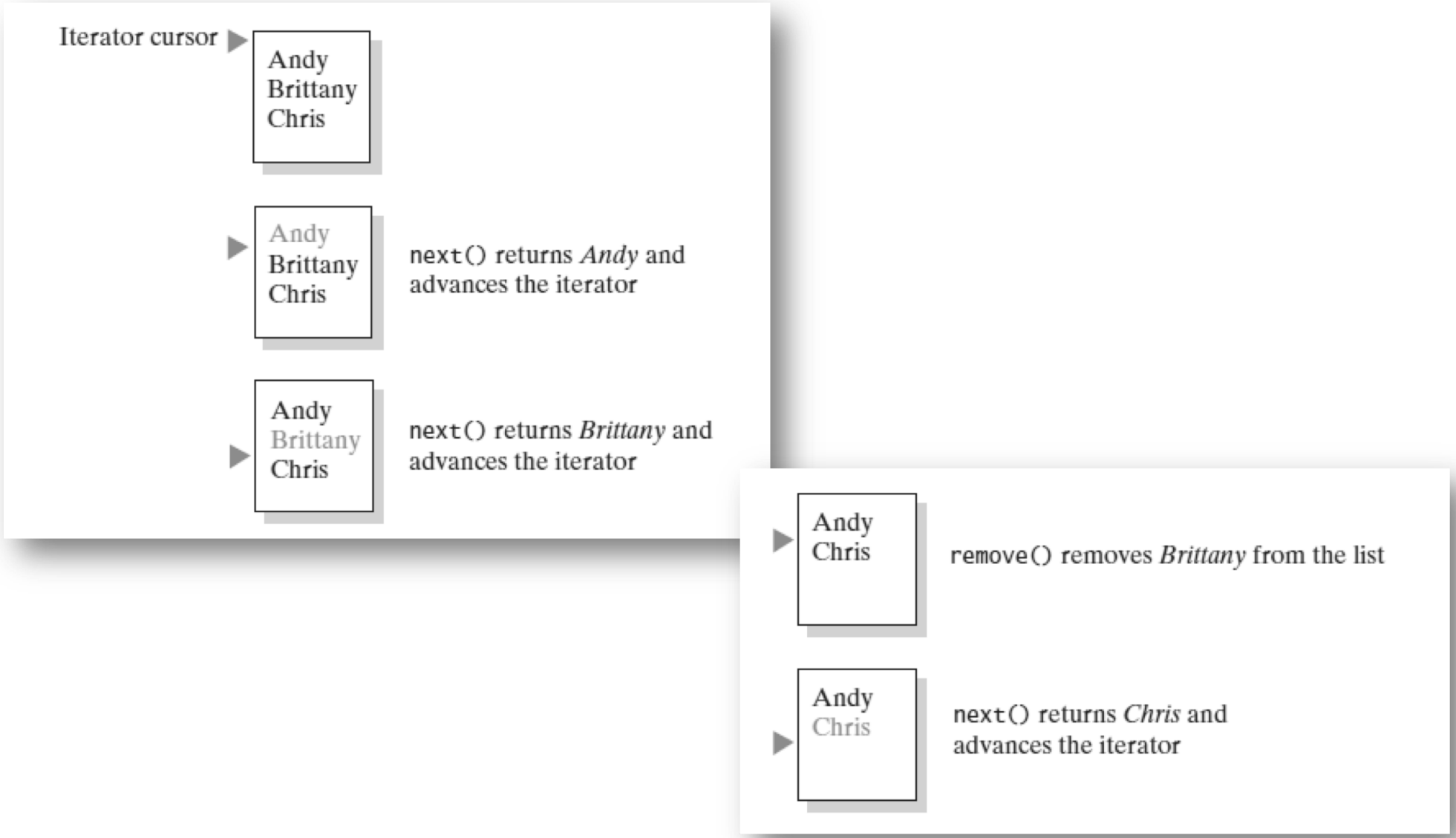
# Using the Interface `Iterator`

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



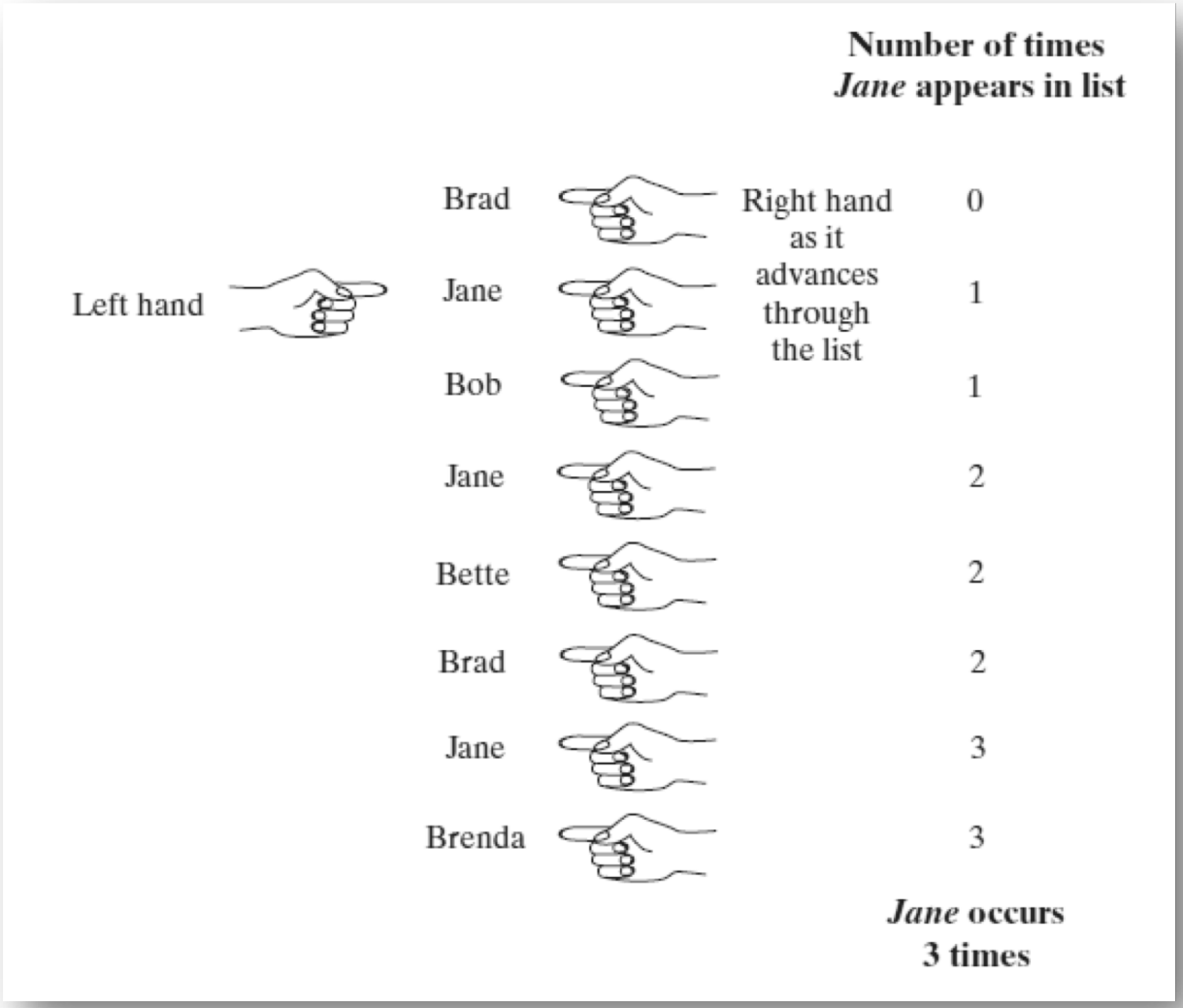
# Using the Interface `Iterator`

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
```

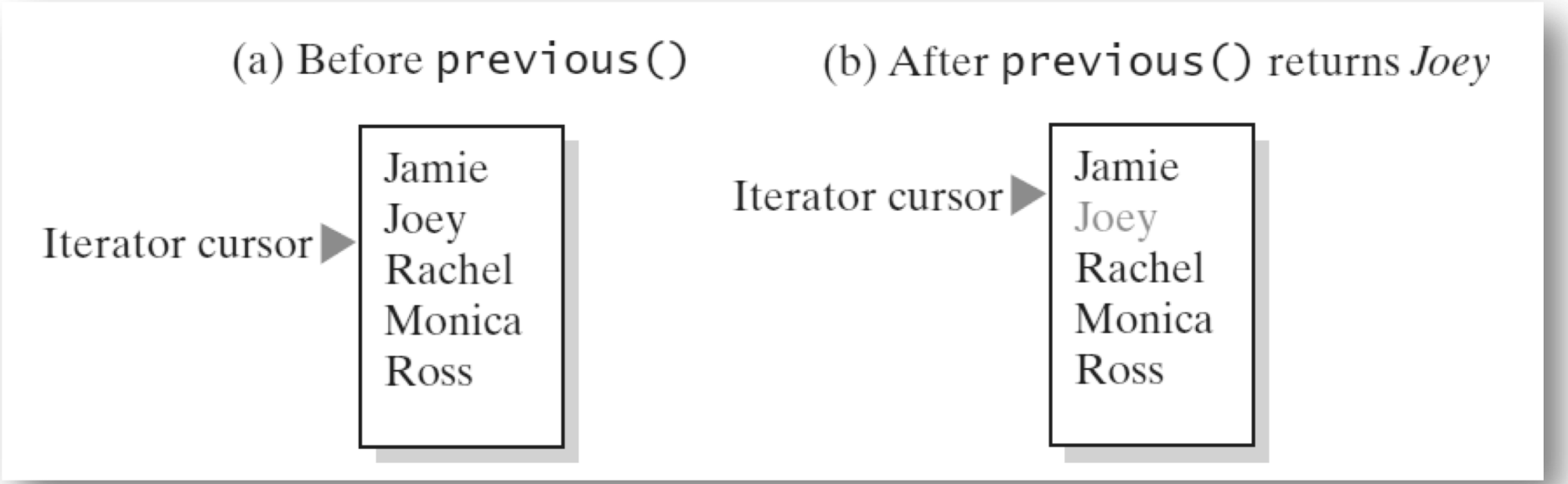
# The Interface `ListIterator`

Java's interface `java.util.ListIterator`

```
82
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
```

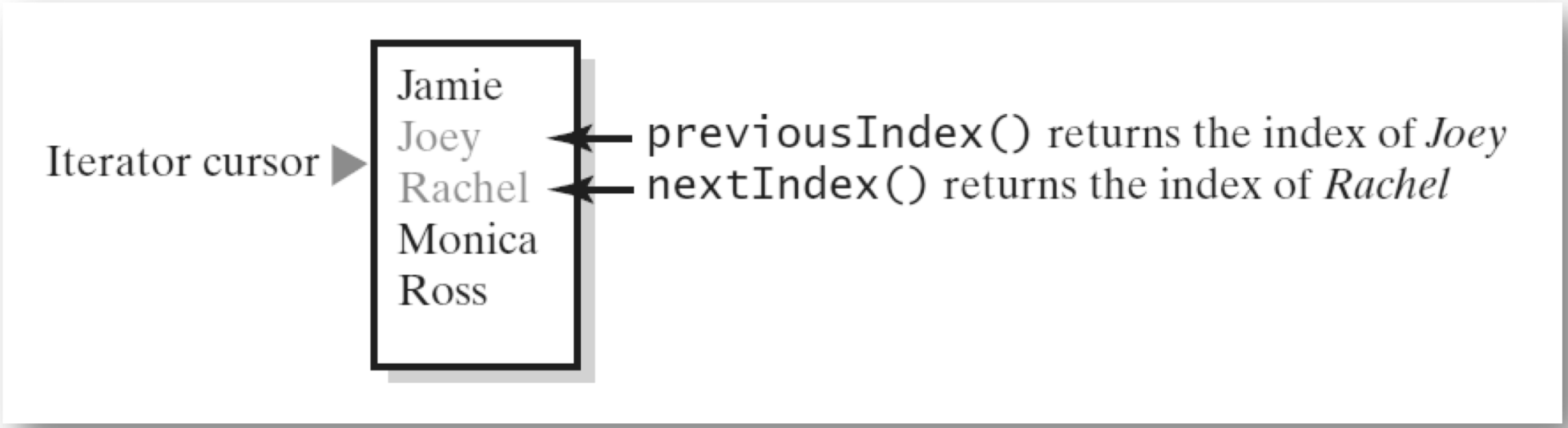
# The Interface `ListIterator`

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