Chapter 10.1-10.2
Simple Sorting

Chapter 6.1-6.3, 6.7
The List ADT
Reference-based Lists
(Linked Lists)

Chapter 10: Sorting and Searching Algorithms

10.1 – Sorting

10.2 – Simple Sorts

10.1 Sorting

- Sorting
  - Ordering an unsorted list of data elements
  - Very common and useful operation

- Efficiency:
  - Relating the number of comparisons to the number of elements in the list (N)

A Test Harness

- Conceptualize an application class called `Sorts`:
- The class could define an array `values` that can hold some number of integers and static methods:
  - `initValues`
    - Initializes the `values` array with random numbers
  - `isSorted`
    - Returns a boolean value indicating whether the `values` array is currently sorted
  - `swap`
    - `index1` and `index2` are parameters of the method
    - swaps the integers between `values[index1]` and `values[index2]`
  - `printValues`
    - Prints the contents of the `values` array to the `System.out` stream
    - (book example) the output is arranged evenly in ten columns

Example of `Sorts` `main` method

```
public static void main(String[] args) throws IOException
 {
   initValues();
   printValues();
   System.out.println("values is sorted: " + isSorted());
   System.out.println();

   swap(0, 1);     // normally we put sorting algorithm here

   printValues();
   System.out.println("values is sorted: " + isSorted());
   System.out.println();
 }
```

Output from Example

```
the values array is:
20 49 07 50 45 69 20 07 88 02
89 87 35 98 23 98 61 03 75 48
25 81 97 79 40 78 47 56 24 07
63 39 52 80 11 63 51 45 25 78
35 62 72 05 98 83 05 14 30 23

values is sorted: false

the values array is:
49 20 07 50 45 69 20 07 88 02
89 87 35 98 23 98 61 03 75 48
25 81 97 79 40 78 47 56 24 07
63 39 52 80 11 63 51 45 25 78
35 62 72 05 98 83 05 14 30 23

values is sorted: false
```

This part varies for each sample run

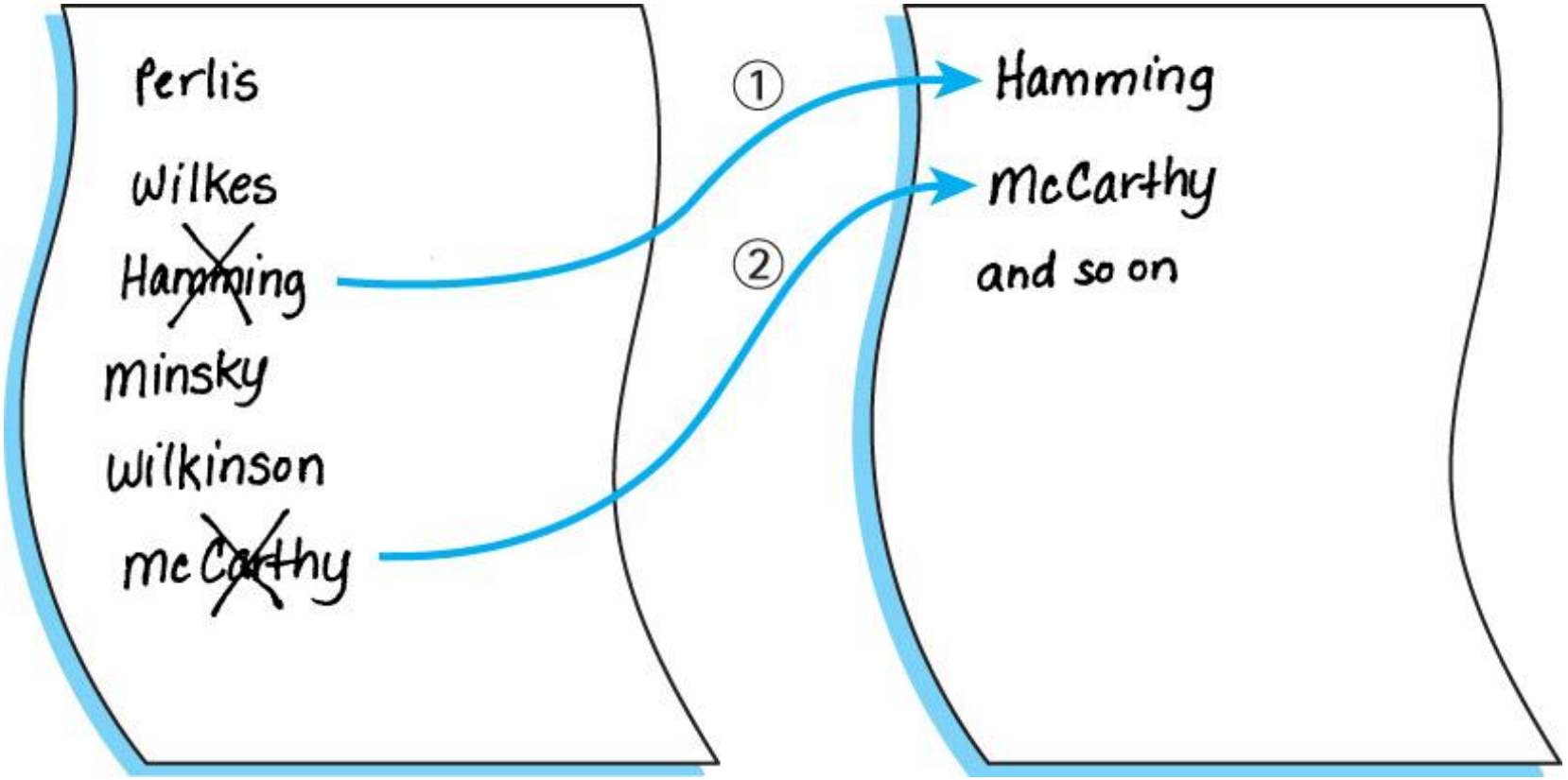
This does to, of course

10.2 Simple Sorts

- In this section we present three “simple” sorts
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Properties of these sorts
  - use an unsophisticated brute force approach
  - are not very efficient
  - are easy to understand and to implement

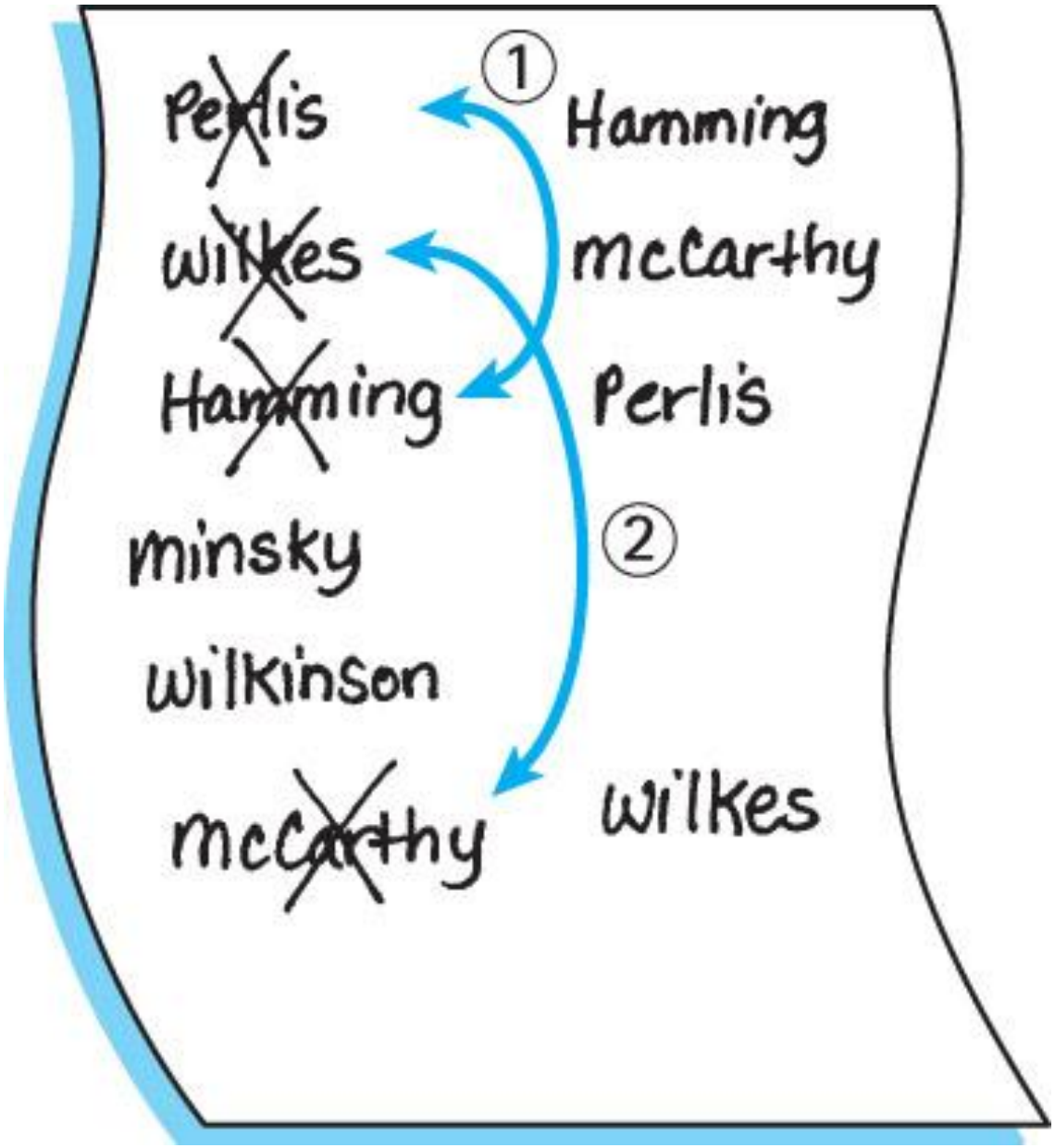
Selection Sort

- If we were handed a list of names on a sheet of paper and asked to put them in alphabetical order, we might use this general approach:
  - *Select* the name that comes first in alphabetical order, and write it on a second sheet of paper.
  - Cross the name out on the original sheet.
  - Repeat steps 1 and 2 for the second name, the third name, and so on until all the names on the original sheet have been crossed out and written onto the second sheet, at which point the list on the second sheet is sorted.


Perlis
Wilkes
Hamming
Minsky
Wilkinson
McCarthy
①
②
Hamming
McCarthy
and so on

An improvement

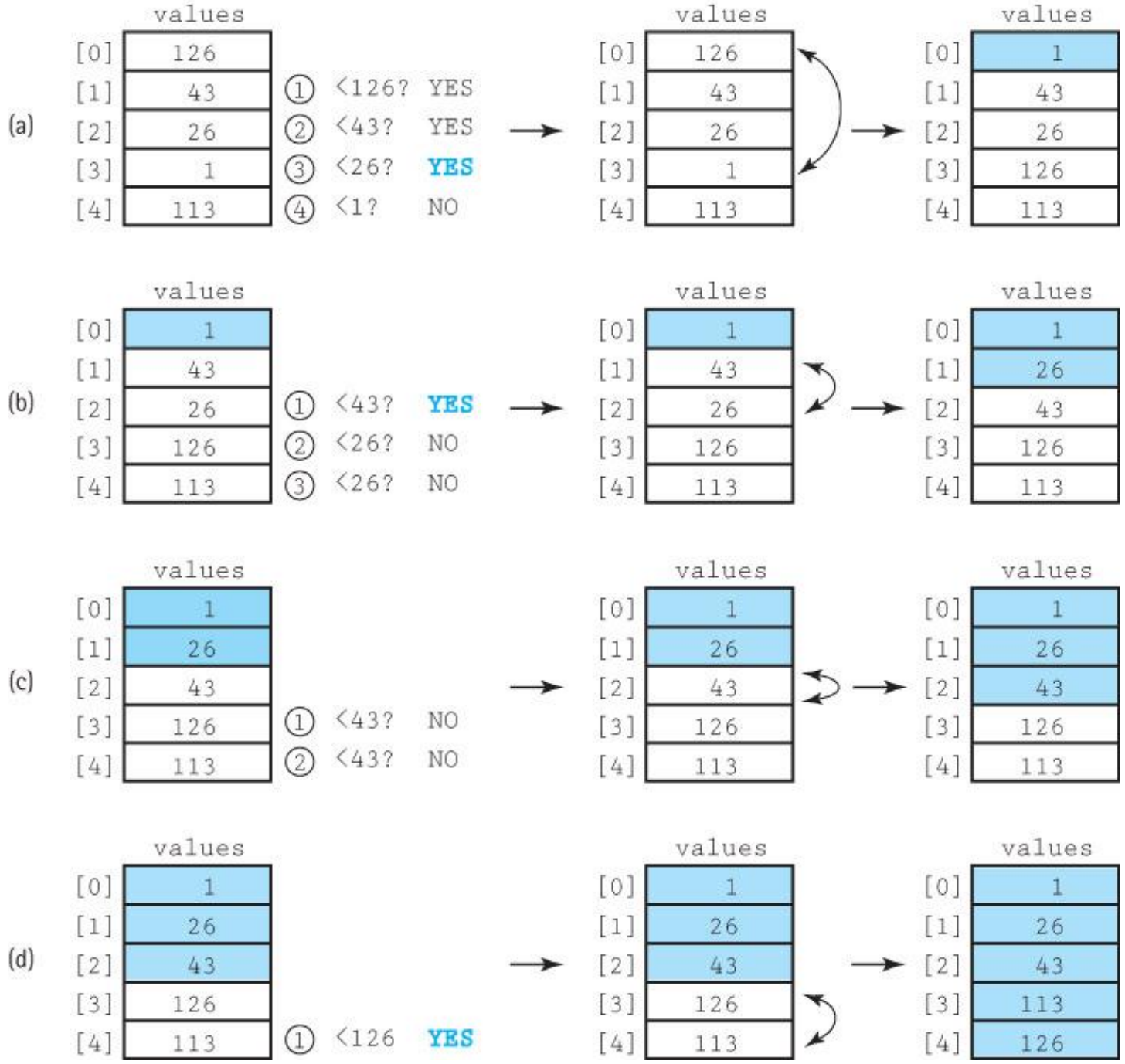
- Simple algorithm with one drawback
  - It requires space to store two complete lists
    - Like 2 arrays or 2 linked lists!!!
- How can this be improved?
- Instead of writing the “first” name onto a separate sheet of paper
  - exchange it with the name in the first location on the original sheet. And so on.


Perlis
Hamming
Wilkes
McCarthy
Hamming
Perlis
1
Minsky
2
Wilkinson
McCarthy
Wilkes

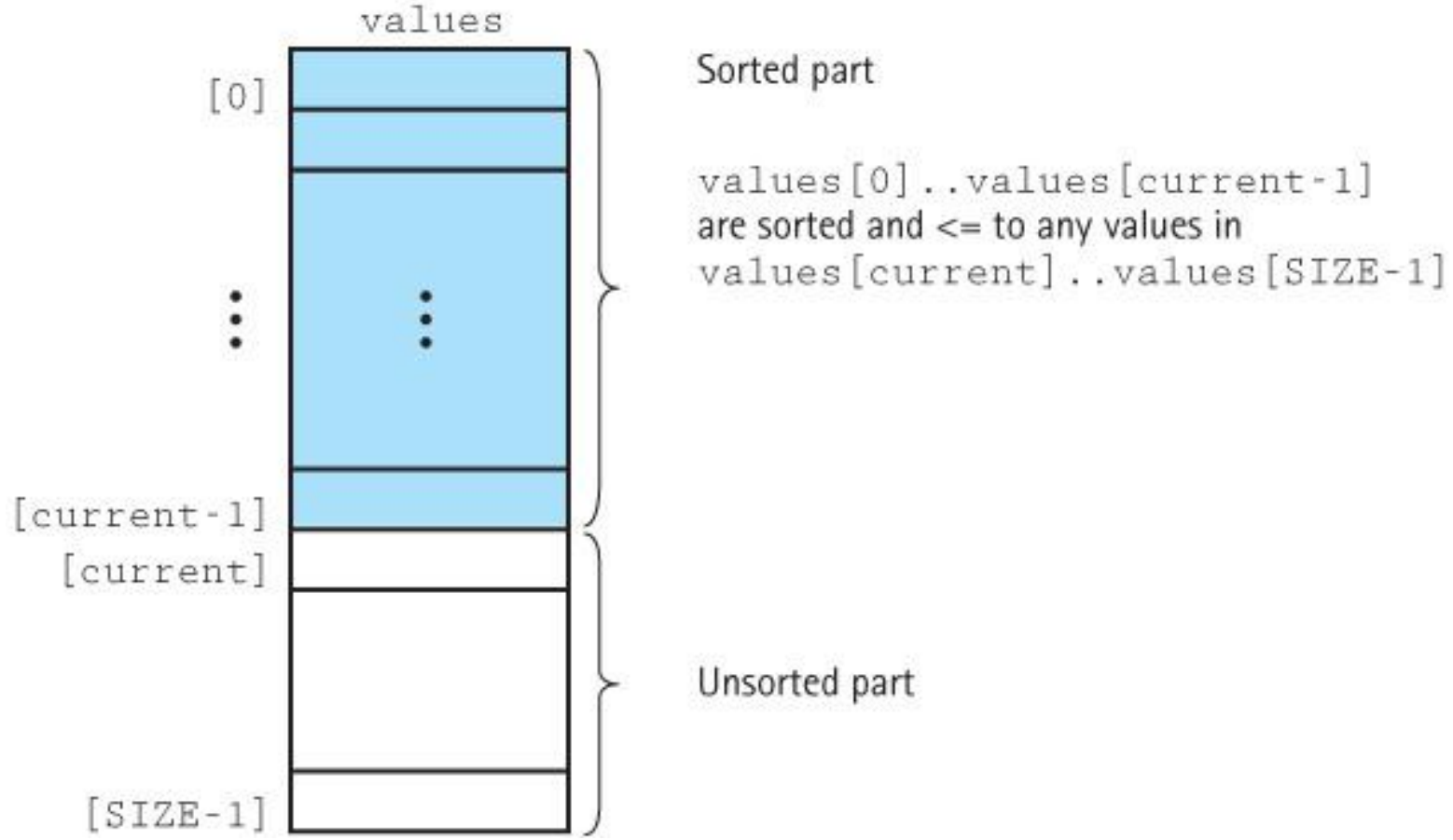
Selection Sort Algorithm

SelectionSort
for current going from 0 to SIZE - 2
    Find the index in the array of the smallest unsorted element
    Swap the current element with the smallest unsorted one

An example is depicted on the following slide …


(a)
values
[0] 126
[1] 43
[2] 26
[3] 1
[4] 113
① <126? YES
② <43? YES
③ <26? YES
④ <1? NO
values
[0] 126
[1] 43
[2] 26
[3] 1
[4] 113
values
[0] 1
[1] 43
[2] 26
[3] 126
[4] 113
(b)
values
[0] 1
[1] 43
[2] 26
[3] 126
[4] 113
① <43? YES
② <26? NO
③ <26? NO
values
[0] 1
[1] 43
[2] 26
[3] 126
[4] 113
values
[0] 1
[1] 26
[2] 43
[3] 126
[4] 113
(c)
values
[0] 1
[1] 26
[2] 43
[3] 126
[4] 113
① <43? NO
② <43? NO
values
[0] 1
[1] 26
[2] 43
[3] 126
[4] 113
values
[0] 1
[1] 26
[2] 43
[3] 126
[4] 113
(d)
values
[0] 1
[1] 26
[2] 43
[3] 126
[4] 113
① <126 YES
values
[0] 1
[1] 26
[2] 43
[3] 126
[4] 113
values
[0] 1
[1] 26
[2] 43
[3] 113
[4] 126

Selection Sort Snapshot



Selection Sort Code

```
// Returns the index of the smallest value in
// values[startIndex]..values[endIndex].
static int minIndex(int startIndex, int endIndex)
{
  int indexOfMin = startIndex;
  for (int index = startIndex + 1; index <= endIndex; index++)
    if (values[index] < values[indexOfMin])
      indexOfMin = index;
  return indexOfMin;
}

// Sorts the values array using the selection sort algorithm.
static void selectionSort()
{
  int endIndex = SIZE - 1;
  for (int current = 0; current < endIndex; current++)
    swap(current, minIndex(current, endIndex));
}
```

Testing Selection Sort

The test harness:

```
initValues();
printValues();
System.out.println("values is sorted: "
                   + isSorted());
System.out.println();

selectionSort();

System.out.println("Selection Sort called\n");
printValues();
System.out.println("values is sorted: "
                   + isSorted());
System.out.println();
```

The resultant output:

```
the values array is:
92 66 38 17 21 78 10 43 69 19
17 96 29 19 77 24 47 01 97 91
13 33 84 93 49 85 09 54 13 06
21 21 93 49 67 42 25 29 05 74
96 82 26 25 11 74 03 76 29 10

values is sorted: false

Selection Sort called
the values array is:
01 03 05 06 09 10 10 11 13 13
17 17 19 19 21 21 21 24 25 25
26 29 29 29 33 38 42 43 47 49
49 54 66 67 69 74 74 76 77 78
82 84 85 91 92 93 93 96 96 97

values is sorted: true
```

Selection Sort Analysis

- We describe the number of comparisons as a function of the number of elements in the array, i.e., `SIZE`. To be concise, in this discussion we refer to SIZE as *N*
- The `minIndex` method is called *N* - 1 times
- Within `minIndex`, the number of comparisons varies:
  - in the first call there are *N* - 1 comparisons
  - in the next call there are *N* - 2 comparisons
  - and so on, until in the last call, when there is only 1 comparison
- The total number of comparisons is

  $(N - 1) + (N - 2) + (N - 3) + \ldots + 1$

  $= N(N - 1)/2 = 1/2N^2 - 1/2N$
- The Selection Sort algorithm is $O(N^2)$

Number of Comparisons Required to Sort Arrays of Different Sizes Using Selection Sort

Number of Elements	Number of Comparisons
10	45
20	190
100	4,950
1,000	499,500
10,000	49,995,000

Bubble Sort

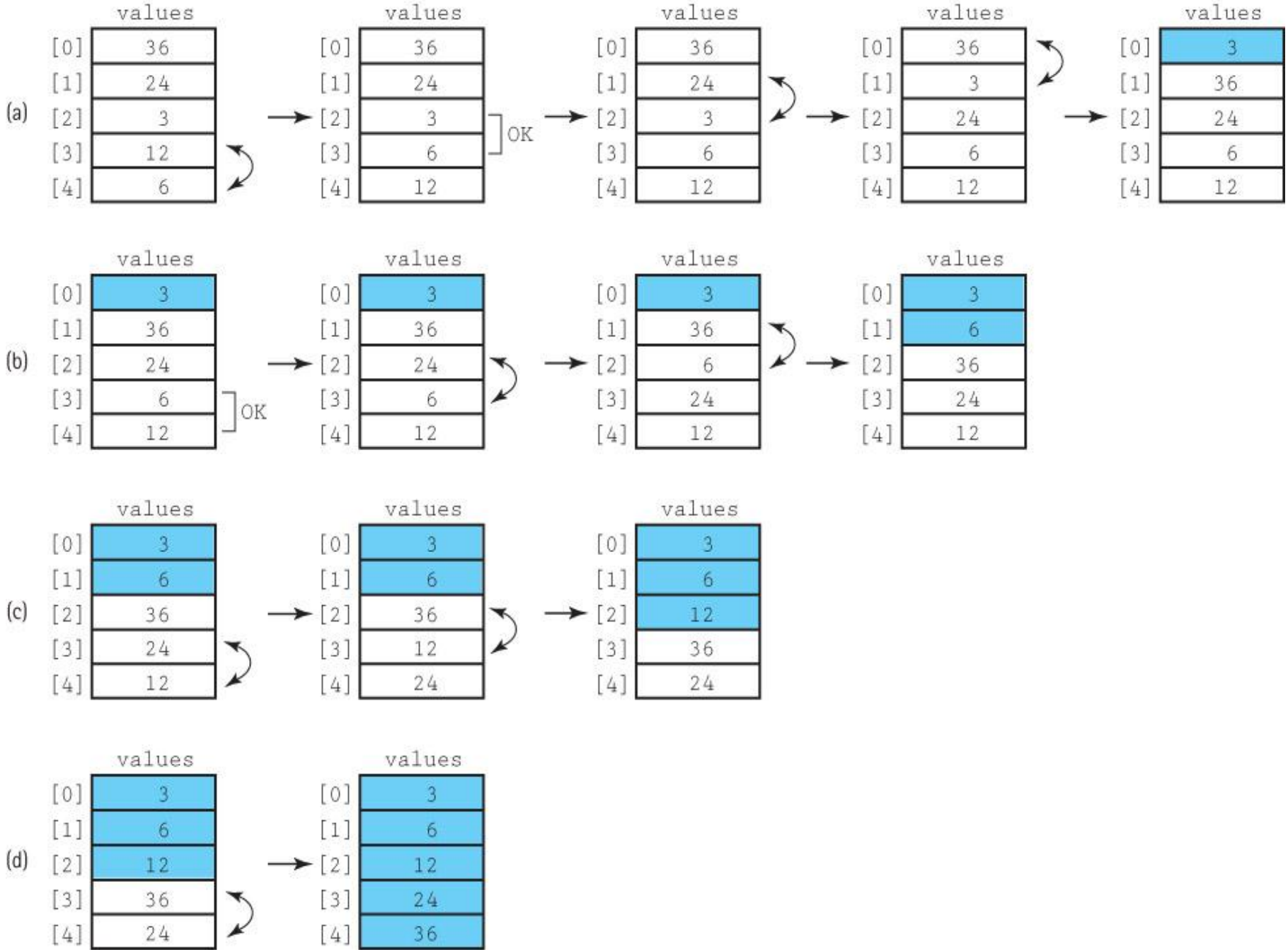
- With this approach the smaller data values “bubble up” to the front of the array …
- Each iteration puts the smallest unsorted element into its correct place, but it also makes changes in the locations of the other elements in the array.
- The first iteration puts the smallest element in the array into the first array position:
  - starting with the last array element, we compare successive pairs of elements, swapping whenever the bottom element of the pair is smaller than the one above it
  - in this way the smallest element “bubbles up” to the top of the array.
- The next iteration puts the smallest element in the unsorted part of the array into the second array position, using the same technique
- The rest of the sorting process continues in the same way

Bubble Sort Algorithm

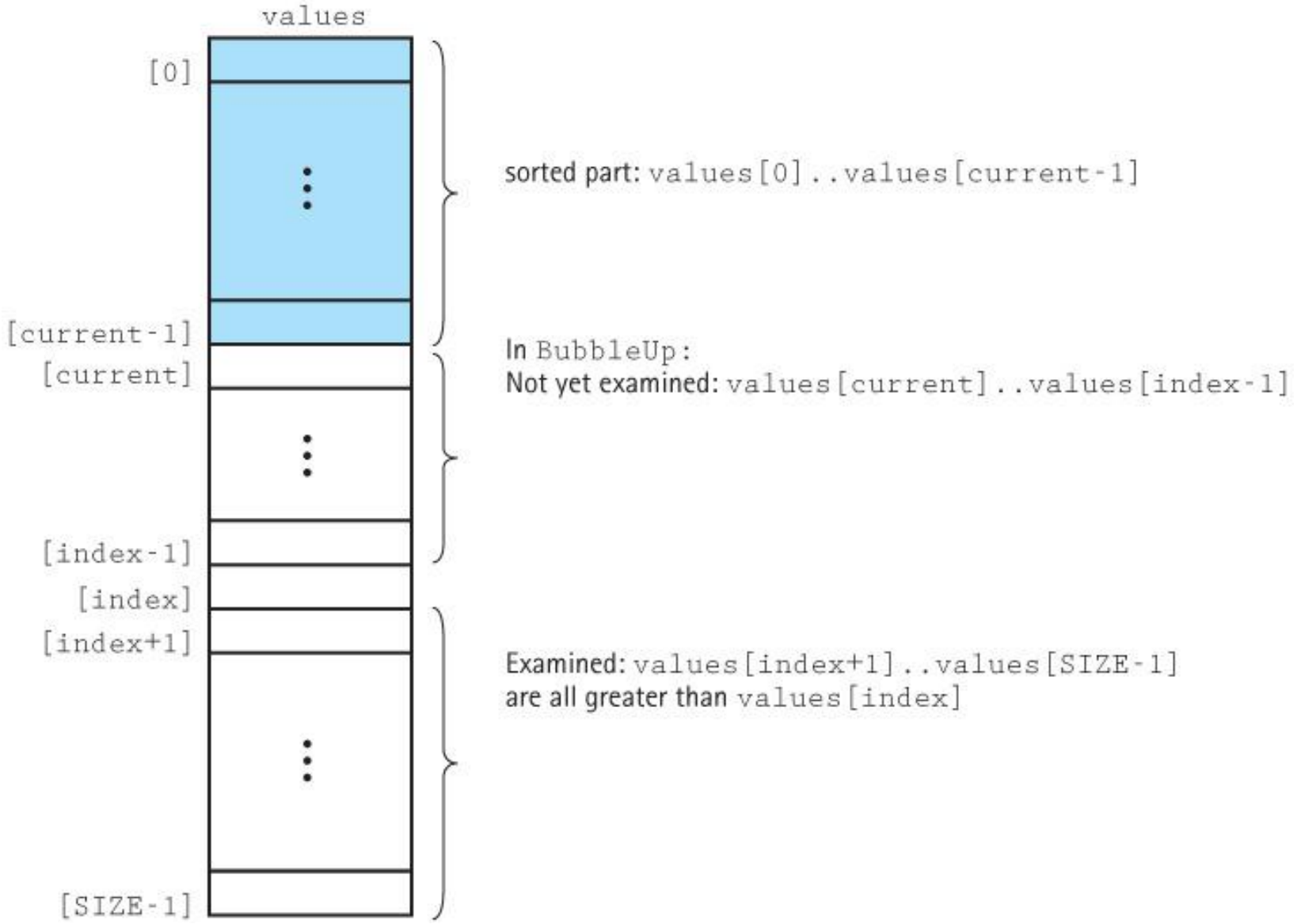
BubbleSort
Set current to the index of first element in the array
while more elements in unsorted part of array
    "Bubble up" the smallest element in the unsorted part,
        causing intermediate swaps as needed
  Shrink the unsorted part of the array by incrementing current

bubbleUp(startIndex, endIndex)
for index going from endIndex DOWNTO startIndex +1
    if values[index] < values[index - 1]
        Swap the value at index with the value at index - 1

An example is depicted on the following slide …


(a)
values
[0] 36
[1] 24
[2] 3
[3] 12
[4] 6
values
[0] 36
[1] 24
[2] 3
[3] 6
[4] 12
OK
values
[0] 36
[1] 24
[2] 3
[3] 6
[4] 12
values
[0] 36
[1] 3
[2] 24
[3] 6
[4] 12
values
[0] 3
[1] 36
[2] 24
[3] 6
[4] 12
(b)
values
[0] 3
[1] 36
[2] 24
[3] 6
[4] 12
OK
values
[0] 3
[1] 36
[2] 24
[3] 6
[4] 12
values
[0] 3
[1] 36
[2] 6
[3] 24
[4] 12
values
[0] 3
[1] 6
[2] 36
[3] 24
[4] 12
(c)
values
[0] 3
[1] 6
[2] 36
[3] 24
[4] 12
values
[0] 3
[1] 6
[2] 36
[3] 12
[4] 24
values
[0] 3
[1] 6
[2] 12
[3] 36
[4] 24
(d)
values
[0] 3
[1] 6
[2] 12
[3] 36
[4] 24
values
[0] 3
[1] 6
[2] 12
[3] 24
[4] 36

Bubble Sort Snapshot



Bubble Sort Code

```
static void bubbleUp(int startIndex, int endIndex)
// Switches adjacent pairs that are out of order
// between values[startIndex]..values[endIndex]
// beginning at values[endIndex].
{
  for (int index = endIndex; index > startIndex; index--)
    if (values[index] < values[index - 1])
      swap(index, index - 1);
}

static void bubbleSort()
// Sorts the values array using the bubble sort algorithm.
{
  int current = 0;
  while (current < SIZE - 1)
  {
    bubbleUp(current, SIZE - 1);
    current++;
  }
}
```

Bubble Sort Analysis

- Analyzing the work required by `bubbleSort` is the same as for the straight selection sort algorithm.
- The comparisons are in `bubbleUp`, which is called $N - 1$ times.
- There are $N - 1$ comparisons the first time, $N - 2$ comparisons the second time, and so on.
- Therefore, `bubbleSort` and `selectionSort` require the same amount of work in terms of the number of comparisons.
- The Bubble Sort algorithm is $O(N^2)$

Insertion Sort

- In Chapter 6, we will create a sorted list by inserting each new element into its appropriate place in an array. Insertion Sort uses the same approach for sorting an array.

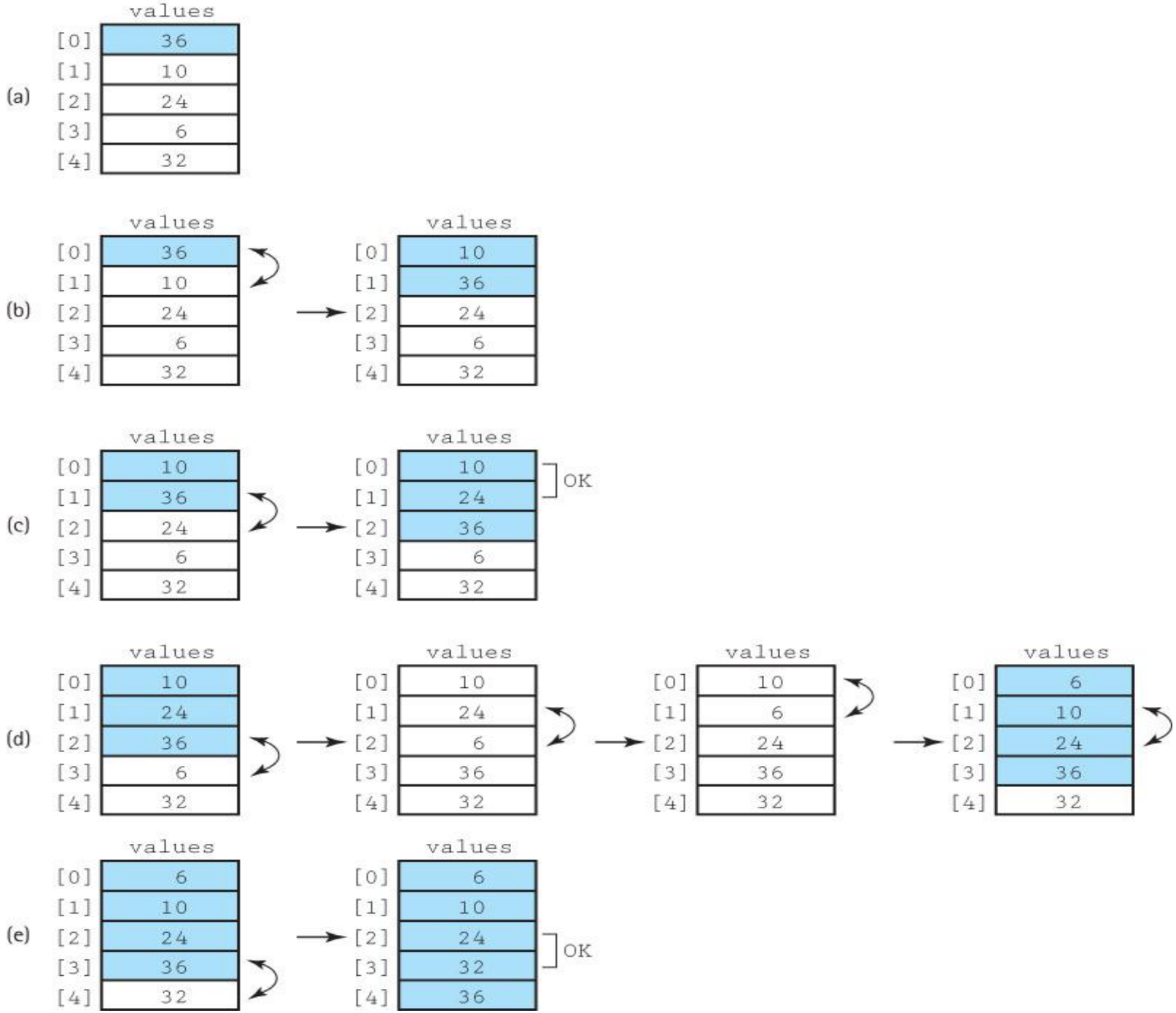
- Each successive element in the array to be sorted is inserted into its proper place with respect to the other, already sorted elements.

- As with the previous sorts, we divide our array into a sorted part and an unsorted part.
  - Initially, the sorted portion contains only one element: the first element in the array.
  - Next we take the second element in the array and put it into its correct place in the sorted part; that is, values[0] and values[1] are in order with respect to each other.
  - Next the value in values[2] is put into its proper place, so values[0]..values[2] are in order with respect to each other.
  - This process continues until all the elements have been sorted.

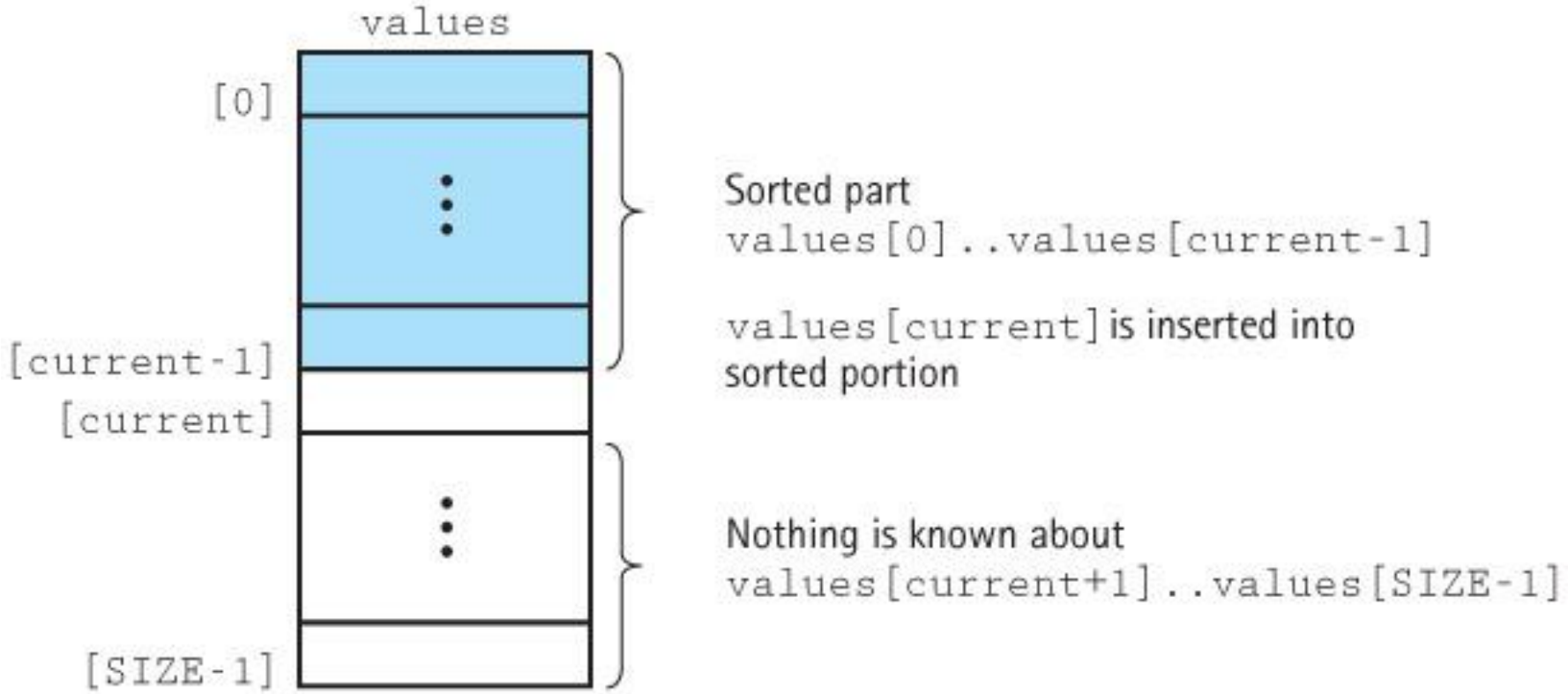
Insertion Sort Algorithm

insertionSort
for count going from 1 through SIZE - 1
    insertElement(0, count)
InsertElement(startIndex, endIndex)
Set finished to false
Set current to endIndex
Set moreToSearch to true
while moreToSearch AND NOT finished
    if values[current] < values[current - 1]
        swap(values[current], values[current - 1])
        Decrement current
        Set moreToSearch to (current does not equal startIndex)
    else
        Set finished to true

An example is depicted on the following slide …


values
[0] 36
[1] 10
[2] 24
[3] 6
[4] 32
(a)
values
[0] 36
[1] 10
[2] 24
[3] 6
[4] 32
values
[0] 10
[1] 36
[2] 24
[3] 6
[4] 32
(b)
values
[0] 10
[1] 36
[2] 24
[3] 6
[4] 32
values
[0] 10
[1] 24
[2] 36
[3] 6
[4] 32
OK
(c)
values
[0] 10
[1] 24
[2] 36
[3] 6
[4] 32
values
[0] 10
[1] 24
[2] 6
[3] 36
[4] 32
values
[0] 10
[1] 6
[2] 24
[3] 36
[4] 32
values
[0] 6
[1] 10
[2] 24
[3] 36
[4] 32
(d)
values
[0] 6
[1] 10
[2] 24
[3] 36
[4] 32
values
[0] 6
[1] 10
[2] 24
[3] 32
[4] 36
OK
(e)

Insertion Sort Snapshot



Insertion Sort Code

```
static void insertElement(int startIndex, int endIndex)
// Upon completion, values[0]..values[endIndex] are sorted.
{
  boolean finished = false;
  int current = endIndex;
  boolean moreToSearch = true;
  while (moreToSearch && !finished)
  {
    if (values[current] < values[current - 1])
    {
      swap(current, current - 1);
      current--;
      moreToSearch = (current != startIndex);
    }
    else
      finished = true;
  }
}

static void insertionSort()
// Sorts the values array using the insertion sort algorithm.
{
  for (int count = 1; count < SIZE; count++)
    insertElement(0, count);
}
```

Insertion Sort Analysis

- The general case for this algorithm mirrors the `selectionSort` and the `bubbleSort`, so the general case is $O(N^2)$.

- But `insertionSort` has a "best" case:
  - The data are already sorted in ascending order
  - `insertElement` is called *N* times, but only one comparison is made each time and no swaps are necessary.
    - (we will verify this)

- The maximum number of comparisons is made only when the elements in the array are in reverse order.
  - We will attempt to show this too.

Lab Time

1. **Open Sort.java:**
2. **Make a similar chart for the bubble sort routines and insertion sort routines**
3. **Lets talk about it**
4. **Recode Sort.java, and why you might need it!**

Number of Elements	**Number of Comparisons**
10	??
20	???
100	????
1,000	??????
10,000	????????

10.3 O(N $\log_2$N) Sorts

- O(N^2) sorts and are very time consuming for sorting large arrays.

- Several sorting methods that work better when *N* is large are presented in this section.

- The efficiency of these algorithms is achieved at the expense of the simplicity seen in the straight selection, bubble, and insertion sorts.

Chapter 6: The List ADT

6.1 – Comparing Objects Revisited

6.2 – Lists

6.3 – Formal Specification

6.7 – Reference-Based Implementations

6.6 – The Binary Search Algorithm

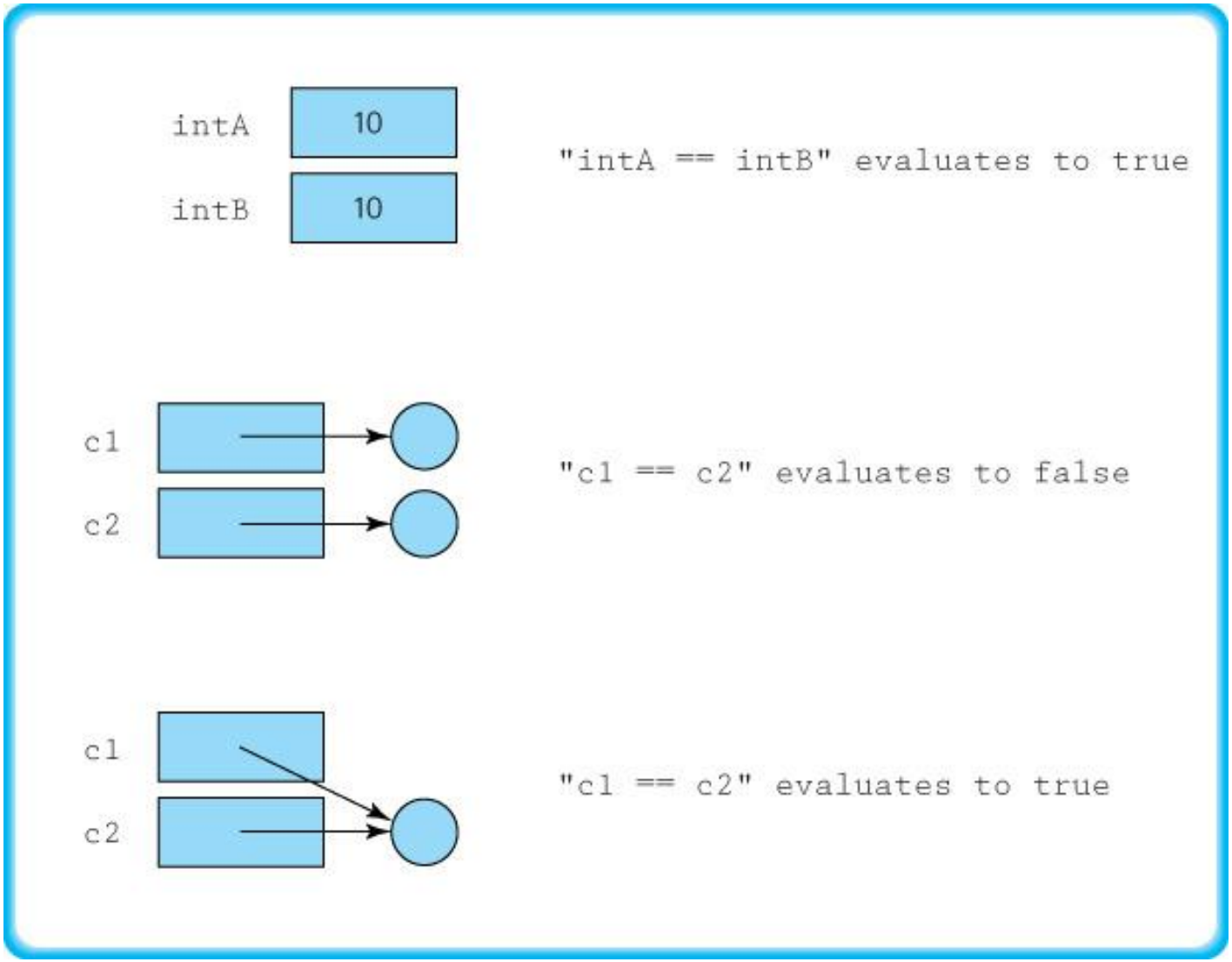
6.4 – Array-Based Implementation

6.5 – Applications: Poker, Golf, and Music

6.1 Comparing Objects Revisited

- Many list operations require us to compare the values of objects, for example
  - check whether a given item is on our to-do list
  - insert a name into a list in alphabetical order
  - delete the entry with the matching serial number from a parts inventory list
- Therefore we need to understand our options for such comparisons.

Using the comparison (==) operator



Using the `equals` method

- Since `equals` is exported from the `Object` class it can be used with objects of any Java class.
- If `c1` and `c2` are objects of the class `Circle`, then we can compare them using

```
c1.equals(c2)
```

- But this method, as defined in the `Object` class, acts much the same as the comparison operator. It returns true if and only if the two variables reference the same object.
- However, we can redefine the `equals` method to fit the goals of the class.

Using the `equals` method

- A reasonable definition for equality of Circle objects is that they are equal if they have equal radii.

- To realize this approach we define the equals method of our `Circle` class to use the `radius` attribute:

```
// Precondition: circle != null
//
// Returns true if the circles have the same radius,
// otherwise returns false.  public boolean
equals(Circle circle)
{
  if (this.radius == circle.radius)
    return true;
  else
    return false;
}
```

Ordering objects

- In addition to checking objects for equality, there is another type of comparison we need.

- To support a sorted list we need to…
  - be able to tell when one object is
    - less than,
    - equal to,
    - or greater than another object.

- The Java library provides an interface, called `Comparable`
  - used to ensure that a class provides this functionality

The `Comparable` Interface

- The `Comparable` interface consists of exactly one abstract method:

```
// Returns a negative integer, zero, or a positive
// integer as this object is less than, equal to,
// or greater than the specified object.
public int compareTo(T o);
```

- The `compareTo` method
  - returns an integer value that indicates the relative "size" relationship between the object upon which the method is invoked and the object passed to the method as an argument.

Using the `Comparable` Interface

- Objects of a class that implements the `Comparable` interface are called…
  - Comparable objects.

- To ensure that all elements placed on our sorted list support the `compareTo` operation,…
  - we require them to be Comparable objects.

- For example, see the definition of `compareTo` for our `Circle` class on the next slide …

A `compareTo` Example

```
public int compareTo(Circle circle)
// Precondition: o != null
//
// Returns a negative integer, zero, or a positive
//   integer as this object is less than, equal to,
//   or greater than the parameter object.
{
  if (this.radius < circle.radius)
    return -1;
  else
    if (this.radius == (circle.radius)
      return 0;
    else
      return 1;
}
```

- Note that the `equals` method and the `compareTo` method of our `Circle` class are compatible with each other.

- To simplify discussion we sometimes refer to the attribute, or combination of attributes, of an object used by the `compareTo` method to determine the logical order of a collection as the **key** of the collection.

6.2 Lists

- **List** A collection that exhibits a linear relationship among its elements
- **Linear relationship** Each element except the first has a unique predecessor, and each element except the last has a unique successor
- **Size** The number of elements in a list; the size can vary over time

Varieties of Lists

- **Unsorted list**
  - elements are placed in no particular order
  - the only relationship between data elements is the list predecessor and successor relationships
- **Sorted list**
  - elements are sorted by some property
  - there is an ordered relationship among the elements in the list, reflected by their relative positions
- **Indexed list**
  - A list in which each element is associated with an index value

Assumptions for our Lists

- Our lists are *unbounded*
- We *allow* duplicate elements on our lists
- We do *not* support null elements
- We have *minimal preconditions* on our operations
- Our sorted lists are sorted in *increasing* order
  - as defined by the `compareTo` operation applied to list objects
- The `equals` and `compareTo` methods of our sorted list elements are consistent
- The indices in use at any given time are *contiguous*
  - starting at 0

6.3 Formal Specification

- We define a small, but useful, set of operations for use with our lists.
- We capture the formal specifications of our List ADT using the Java interface construct.
- We pull common list method descriptions together into a single interface, called `ListInterface`.
- We extend the `ListInterface` with a separate interface for indexed lists, since indexing a list allows additional operations.

List Iteration

- Because a list has a linear relationship among its elements, we can support *iteration* through a list.

- Iteration means that we provide a mechanism to process the entire list, element by element, from the first element to the last element.

- Each of our list variations provides the operations `reset` and `getNext` to support this activity.

List Operations

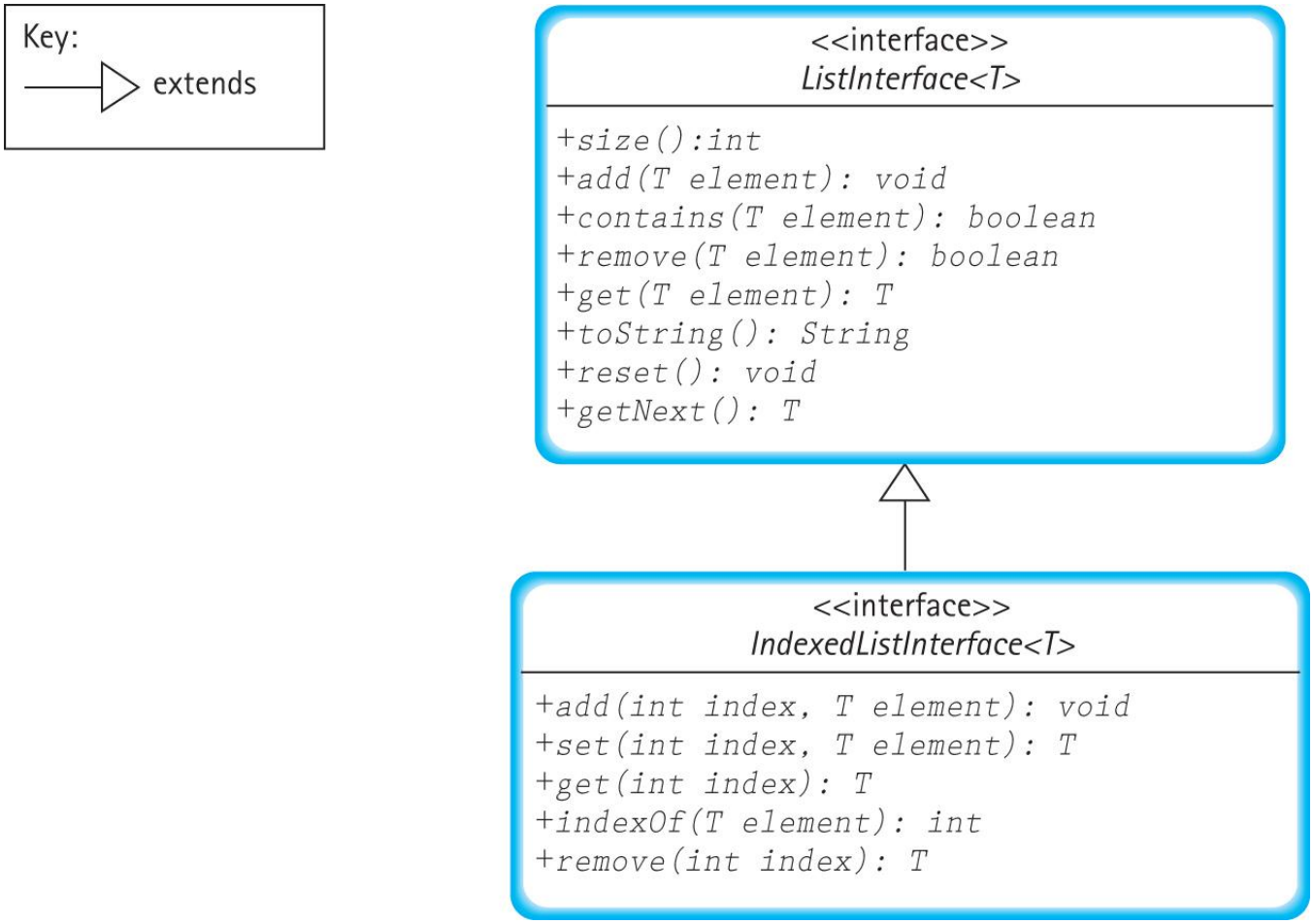
- **size**: Returns the number of elements on the list.
- **add**: Adds an element to the list.
- **contains**: Passed an argument and returns a boolean indicating whether the list contains an equivalent element.
- **remove**: Passed an argument and, if an equivalent element exists on the list, removes one instance of that element. Returns a boolean value indicating whether an element was actually removed.
- **get**: Passed an argument, it returns an equivalent object if one exists on the list. If not, returns null.
- **toString**: Returns a nicely formatted string representing the list.
- **reset**: Initializes the list. Sets the current position (the position of the next element to be processed) to the first element on the list.
- **getNext**: Returns the next element, and updates the current position.

List Operations

- Our Indexed List also supports the operations
  - **add**: Passed an element and an integer index it adds the element to the list at the position index
  - **set**: Passed an element and an integer index it replaces the current element at the position index with the argument element
  - **get**: Passed an integer index it returns the element from the list at that position
  - **indexOf**: Passed an element it returns the index of the first such matching element (or -1)
  - **remove**: Passed an integer index it removes the element at that index

- Each method that accepts an index as an argument throws an exception if the index is invalid.

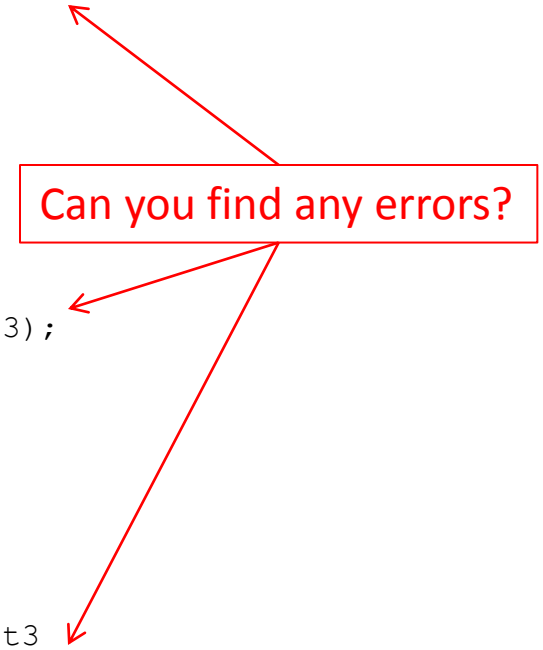
UML Diagram of our List Interfaces



Example Use

Code Section

```
ListInterface<String> list1
  = new ArrayUnsortedList<String>(3);
list1.add("Wirth");
list1.add("Dykstra");
list1.add("DePasquale");
list1.add("Dahl");
list1.add("Nygaard");
list1.remove("DePasquale");

ListInterface<String> list2
  = new ArraySortedList<String>(3);
list2.add("Wirth");
list2.add("Dykstra");
list2.add("DePasquale");
list2.add("Dahl");
list2.add("Nygaard");
list2.remove("DePasquale");

IndexedListInterface<String> list3
  = new ArrayIndexedList<String>(3);
list3.add(0, "Wirth");
list3.add(0, "Dykstra");
list3.add(0, "DePasquale");
list3.add(3, "Dahl");
list3.add(2, "Nygaard");
list3.remove("DePasquale");

System.out.print("Unsorted ");
System.out.println(list1);
System.out.print("Sorted ");
System.out.println(list2);
System.out.print("Indexed ");
System.out.println(list3);
```



Output

Unsorted List:
- Wirth
- Dykstra
- Nygaard
- Dahl

Sorted List:
- Dahl
- Dykstra
- Nygaard
- Wirth

Indexed List:
[0] Dykstra
[1] Nygaard
[2] Wirth
[3] Dahl

6.7 Reference-Based Implementation

- In this section we develop list implementations using references (links).

- We follow the same basic pattern of development here that we did when we developed list implementations using arrays.
  - Our reference-based implementations fulfill the interfaces we developed in Section 6.3.
  - We first implement an unsorted version, and then extend it with a sorted version.

Reference Based Lists

- As we did for our reference-based stacks and queues, we use the `LLNode` class from the `support` package to provide our nodes.
  - the information attribute of a node contains the list element
  - the link attribute contains a reference to the node holding the next list element

- We maintain a variable, `list`, that references the first node on the list.

Reference Based Lists

- We'll implement both unsorted and sorted lists using the linked approach.

- We do not implement indexed lists. There is no simple way to efficiently implement the add, set, get, and remove operations involving index arguments.

The `RefUnsortedList` Class

- Implements the `ListInterface`
- The `size`, `toString`, `reset`, and `getNext` methods are straightforward
- Because the list is unsorted, and order of the elements is not important, we can just `add` new elements to the front of the list :

```
public void add(T element)
// Adds element to this list.
{
  LLNode<T> newNode = new LLNode<T>(element);
  newNode.setLink(list);
  list = newNode;
  numElements++;
}
```

The `find` helper method

```
protected void find(T target)
{
  location = list;
  found = false;

  while (location != null)
  {
    if (location.getInfo().equals(target))
    {
      found = true;
      return
    }
    else
    {
      previous = location;
      location = location.getLink();
    }
  }
}
```

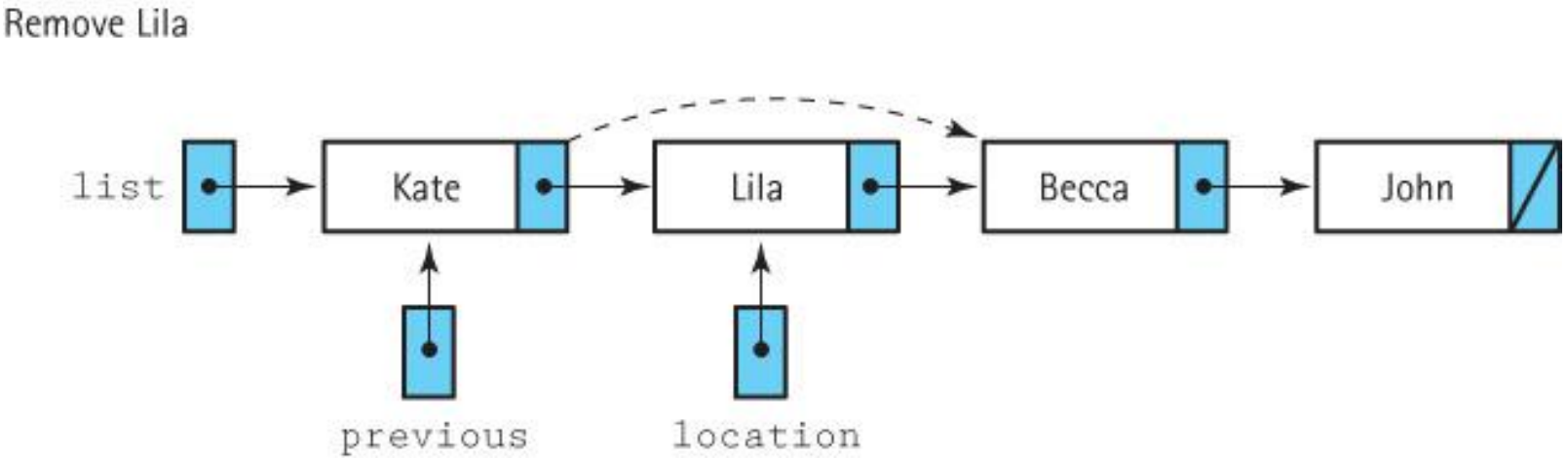
Used by the `contains`, `get`, and `remove` methods.

Makes `contains` and `get` straightforward.

Sets a variable named `previous` - used by the `remove` method.

The `remove` method

- To remove an element, we first find it using the `find` method, which sets the `location` variable to indicate the target element and sets the `previous` variable to a reference in the previous node.
- We can now change the link of the previous node to reference the node following the one being removed.
- Removing the first node must be treated as a special case because the main reference to the list (`list`) must be changed.



The `remove` method

```
public boolean remove (T element)
// Removes an element e from this list such that e.equals(element)
// and returns true; if no such element exists returns false.
{
  find(element);
  if (found)
  {
    if (list == location)
      list = list.getLink();    // remove first node
    else
      previous.setLink(location.getLink());  // remove node at location

    numElements--;
  }
  return found;
}
```

The `RefSortedList` Class

- Implements the `ListInterface`
- Extends the `RefUnsortedList` with an `add` method
- Adding an element to a reference-based sorted list requires three steps:
  1. Find the location where the new element belongs
  2. Create a node for the new element
  3. Correctly link the new node into the identified location

1. Finding the location where the new element belongs

- To link the new node into the identified location, we also need a reference to the previous node.

- While traversing the list during the search stage, each time we update the `location` variable, we first save its value in a `prevLoc` variable:

```
prevLoc = location;
location = location.getLink();
```

2. Create a node for the new element

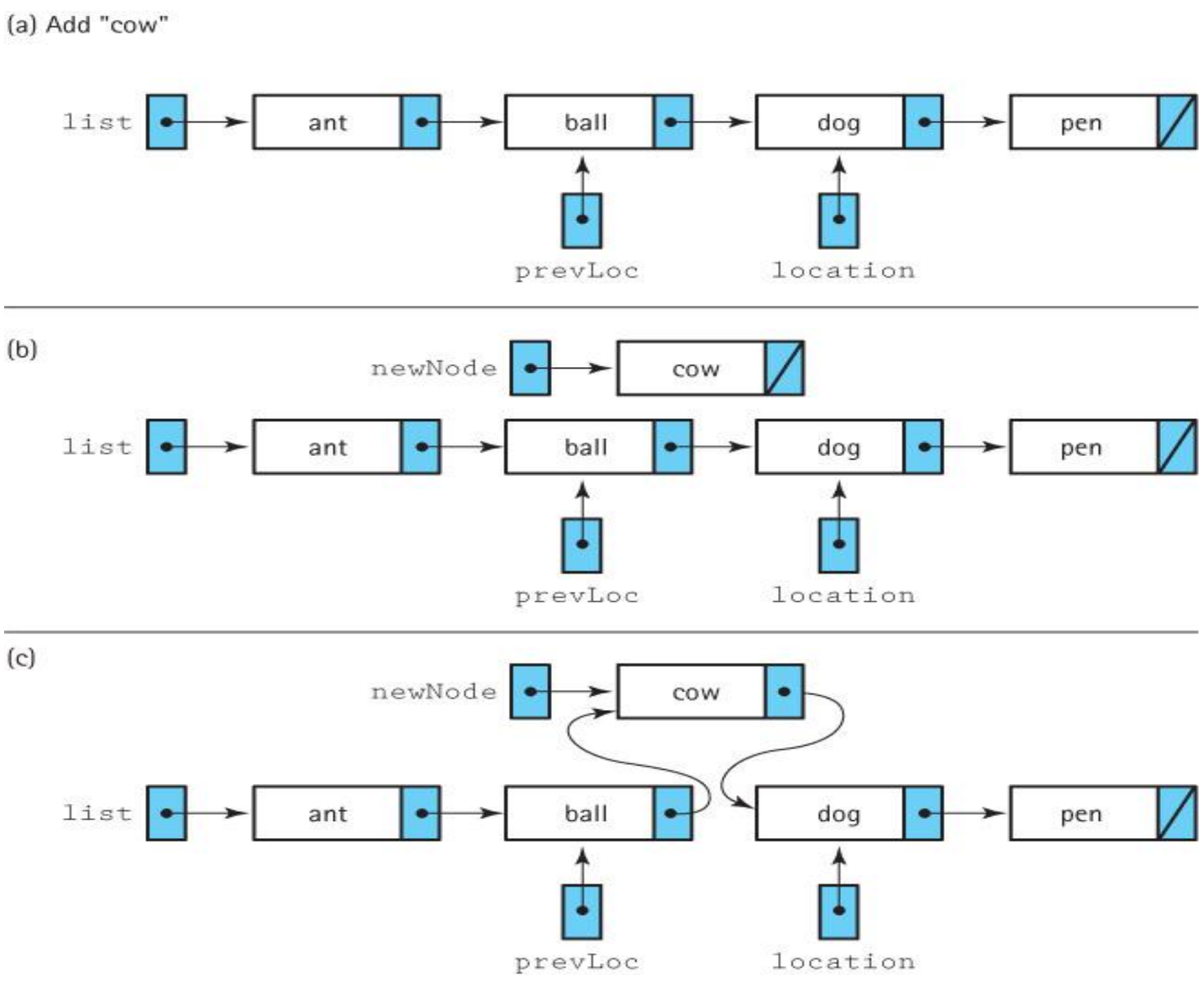
- instantiate a new `LLNode` object called `newNode`, passing its constructor the new element for use as the information attribute of the node

```
LLNode<T> newNode = new LLNode<T>(element);
```

3. Correctly link the new node into the identified location

- We change the link in the `newNode` to reference the node indicated by `location` and change the link in our `prevLoc` node to reference the `newNode`:

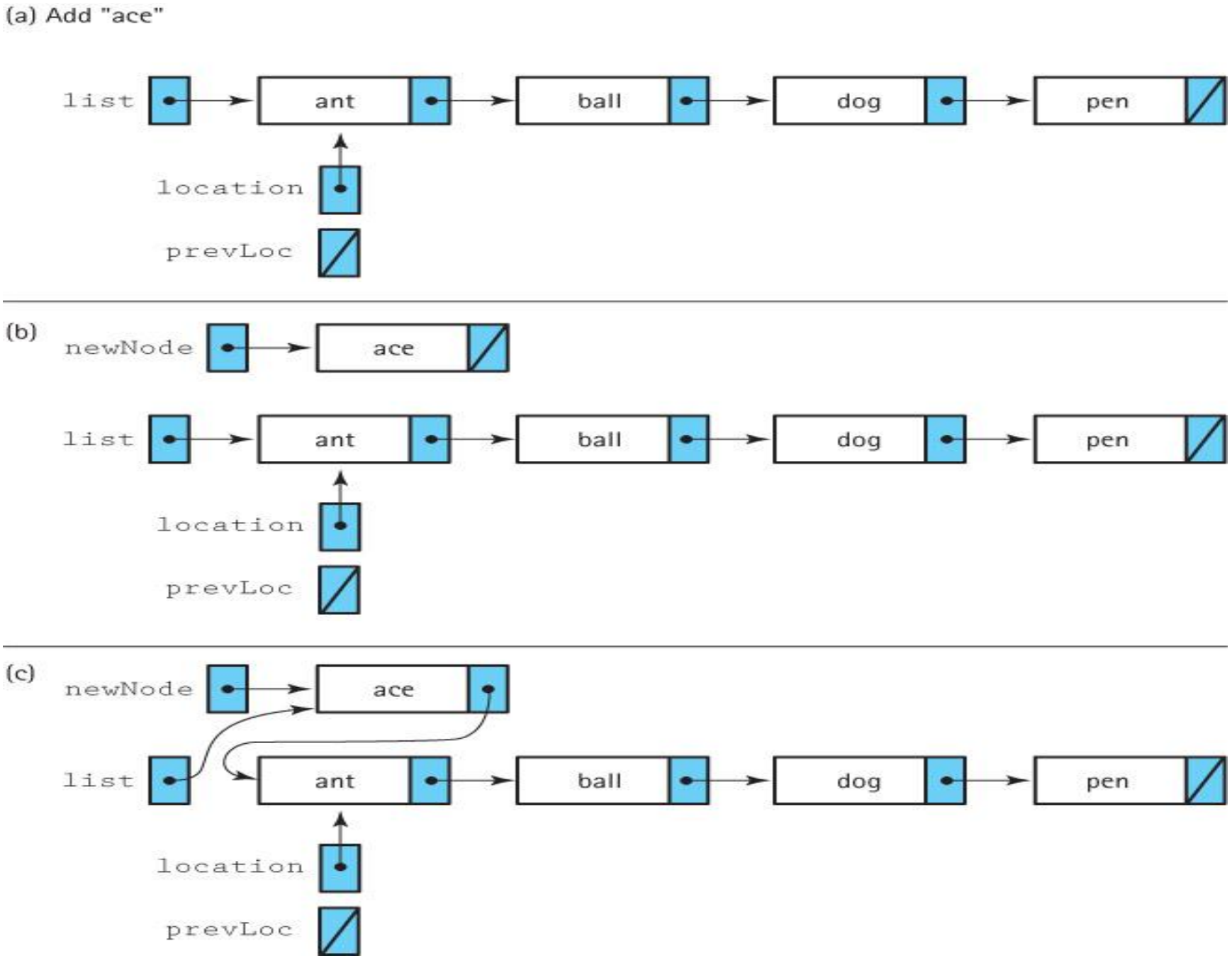
```
newNode.setLink(location);
prevLoc.setLink(newNode);
```


(a) Add "cow"
list
ant
ball
dog
pen
prevLoc
location
(b)
newNode
cow
list
ant
ball
dog
pen
prevLoc
location
(c)
newNode
cow
list
ant
ball
dog
pen
prevLoc
location

Special case – `location` indicates first node of list

- In this case we do not have a previous node
- We must change the main reference to the list (`list`)

```
 if (prevLoc == null)
{
  // Insert as first node.
  newNode.setLink(list);
  list = newNode;
}
```


(a) Add "ace"
list
ant
ball
dog
pen
location
prevLoc
(b)
newNode
ace
list
ant
ball
dog
pen
location
prevLoc
(c)
newNode
ace
list
ant
ball
dog
pen
location
prevLoc

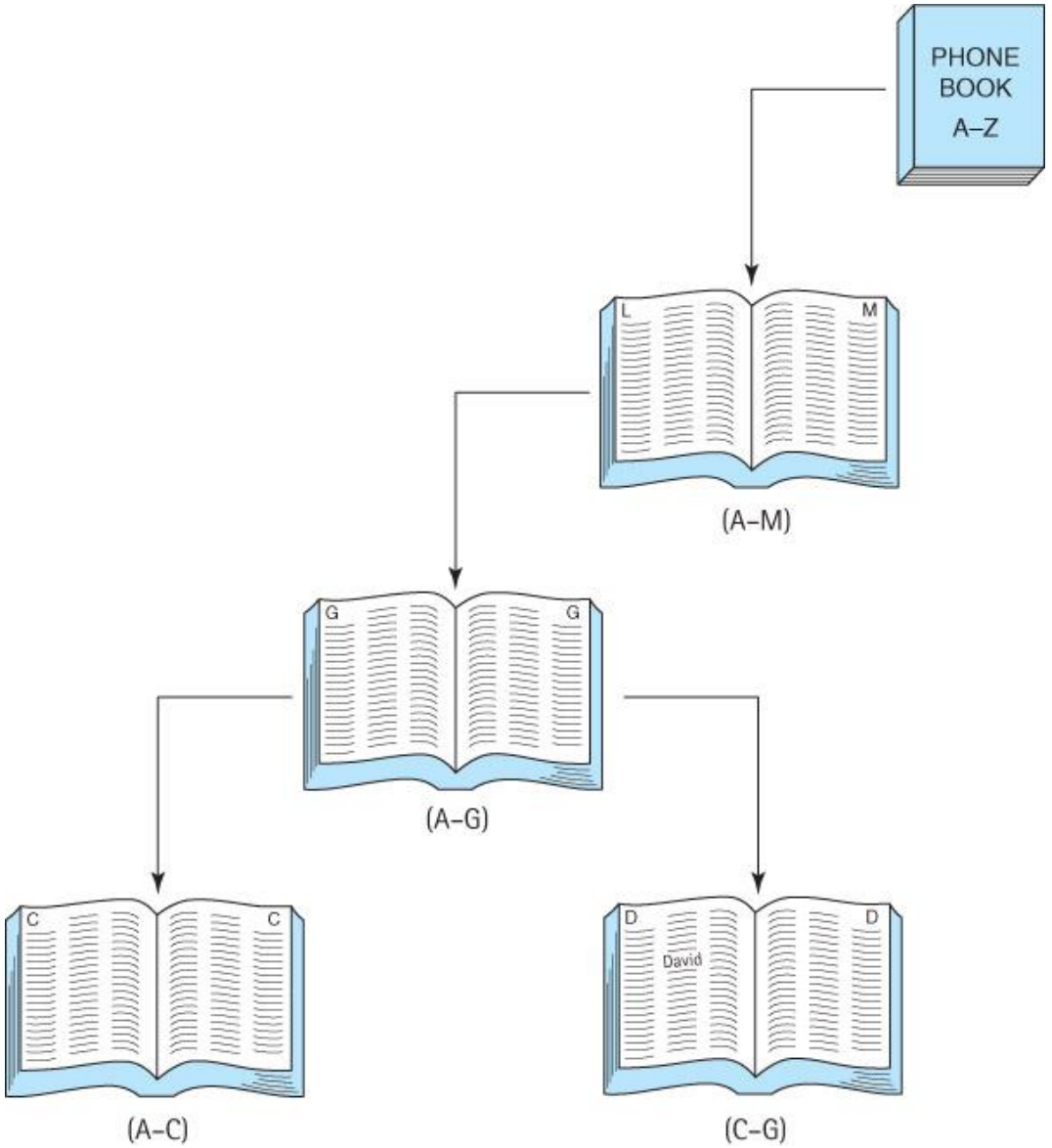
The `add` method

```
package ch06.lists; import support.LLNode;
public class RefSortedList<T extends Comparable<T>>
             extends RefUnsortedList<T>
             implements ListInterface<T> {
  public RefSortedList() { super(); }
  public void add(T element) {// Adds element to this list.
    LLNode<T> prevLoc;        // trailing reference
    LLNode<T> location;       // traveling reference
    T listElement;            // current list element being compared
    location = list;          // Set up search for insertion point.
    prevLoc = null;           // Set up search for insertion point.
    while (location != null){ // Find insertion point.
      listElement = location.getInfo();
      if (listElement.compareTo(element) < 0) {  // list element < add element
         prevLoc = location;
         location = location.getLink();}
      else break; }
    LLNode<T> newNode = new LLNode<T>(element);   // Prepare node for insertion.
    if (prevLoc == null) {    // Insert node into list.
      newNode.setLink(list);  // Insert as first node.
      list = newNode; }
    else {                    // Insert elsewhere.
      newNode.setLink(location);
      prevLoc.setLink(newNode); }
    numElements++;
  }
}
```

Lab2: Reference based lists

- Open the files in Lab 2 in eclipse and code according to the comments

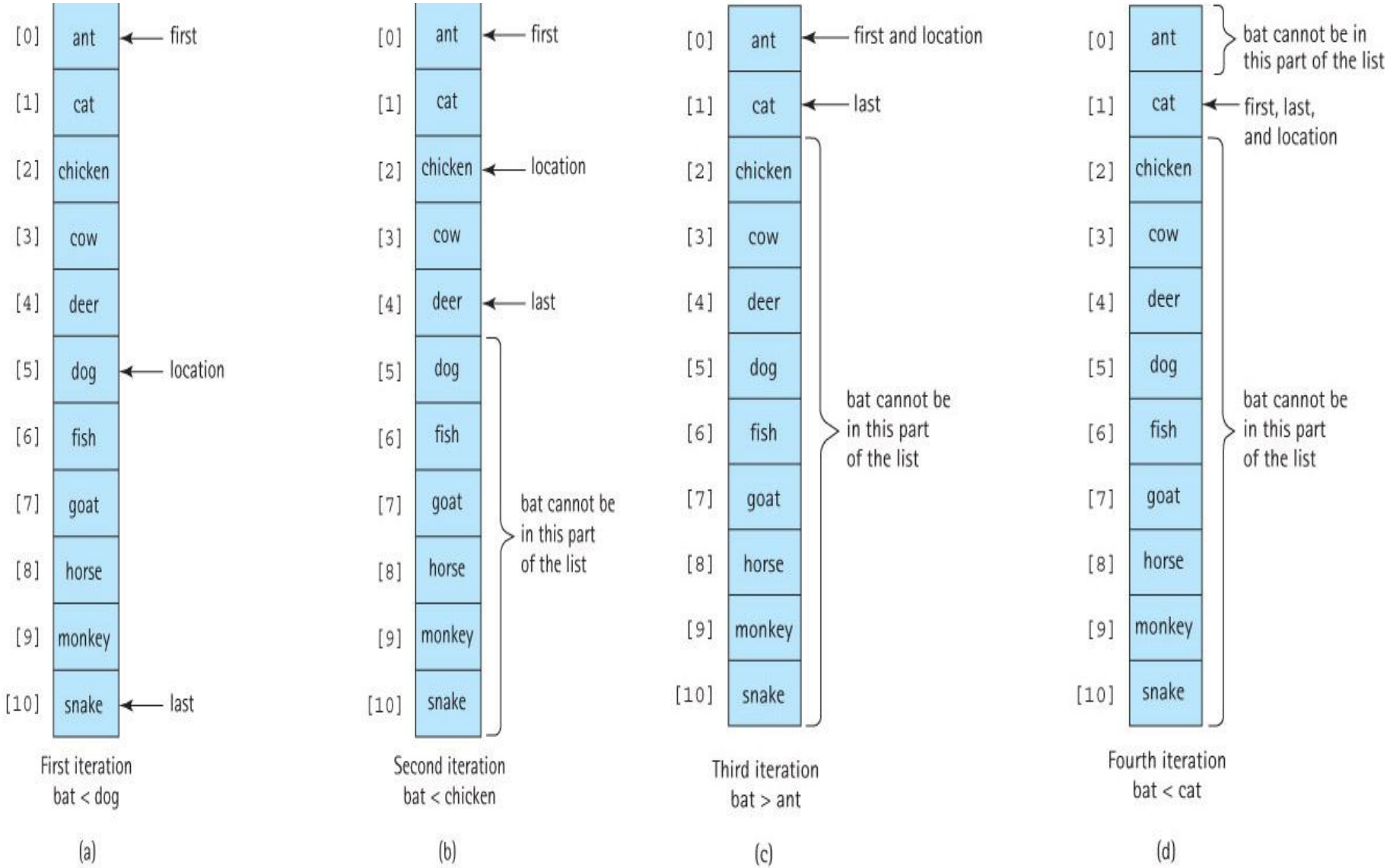
6.6 The Binary Search Algorithm



Searching in a phone book

- Suppose we are looking for “David” in a phone book
- We open the phone book to the middle and see that the names there begin with M
  - M is larger than (comes after) D
  - We can now limit our search to the section that contains A to M
- We turn to the middle of the first half and see that the names there begin with G
  - G is larger than D
  - We can now limit our search to the section that contains A to G
- We turn to the middle page of this section, and find that the names there begin with C
  - C is smaller than D
  - We can now limit our search to the section that contains C to G
- And so on, until we are down to the single page that contains the name “David.”

Searching for “bat” in a sorted array



The revised `find` method

```
protected void find(T target)
{
  int first = 0;
  int last = numElements - 1;
  int compareResult;
  Comparable targetElement = (Comparable)target;

  found = false;
  while (first <= last)
  {
    location = (first + last) / 2;
    compareResult = targetElement.compareTo(list[location]);
    if (compareResult == 0)
    {
      found = true;
      break;
    }
    else if (compareResult < 0)
    // target element is less than element at location
      last = location - 1;
    else   // target element is greater than element at location
      first = location + 1;
  }
}
```

Recursive Binary Search

- Consider this informal description of the binary search algorithm:
  - *To search a list, check the middle element on the list – if it's the target element you are done, if it's less than the target element search the second half of the list, otherwise search the first half of the list.*

- There is something inherently recursive about this description:
  - We search the list by searching half the list.
  - The solution is expressed in smaller versions of the original problem: if the answer isn't found in the middle position, perform a binary search (a recursive call) to search the appropriate half of the list (a smaller problem).

The recursive `find` method

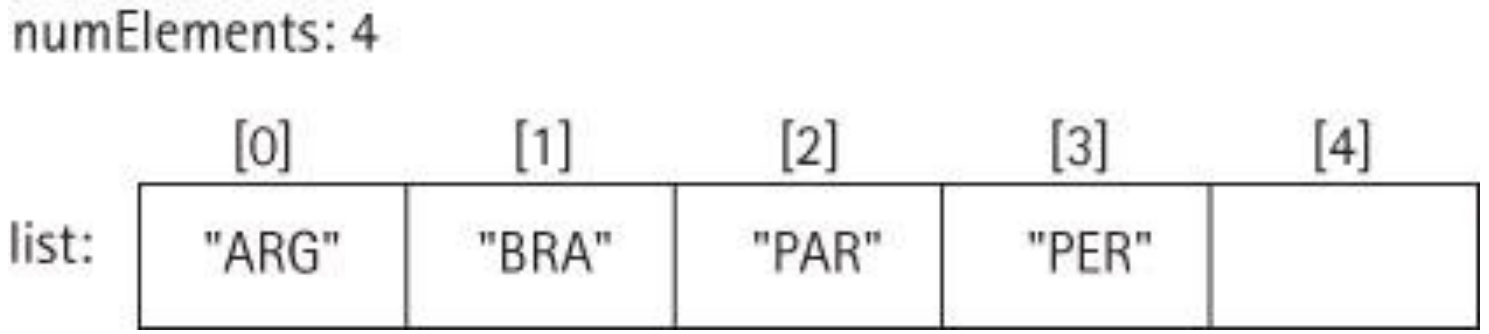
```
protected void recFind(Comparable target, int fromLocation, int toLocation)
{
  if (fromLocation > toLocation)  // Base case 1
    found = false;
  else
  {
    int compareResult;
    location = (fromLocation + toLocation) / 2;
    compareResult = target.compareTo(list[location]);
    if (compareResult == 0)       // Base case 2
      found = true;
    else if (compareResult < 0)
      // target is less than element at location
      recFind (target, fromLocation, location - 1);
    else
      // target is greater than element at location
      recFind (target, location + 1, toLocation);
  }
}

protected void find(T target)
{
  Comparable targetElement = (Comparable)target;
  found = false;
  recFind(targetElement, 0, numElements - 1);
}
```

Efficiency Analysis

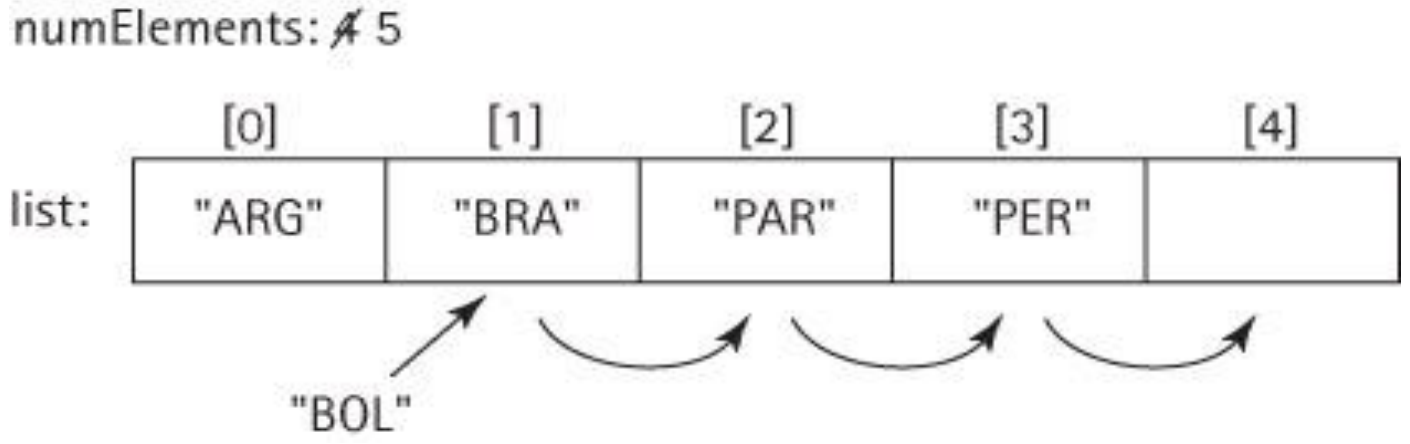
	Maximum Number of Iterations	
Length	Linear Search	Binary Search
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14

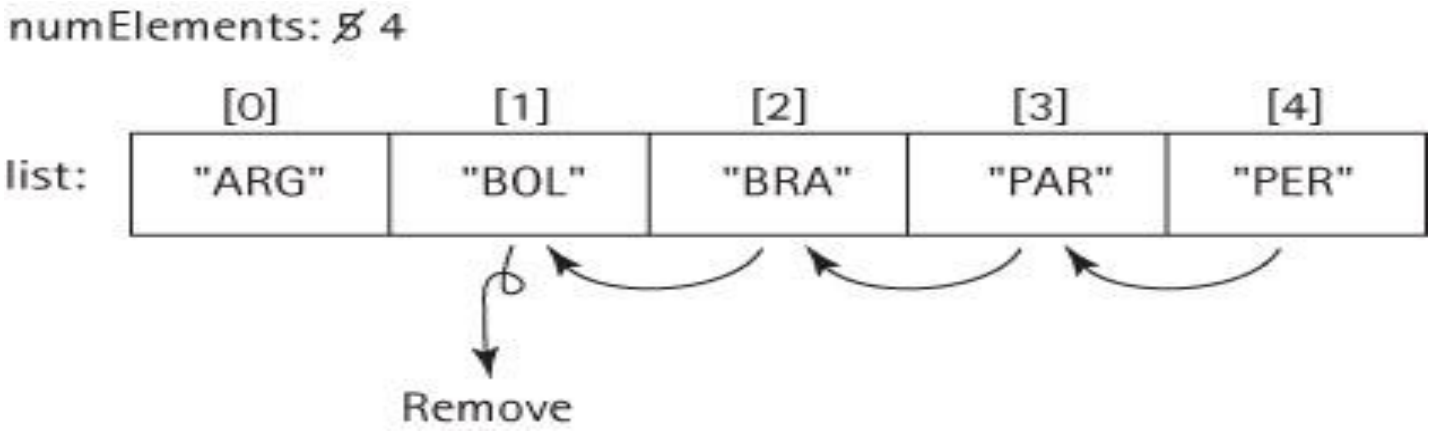
6.4 Array-Based Implementation

- Basic Approach:



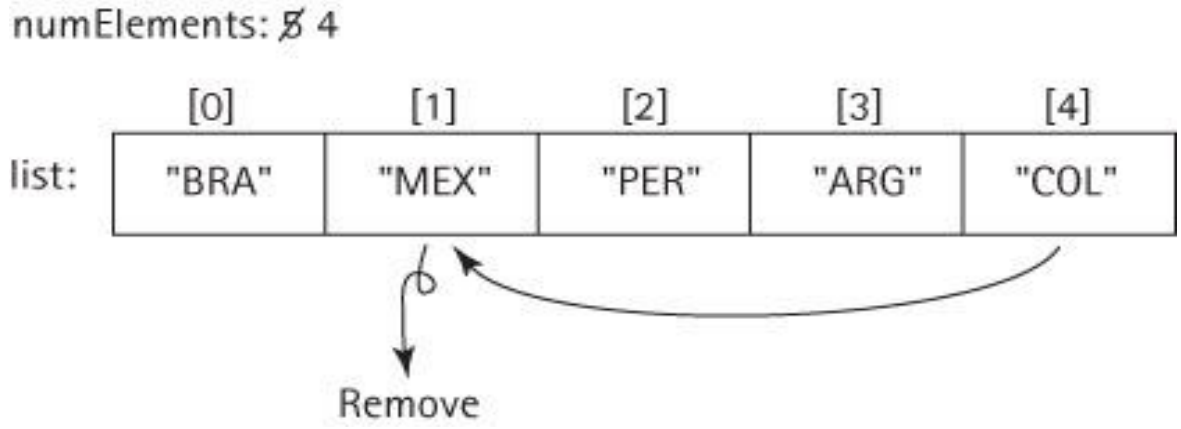
Adding/Removing from middle





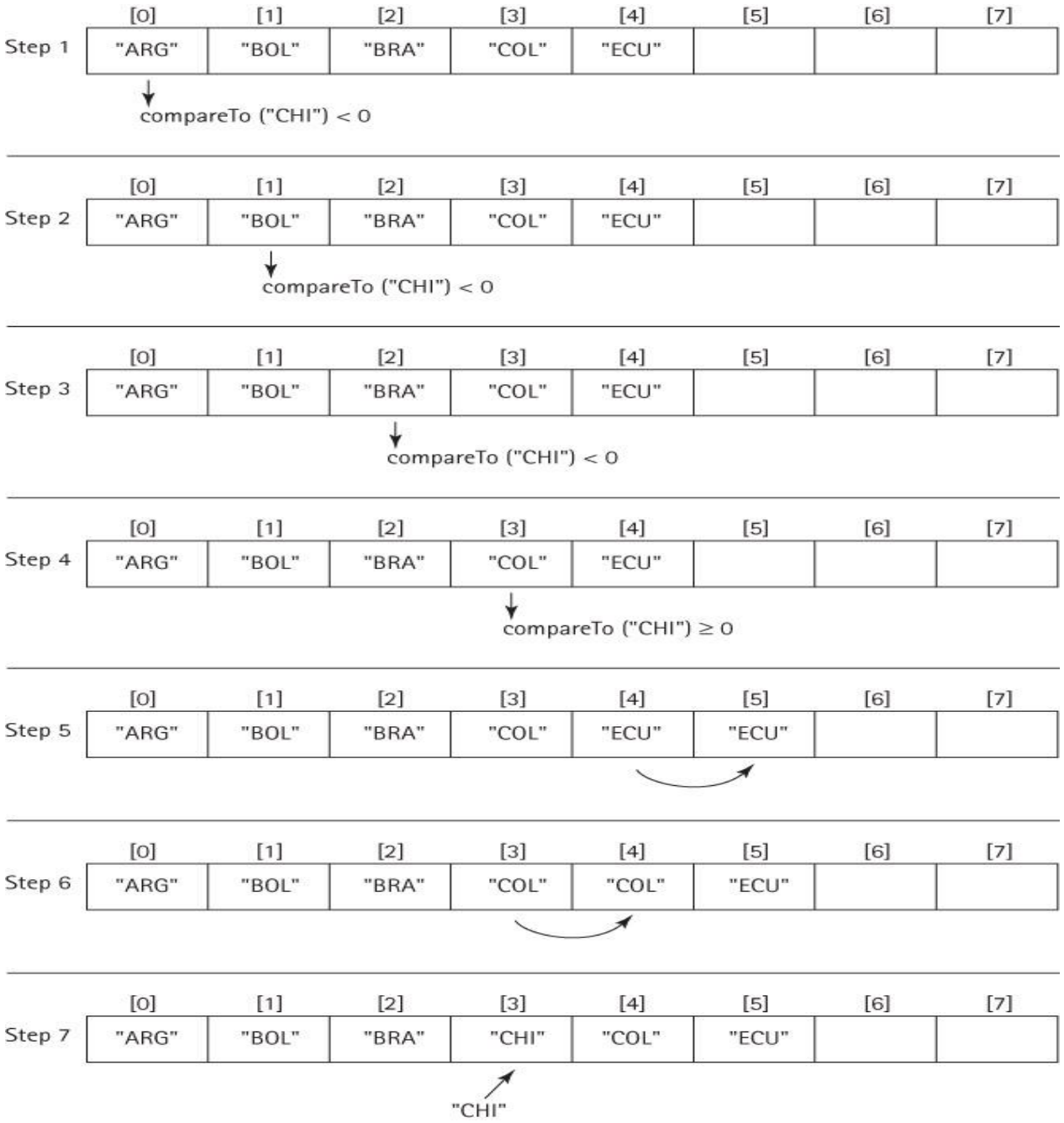
The `ArrayUnsortedList` Class

- Implements the `ListInterface`
- We create a helper method `find` for array search, which is used by several other methods
  - `find` sets `location` and `found` instance variables
- We are not concerned with the order in which elements are stored. So, for example, the `remove` method can be improved:



The `ArraySortedList` Class

- Implements the `ListInterface`
- Extends the `UnsortedListInterface`
- the `remove` method maintains sorted order
- the `add` method:
    - check to ensure that there is room for it, invoking our enlarge method if there is not.
    - find the place where the new element belongs.
    - create space for the new element.
    - put the new element in the created space.


Step 1
[0] [1] [2] [3] [4] [5] [6] [7]
"ARG" "BOL" "BRA" "COL" "ECU"
compareTo ("CHI") < 0
Step 2
[0] [1] [2] [3] [4] [5] [6] [7]
"ARG" "BOL" "BRA" "COL" "ECU"
compareTo ("CHI") < 0
Step 3
[0] [1] [2] [3] [4] [5] [6] [7]
"ARG" "BOL" "BRA" "COL" "ECU"
compareTo ("CHI") < 0
Step 4
[0] [1] [2] [3] [4] [5] [6] [7]
"ARG" "BOL" "BRA" "COL" "ECU"
compareTo ("CHI") ≥ 0
Step 5
[0] [1] [2] [3] [4] [5] [6] [7]
"ARG" "BOL" "BRA" "COL" "ECU" "ECU"
Step 6
[0] [1] [2] [3] [4] [5] [6] [7]
"ARG" "BOL" "BRA" "COL" "COL" "ECU"
Step 7
[0] [1] [2] [3] [4] [5] [6] [7]
"ARG" "BOL" "BRA" "CHI" "COL" "ECU"
"CHI"

The Sorted List `add` method

```
public void add(T element)
 // Adds element to this list.
 {
   T listElement;
   int location = 0;

   if (numElements == list.length)
     enlarge();

   while (location < numElements)
   {
     listElement = (T)list[location];
     if (((Comparable)listElement).compareTo(element) < 0)
        location++;
     else
       break;
   }

   for (int index = numElements; index > location; index--)
     list[index] = list[index - 1];

   list[location] = element;
   numElements++;
 }
```

Implementing ADTs
“by Copy” or “by Reference”

- When designing an ADT we have a choice about how to handle the elements.
  - **By Copy:** The ADT manipulates copies of the data used in the client program.
    - Making a valid copy of an object can be a complicated process.
  - **By Reference:** The ADT manipulates references to the actual elements passed to it by the client program.
    - Most commonly used approach (we use this one)

“By Copy” Notes

- Valid copies of an object are typically created using the object's `clone` method.
- Classes that provide a `clone` method must indicate this to the runtime system by implementing the `Cloneable` interface.
- Drawbacks:
  - Copy of object might not reflect up-to-date status of original object
  - Copying objects takes time, especially if the objects are large and require complicated deep-copying methods.
  - Storing extra copies of objects also requires extra memory.

“By Reference” Notes

- Because the client program retains a reference to the element, we say we have exposed the contents of the collection ADT to the client program.

- The ADT allows direct access to the individual elements of the collection by the client program through the client program’s own references.

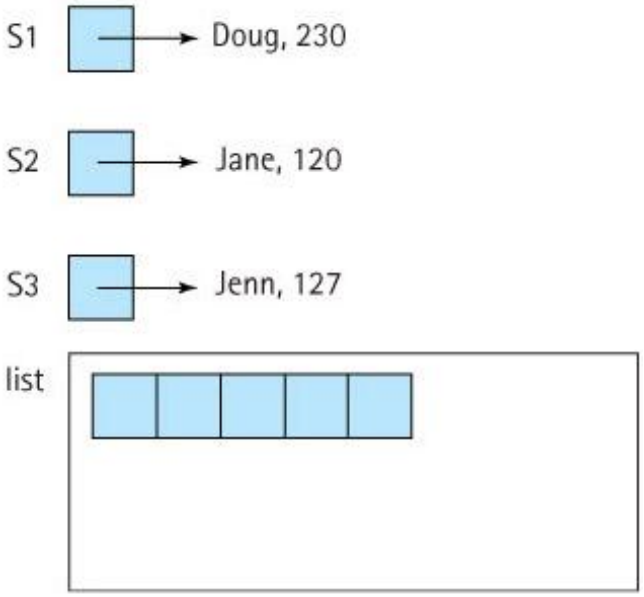
- Drawbacks:
  - We create aliases of our elements, therefore we must deal with the potential problems associated with aliases.
  - This situation is especially dangerous if the client program can use an alias to change an attribute of an element that is used by the ADT to determine the underlying organization of the elements – for example if it changes the key value for an element stored in a sorted list.

An Example

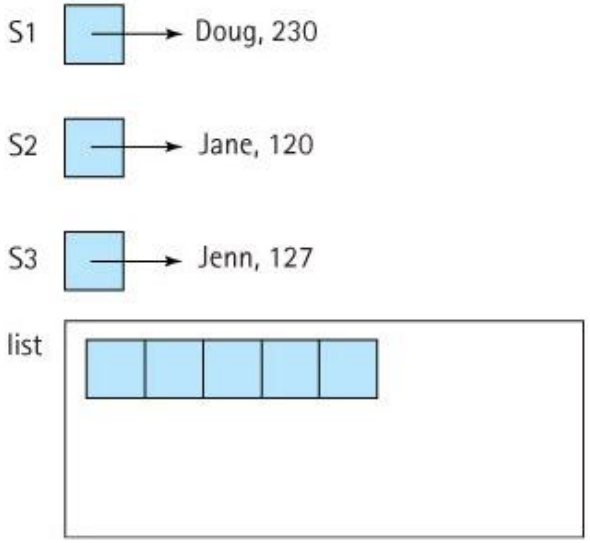
- The next three slides show the results of a sequence of operations using each of the two approaches to store a sorted list:
  - We have three objects that hold a person's name and weight (Slide 1)
  - We add the three objects onto a list that sorts objects by the variable weight
  - We transform one of the original objects with a diet method, that changes the weight of the object

Example Step 1: The Three Objects

By Copy Approach

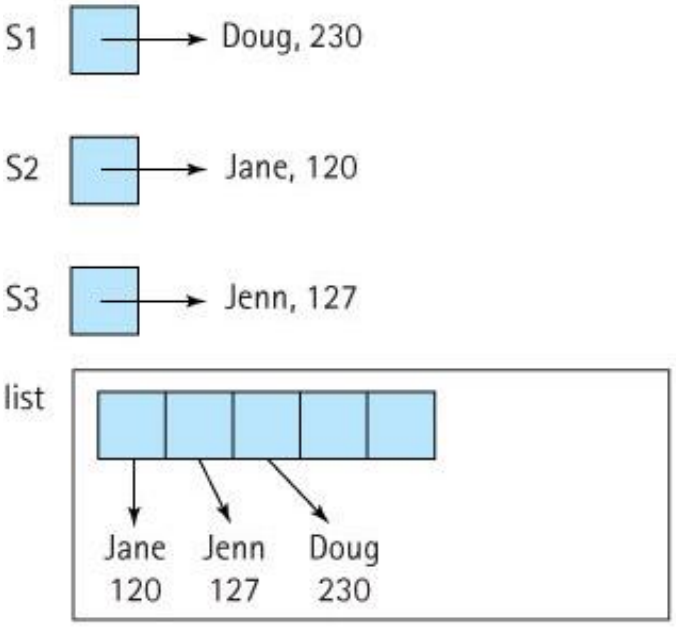


By Reference Approach

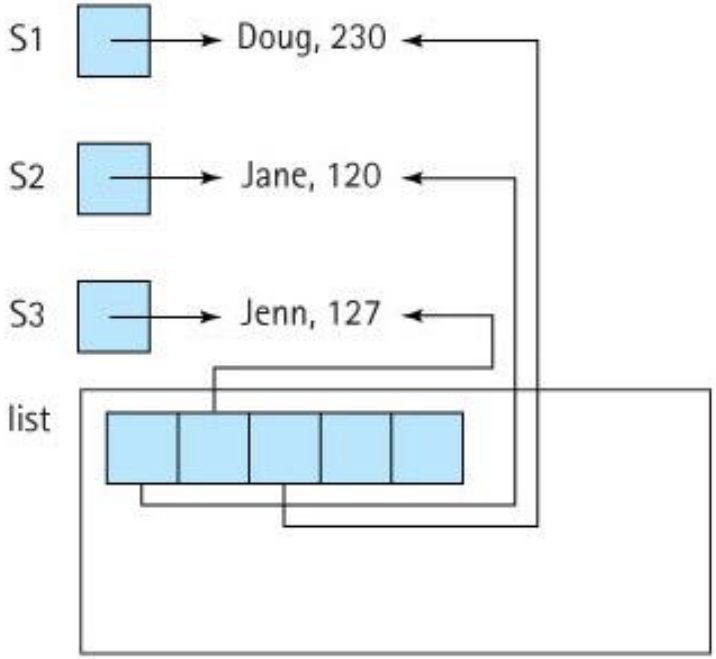


Example Step 2: Add Objects to List

By Copy Approach

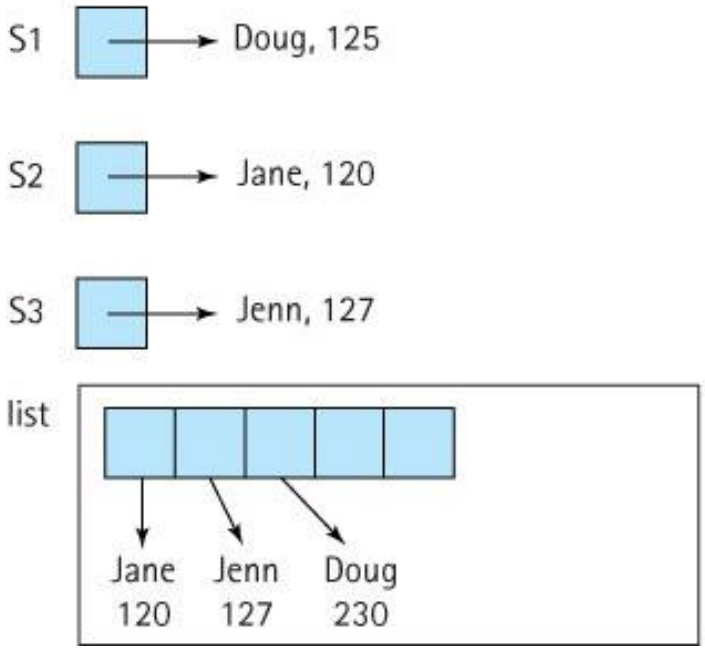


By Reference Approach



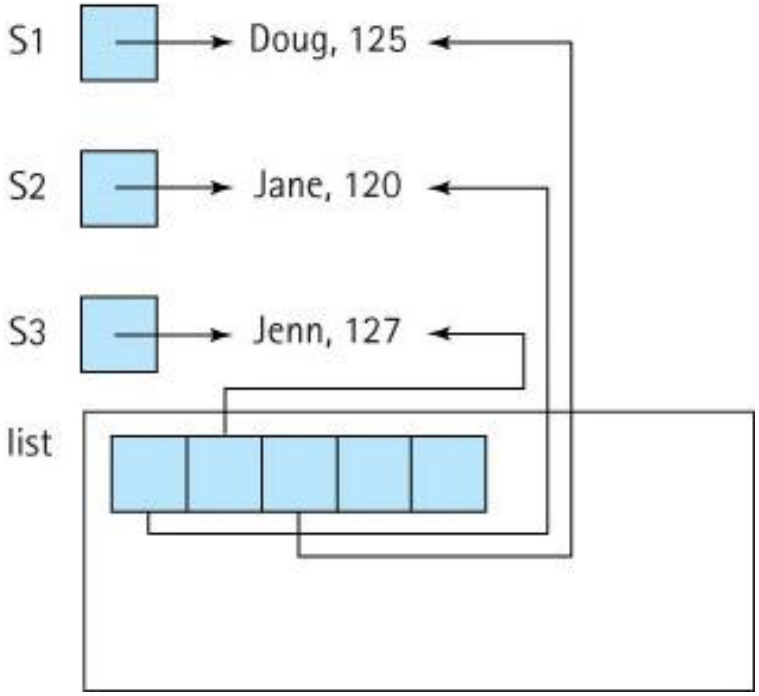
Example Step 3: S1.diet(-105)

By Copy Approach



Problem: List copy is out of date

By Reference Approach



Problem: List is no longer sorted

Which approach is better?

- It depends!

- If processing time and space ARE important
  - The “by reference” approach is probably best
  - We expect on the application to behave properly

- If processing time and space are NOT important , but we are concerned with maintaining careful control over the access to and integrity of our lists
  - The “by copy” approach is probably best
  - (maybe our list objects are not too large)

- It depends on what the list is used for!

The `ArrayIndexedList` Class

- Implements the `IndexedListInterface`

- Extends the `ArrayUnsortedList` class with the methods needed for creation and use of an indexed list:
  - `void add(int index, T element)`
  - `void set(int index, T element)`
  - `T get(int index)`
  - `int indexOf(T element)`
  - `T remove(int index)`

  The implementations are straightforward

- Overrides the `toString` method of the `ArrayUnsortedList` class

6.5 Applications: Poker, Golf, and Music

- We look at three applications, to see how our list implementations can be used to help solve problems.
  - Poker: Our program simulates poker hands to help verify formal analysis
  - Golf: Rank players based on their scores (demonstrates use of the Comparable interface)
  - Music: Organize a collection of songs

Code and Demo

- Code Walkthrough
- Note how each problem is solved using the most suitable type of list.

Homework

- Consider a function removeAll – removes all elements from the list that are equal to the argument element and returns int indicating how many elements were removed. Design a method to be added to the RefUnsortedList class that implements the operation. Code and test your method.

- Textbook question 47 a-d, 48

- Benchmark the Reference List sorting algorithm. Test it for 10, 100, 1000, 10000 and 100000 random numbers ranging from 1 to the size of the list. Do this three times and report the average algorithm execution time.
  - For 2 points extra credit output the average number of comparisons for each trio of insertion sorts!