CS 304 Lecture 6

The `List` ADT

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

Northeastern Illinois University

Chicago, Illinois, 60625

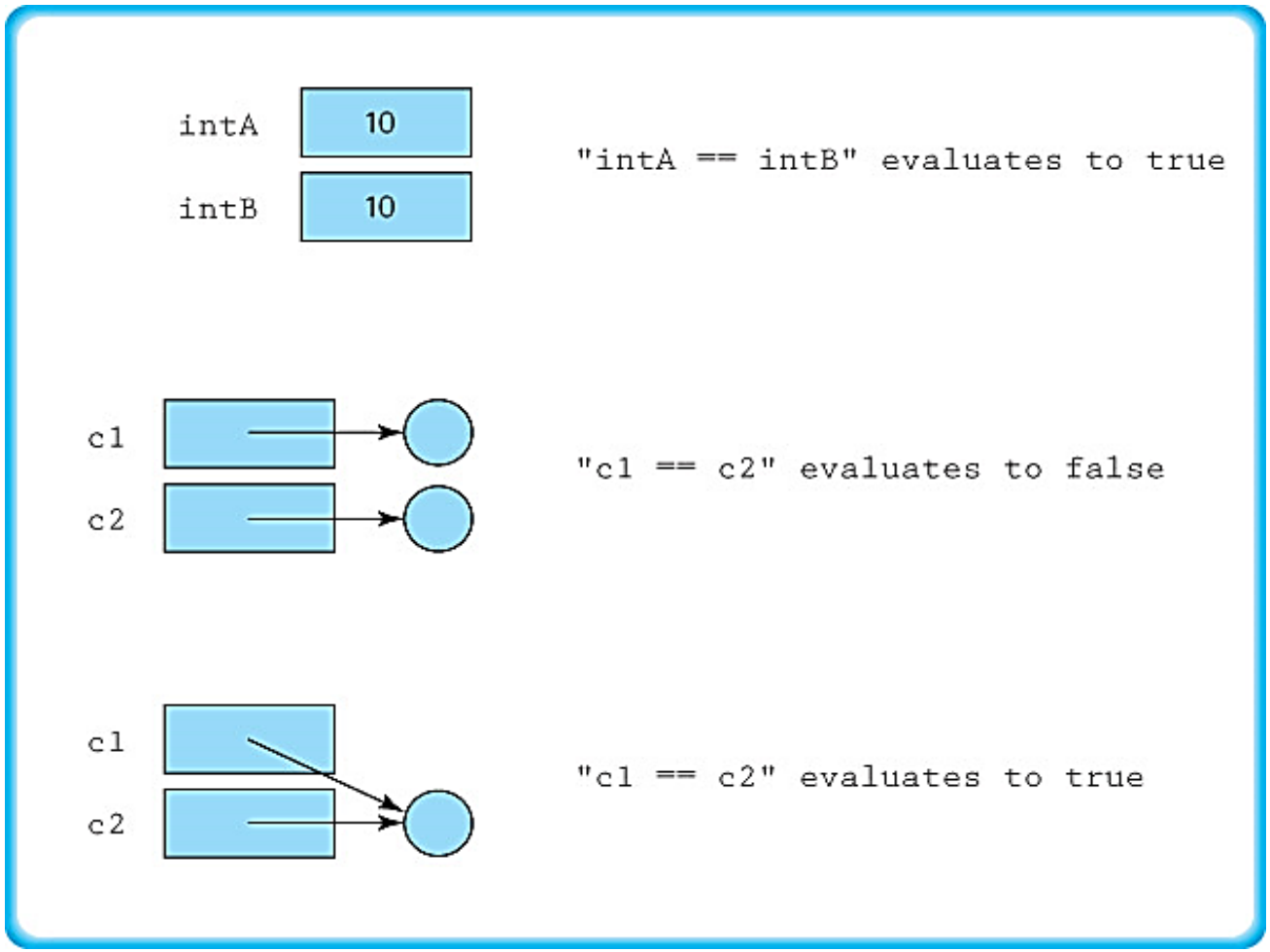
4 October 2016

The `List` ADT

- So far we have covered several abstract data types: **`StringLog`**, **`Stack`**, and **`Queue`**. We have been using arrays and linked lists to implement these ADTs.
- Now we want to discuss the implementations for the **`List`** ADT.
- First of all, let's think about the object comparisons as 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.

Object comparisons

- We learned that you can use the comparison operator (==) to compare two objects.

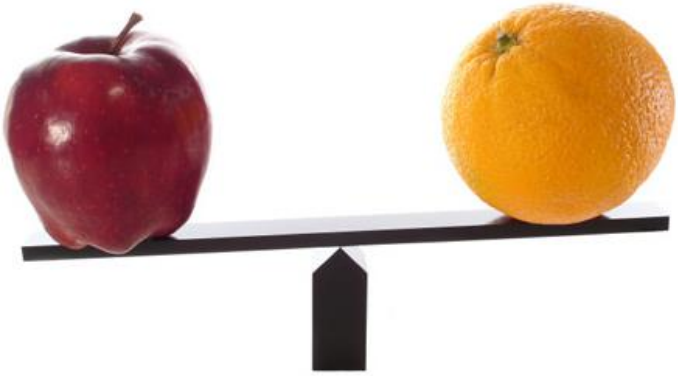


Using the `equals` method

- Since `equals` is exported from the `Object` class, it can be used with objects of any Java class.
- For example, 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:

```
public boolean equals(Circle c)
  // Precondition: c != null
  //
  // Returns true if both circles have the same radius,
  // otherwise returns false.
  {
    if (this.radius == c.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`**, which can be used to ensure that a class provides this functionality. The **`Comparable`** interface consists of exactly one abstract method:

```
public int compareTo(T o);
```

It 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

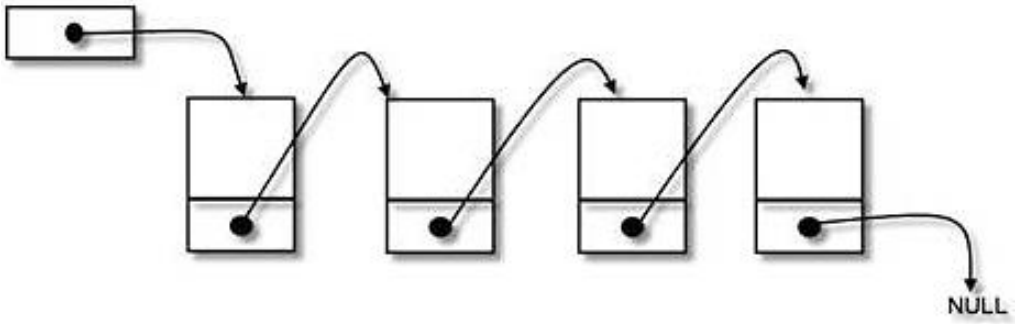
```
public int compareTo(Circle c)
// returns negative integer, zero, or a positive integer
// as this object is less than, equal to, or greater than
// the parameter object.
{
  if (this.radius < c.radius)
    return -1;
  else
    if (this.radius == c.radius)
      return 0;
    else
      return 1;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

List implementation

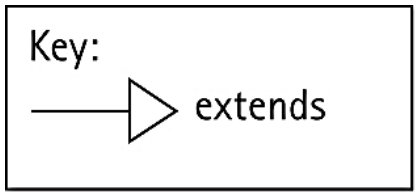
- The lists in the textbook are *unbounded*.
- Duplicate elements are allowed on the lists.
- They do *not* support `null` elements.
- The sorted lists are sorted in *increasing* order, as defined by the `compareTo` operation applied to list objects.
- In the indexed lists, the indices in use at any given time are *contiguous*, starting at 0.

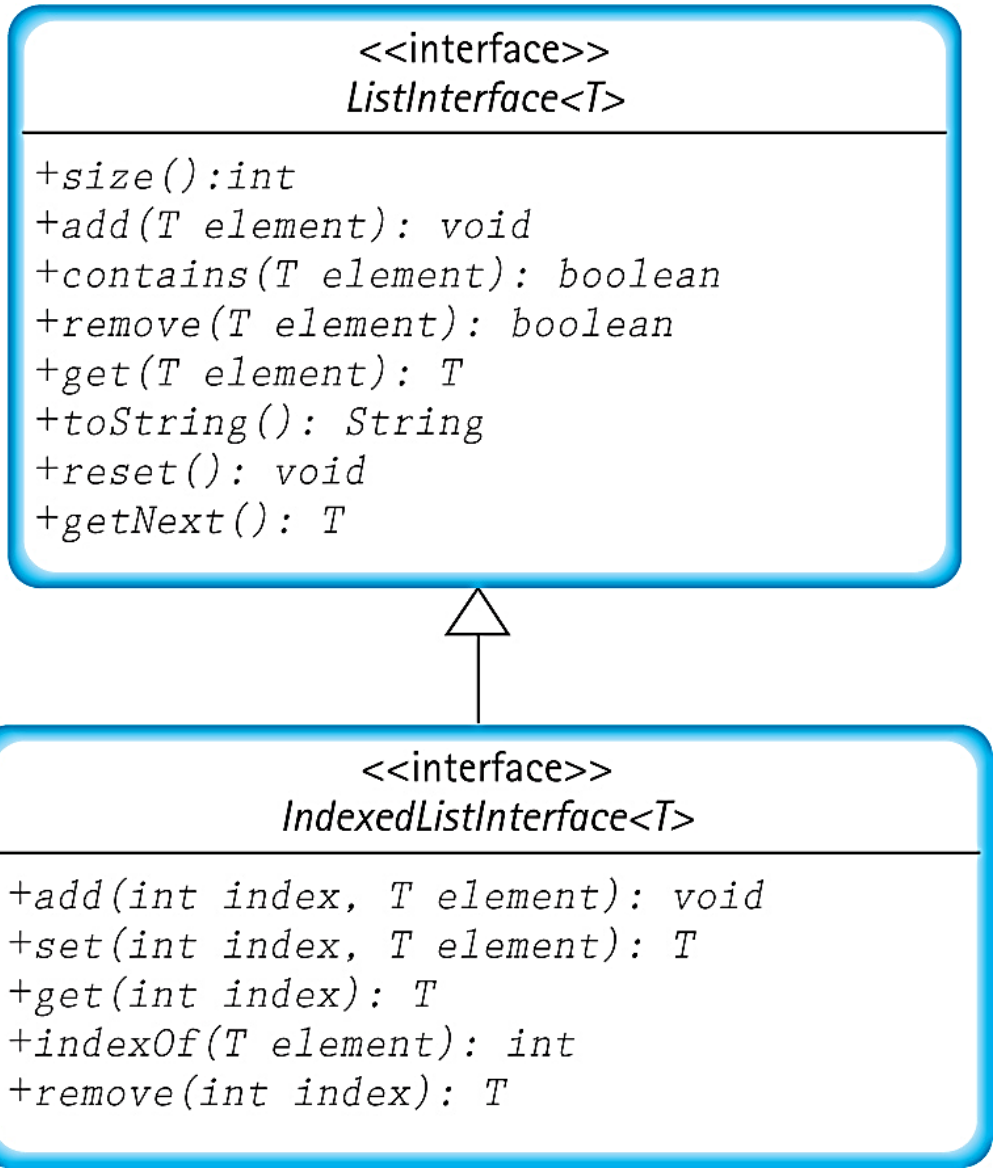
List iteration

- 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 the textbook's list variations provides the operations **`reset`** and **`getNext`** to support this activity.

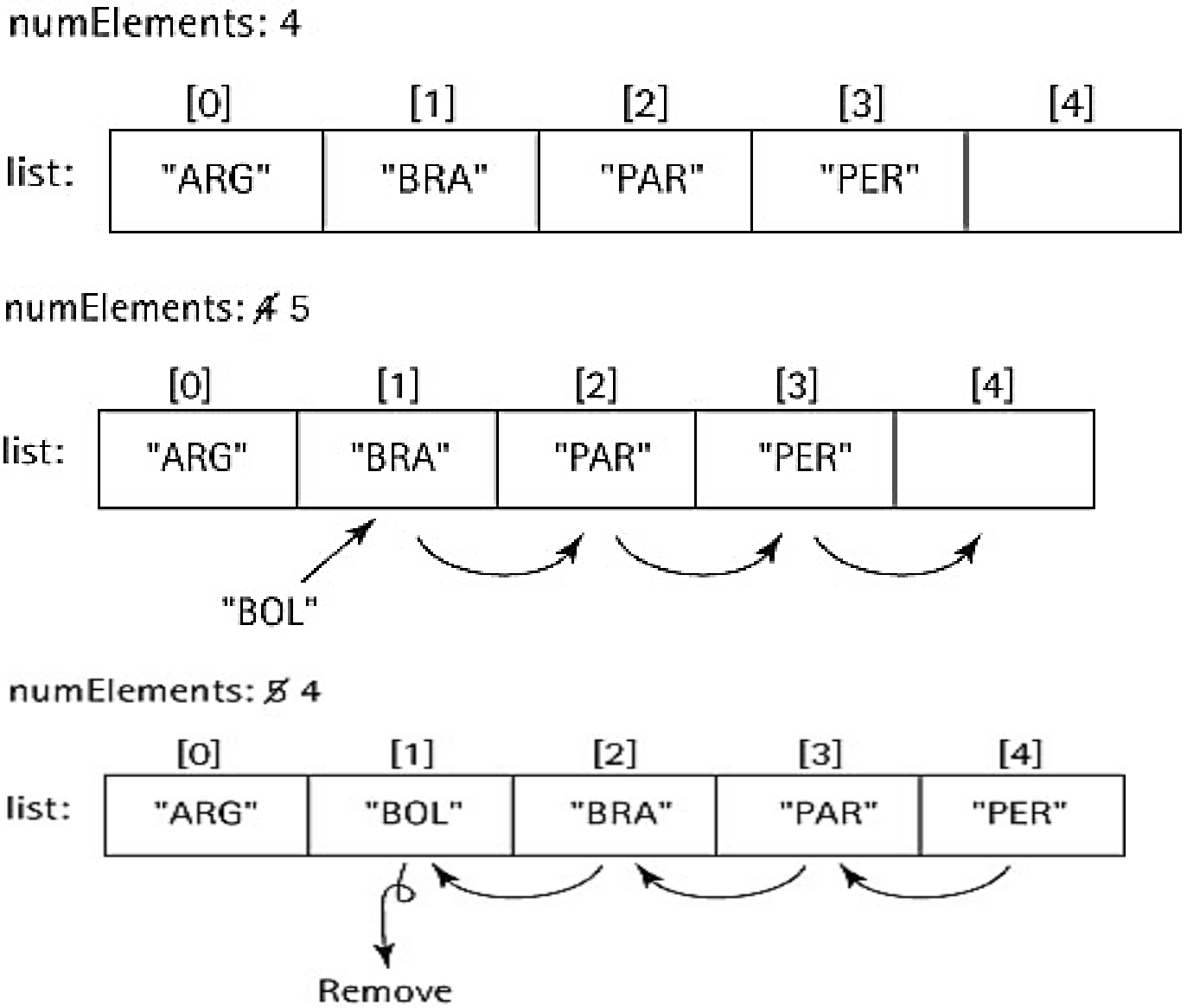


List operations



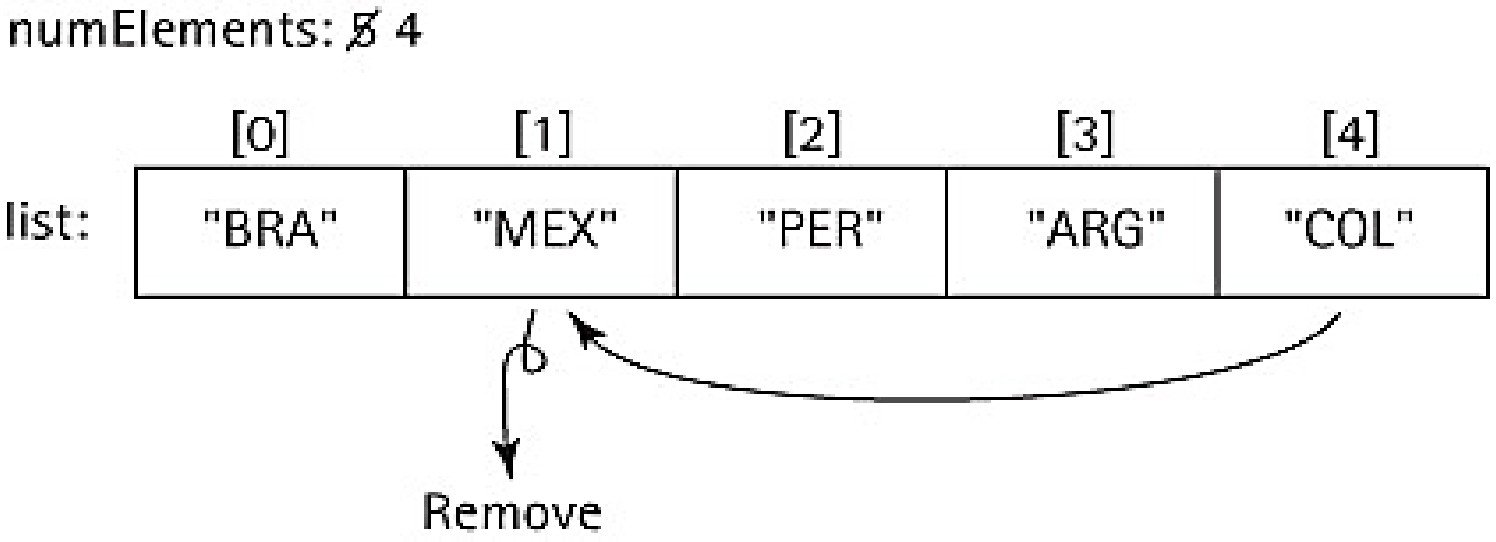


Array-based implementation



The `ArrayUnsortedList` Class

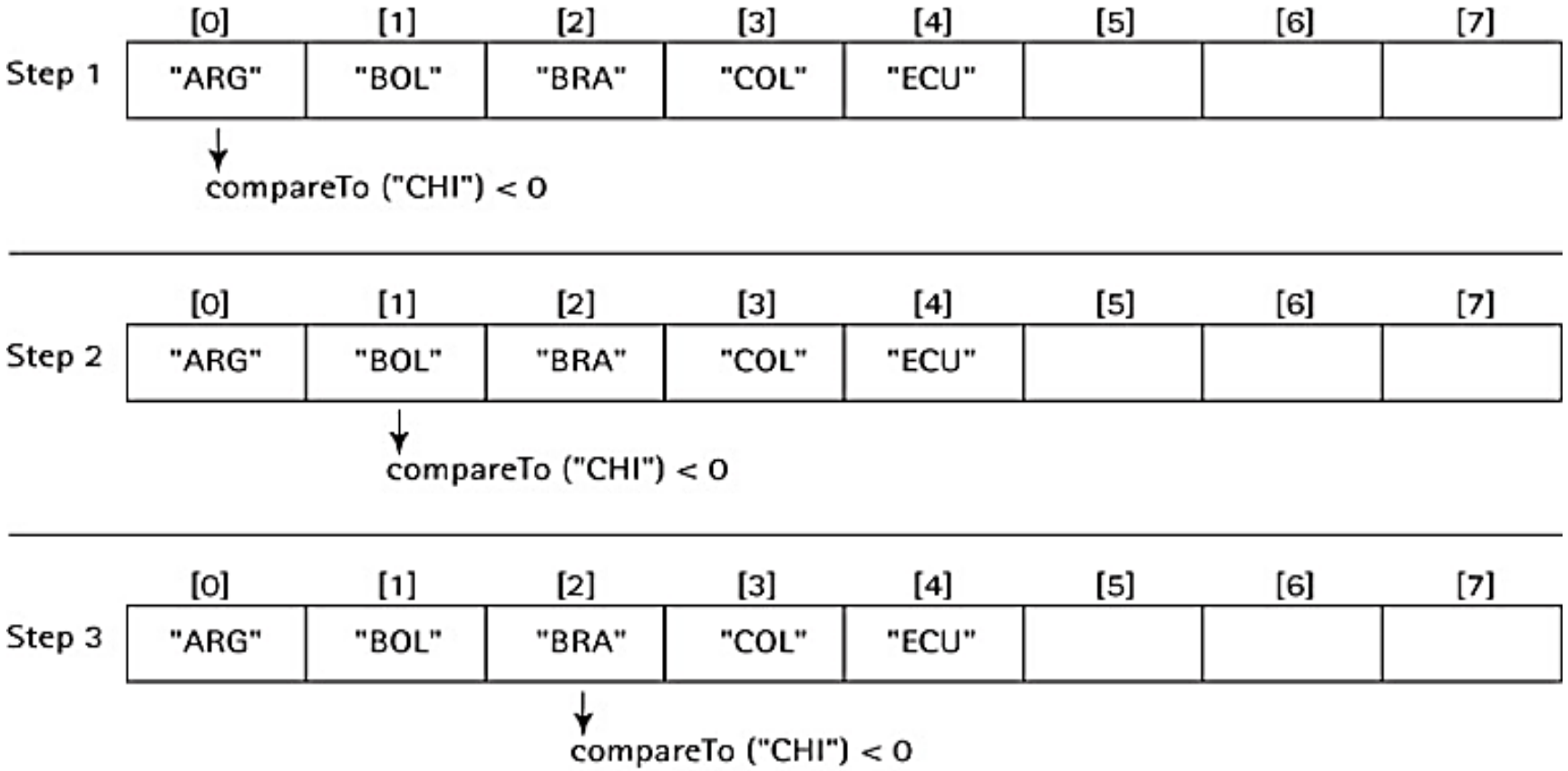
- Implements `ListInterface`.
- We create a helper method `find` for array search, which is used by several other methods.
  - `find` sets the `location` and `found` instance variables.
- In the unsorted array based list, we are not concerned with the order in which elements are stored. So, for example,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



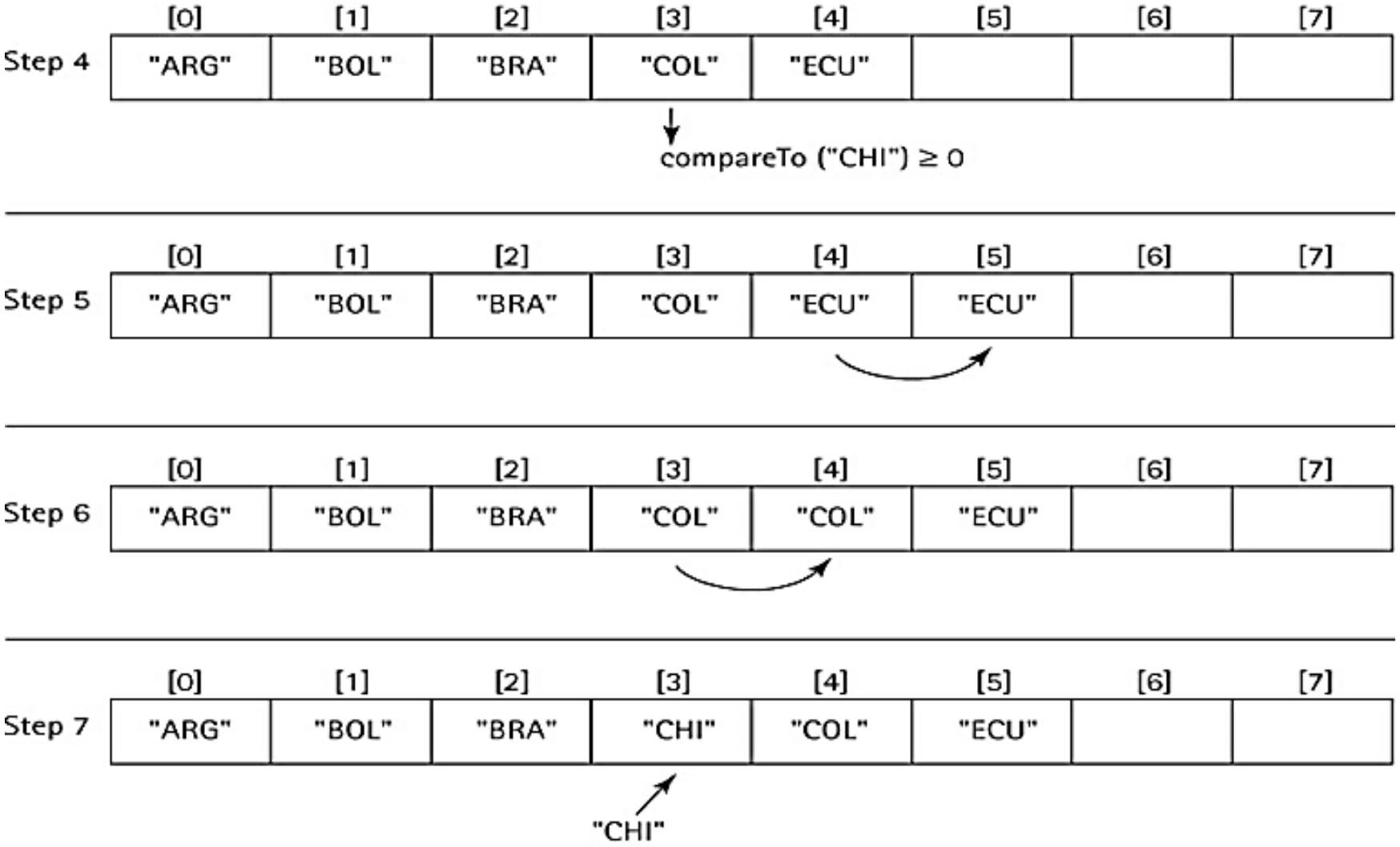
The `ArraySortedList` Class

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

The `ArraySortedList` Class

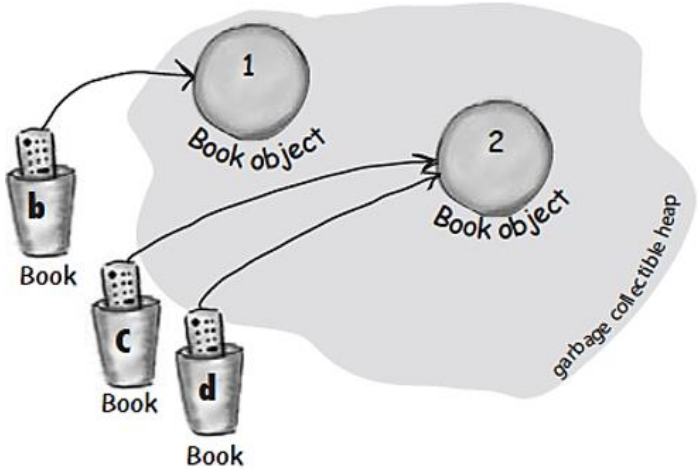


The `ArraySortedList` Class



Implementing ADTs "by copy" or "by reference"

- When designing an ADT we have a choice about how to handle the elements—"by copy" or "by reference."
  - **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. This is the most commonly used approach and is the approach we use in our course.

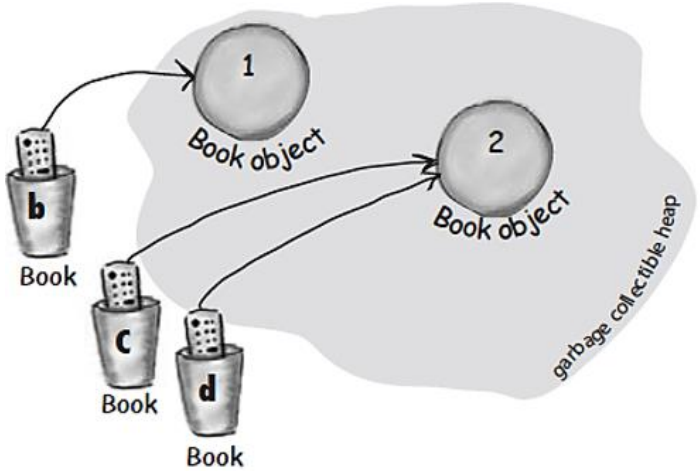


Implementing ADTs "by copy" or "by reference"

- By copy
  - Use object's `clone` method to make copies. However, you have to implement the `Cloneable` interface in advance.
  - Drawbacks:
    - Copy of object might not reflect up-to-date status of original object
    - Copying objects takes time, especially with deep-copying methods.
    - Storing extra copies of objects also requires extra memory.

Implementing ADTs "by copy" or "by reference"

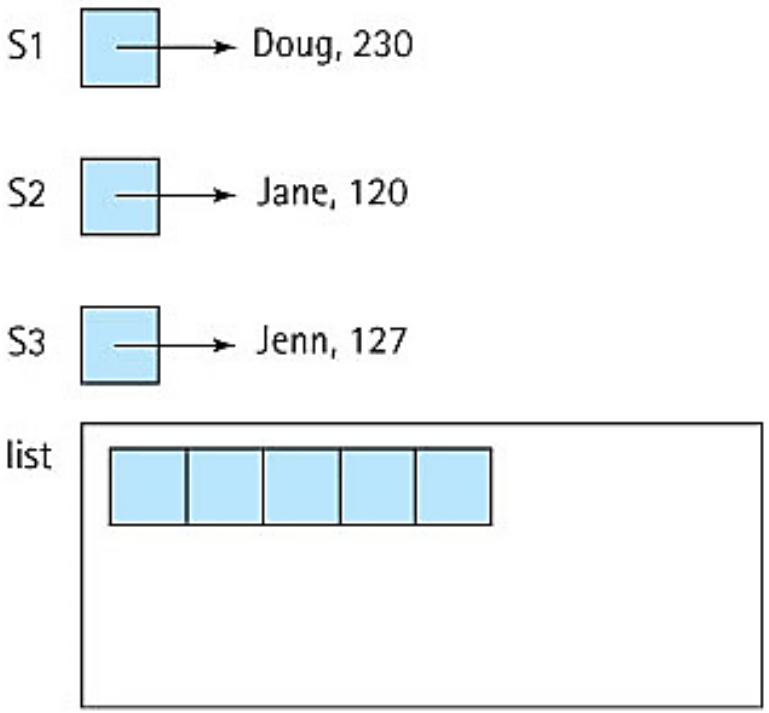
- By reference
  - The contents of the collection ADT are exposed to the client program so it has direct access to the individual elements in the collection.
  - Drawbacks:
    - Objects are accessed through aliases so the client program could accidently change an attribute of the objects. This might in turn cause problems. For example if it changes the key value for an element stored in a sorted list, then the order of the list is broken.



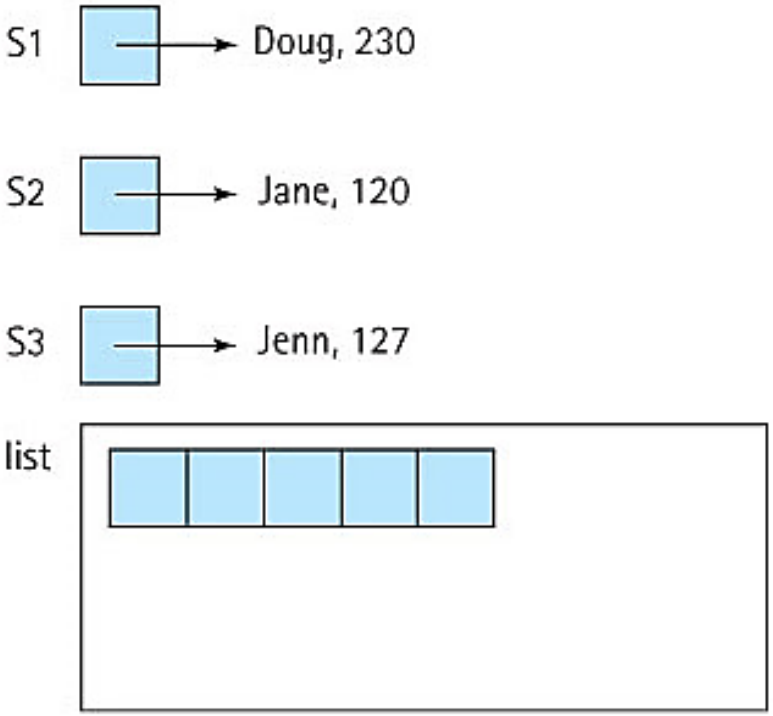
Implementing ADTs "by copy" or "by reference"

- An example – adding elements (name, weight) into lists

By copy approach



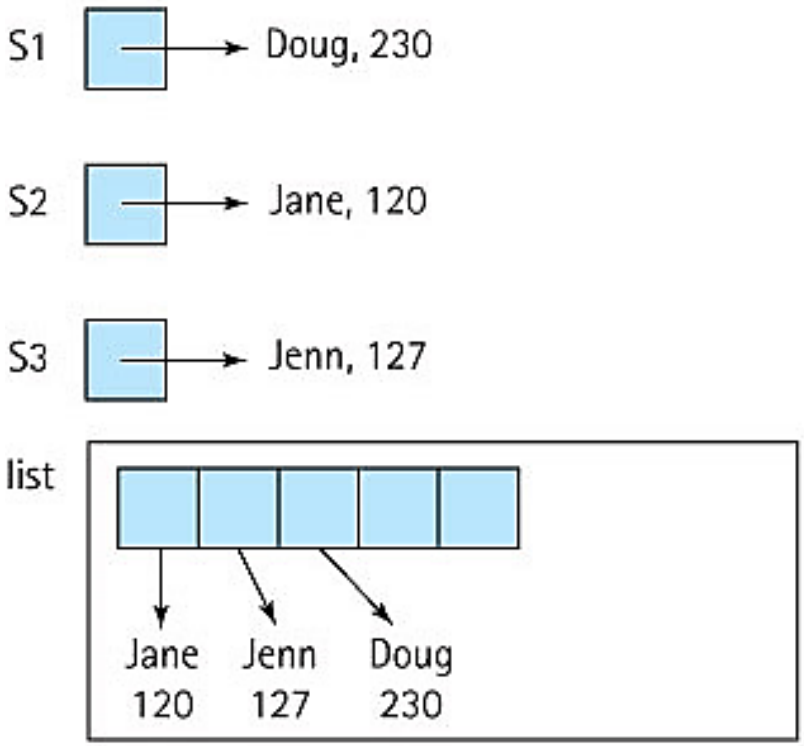
By reference approach



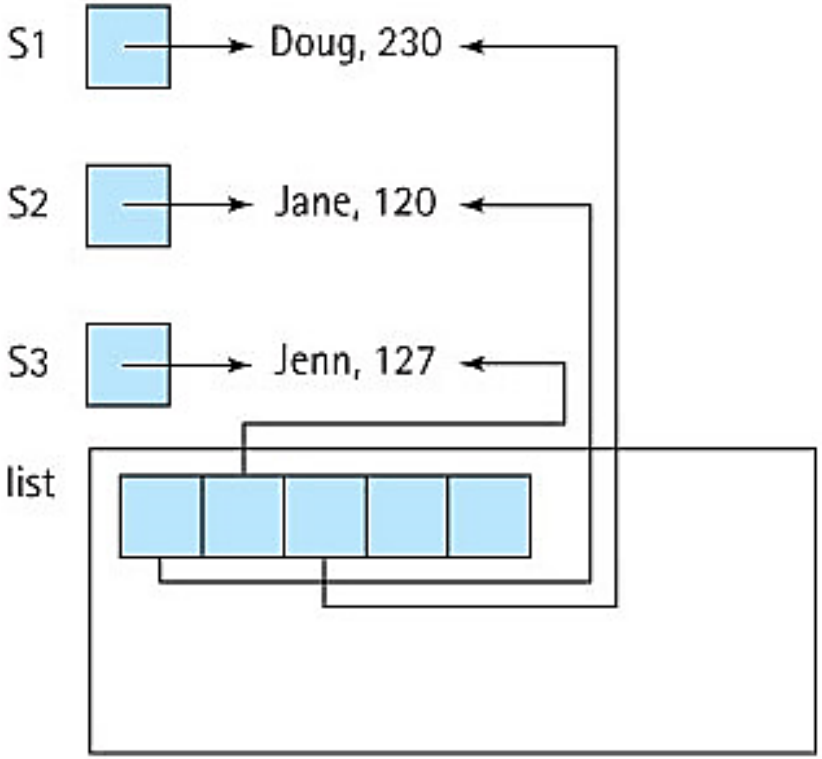
Implementing ADTs "by copy" or "by reference"

- An example – adding elements (name, weight) into lists

By copy approach



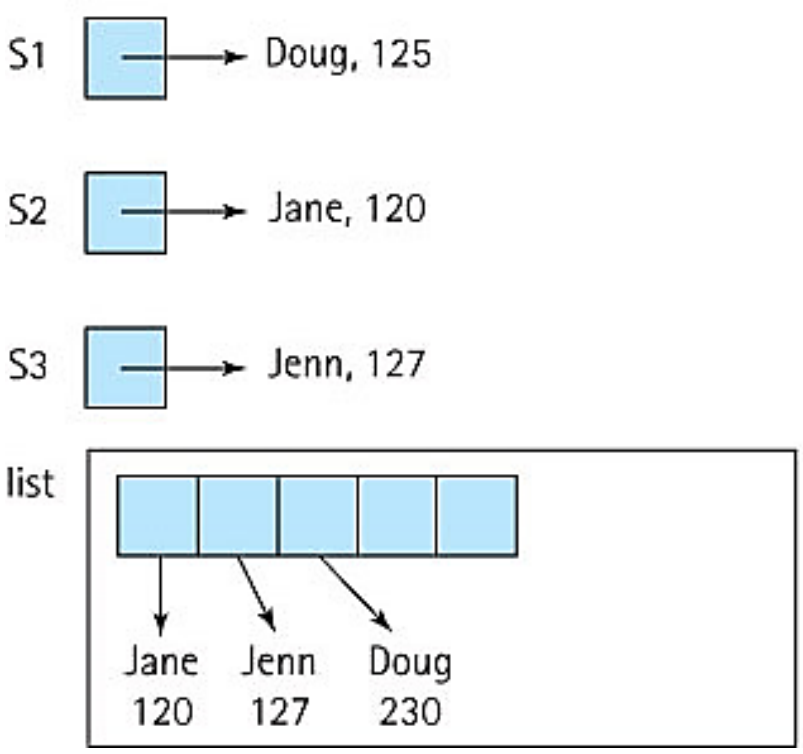
By reference approach



Implementing ADTs "by copy" or "by reference"

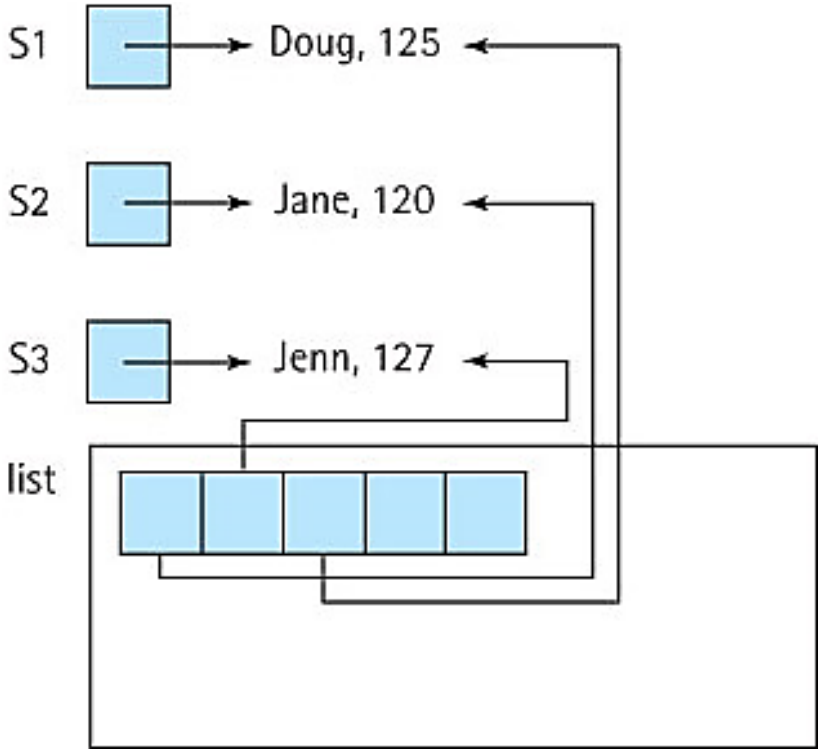
- Call `S1.diet(-105);`

By copy approach



Problem: List copy is out of date

By reference approach



Problem: List is no longer sorted

Implementing ADTs "by copy" or "by reference"

- So which one is better? **By copy** or **by reference**?
- If processing time and space are issues, and if we are comfortable counting on the application programs to behave properly, then the "*by reference*" approach is probably best.
- If we are not too concerned about time and space (maybe our list objects are not too large), but we are concerned with maintaining careful control over the access to and integrity of our lists, then the "*by copy*" approach is probably best.
- The suitability of either approach depends on what the list is used for.

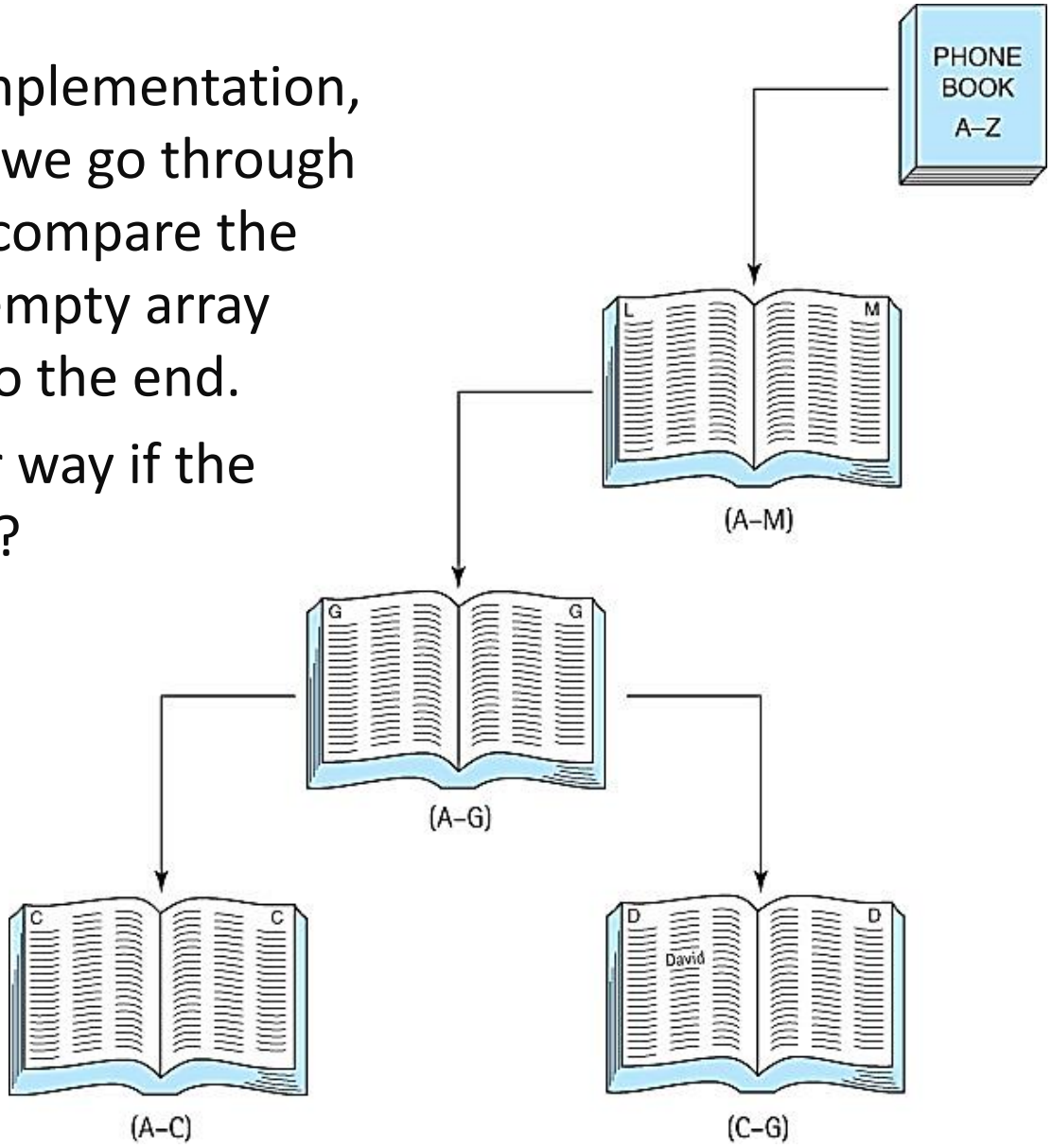
The `ArrayIndexedList` Class

- Implements `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)`
- Overrides the `toString` method of the `ArrayUnsortedList` class.

The binary search algorithm

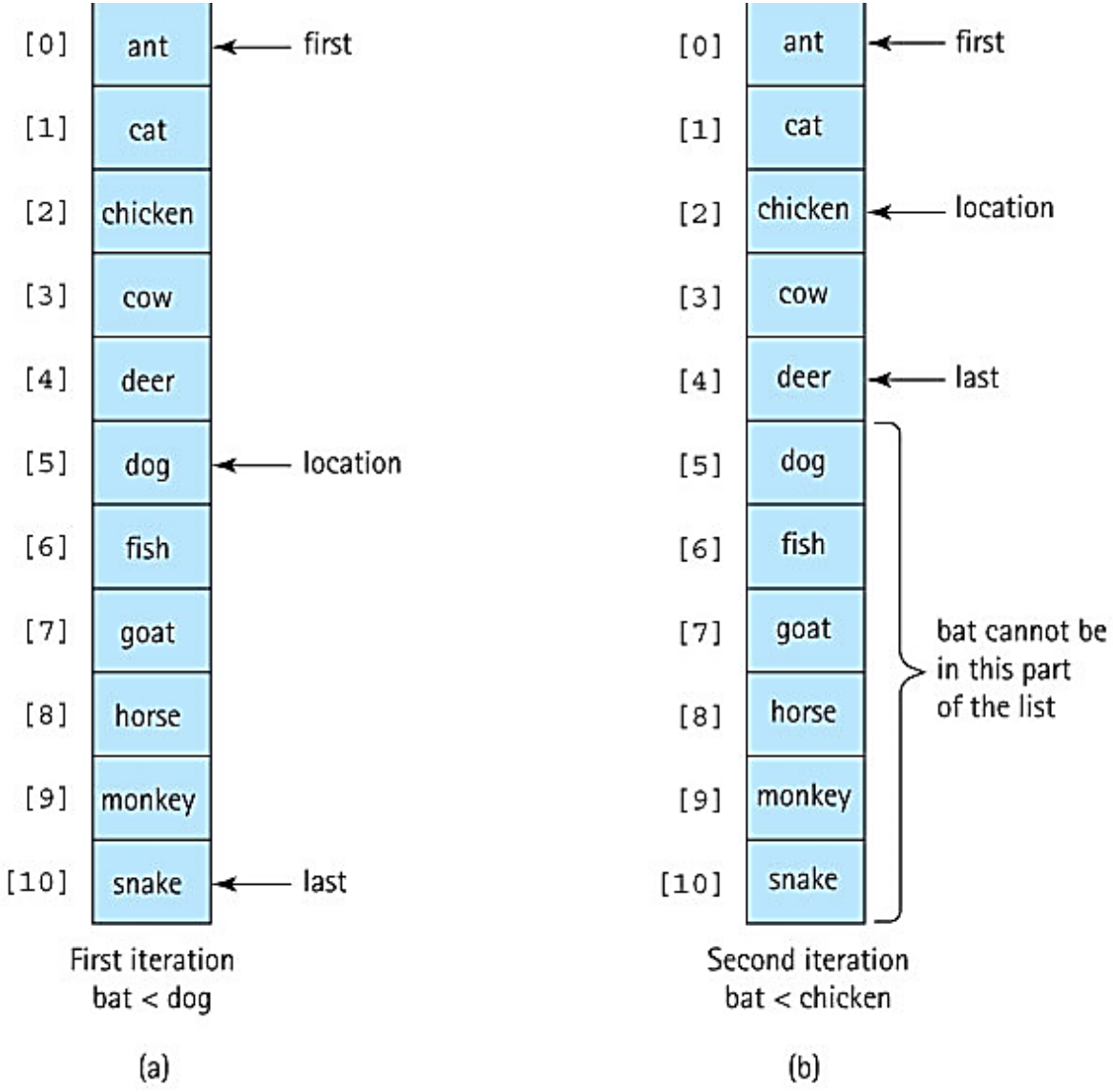
- In the array based list implementation, to search for an element, we go through the underlying array and compare the element with every non-empty array slot, from the beginning to the end.
- Would there be a better way if the underlying array is sorted?

Use binary search!



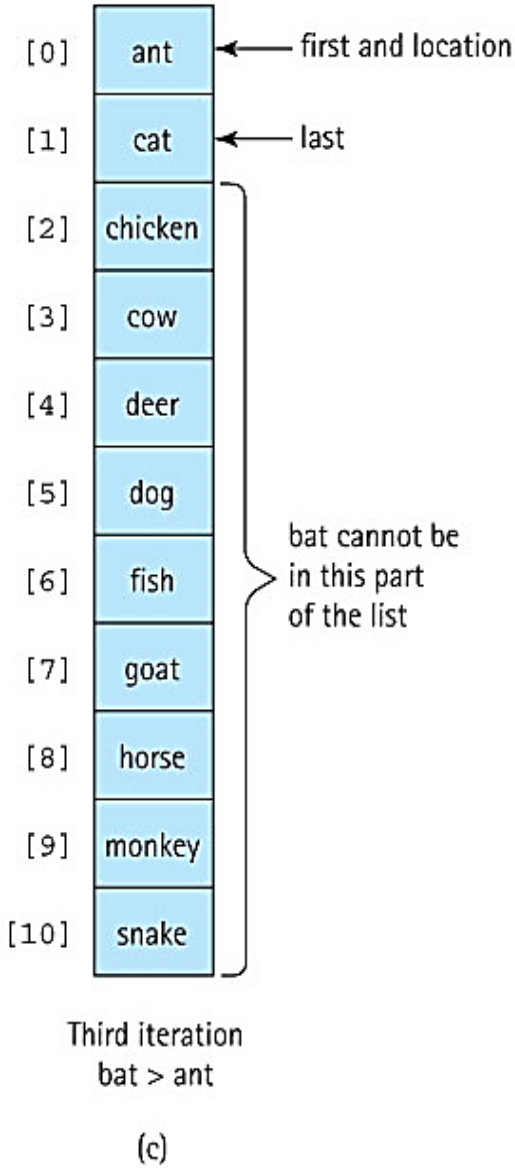
The binary search algorithm

Searching for "bat" in a sorted array



The binary search algorithm

Searching for "bat" in a sorted array



The binary search algorithm

```
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

- An alternative solution is to do the binary search recursively.
  - 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).

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

Recursive binary search

```
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);
  }
}
```

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

Reference based implementation

- Same as the reference based stacks and queues, the **`LLNode`** class from the **`support`** package is used to provide the nodes.
  - The **`info`** attribute of a node contains the list element.
  - The **`link`** attribute contains a reference to the node holding the next list element.
- We need to maintain a variable, **`list`**, that references the first node on the list.

The `RefUnsortedList` Class

- Implements `ListInterface`.
- The `size`, `toString`, `reset`, and `getNext` methods are straightforward.
- Because the list is unsorted, and the 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

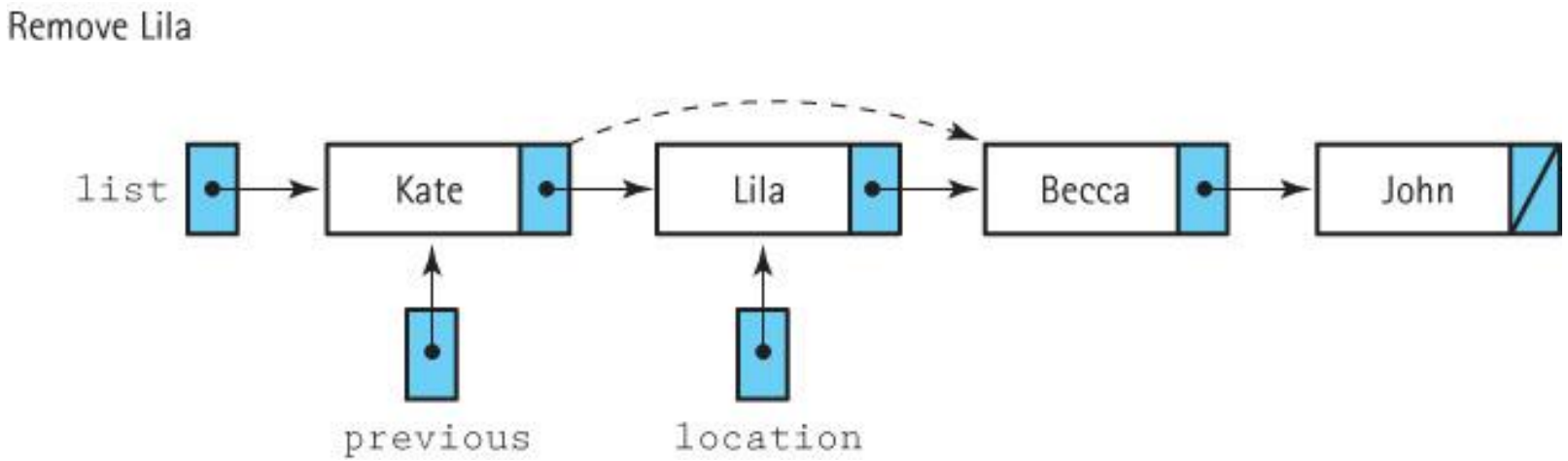
- Used by the **`contains`**, **`get`**, and **`remove`** methods.
- Makes **`contains`** and **`get`** straightforward.
- Sets a variable named **`previous`** - used by the **`remove`** method.

```
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();
    }
  }
}
```

The `remove` method

- To remove an element, we first find it using the `find` method, which sets the `location` instance variable of the `RefUnsortedList` class to indicate the target element and sets the `previous` instance 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 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 `ListInterface`.
- Extends `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.

The `RefSortedList` Class

- Find 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();
```

- 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.

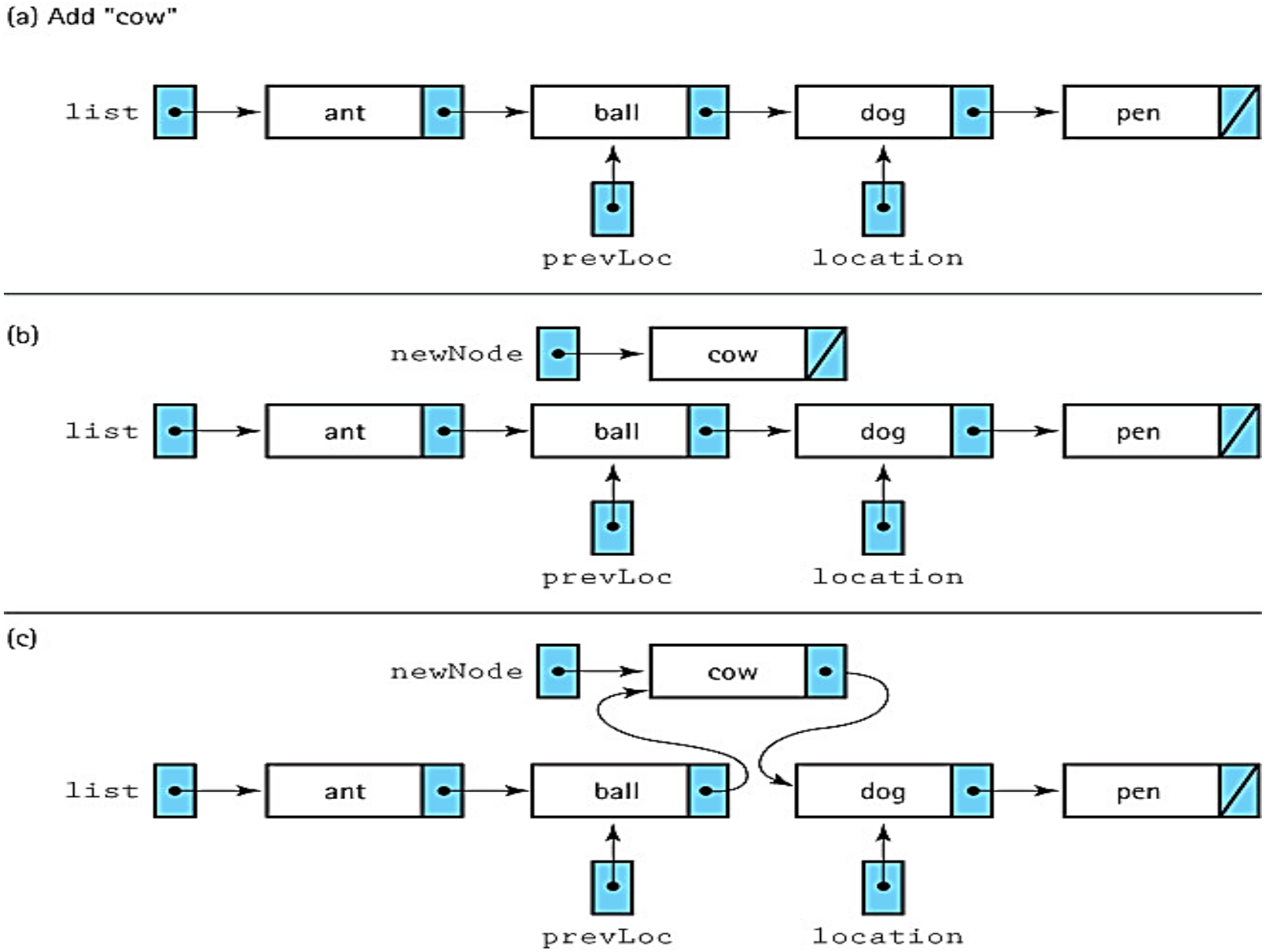
```
LLNode<T> newNode = new LLNode<T>(element);
```

The `RefSortedList` Class

- 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);
```

The `RefSortedList` Class

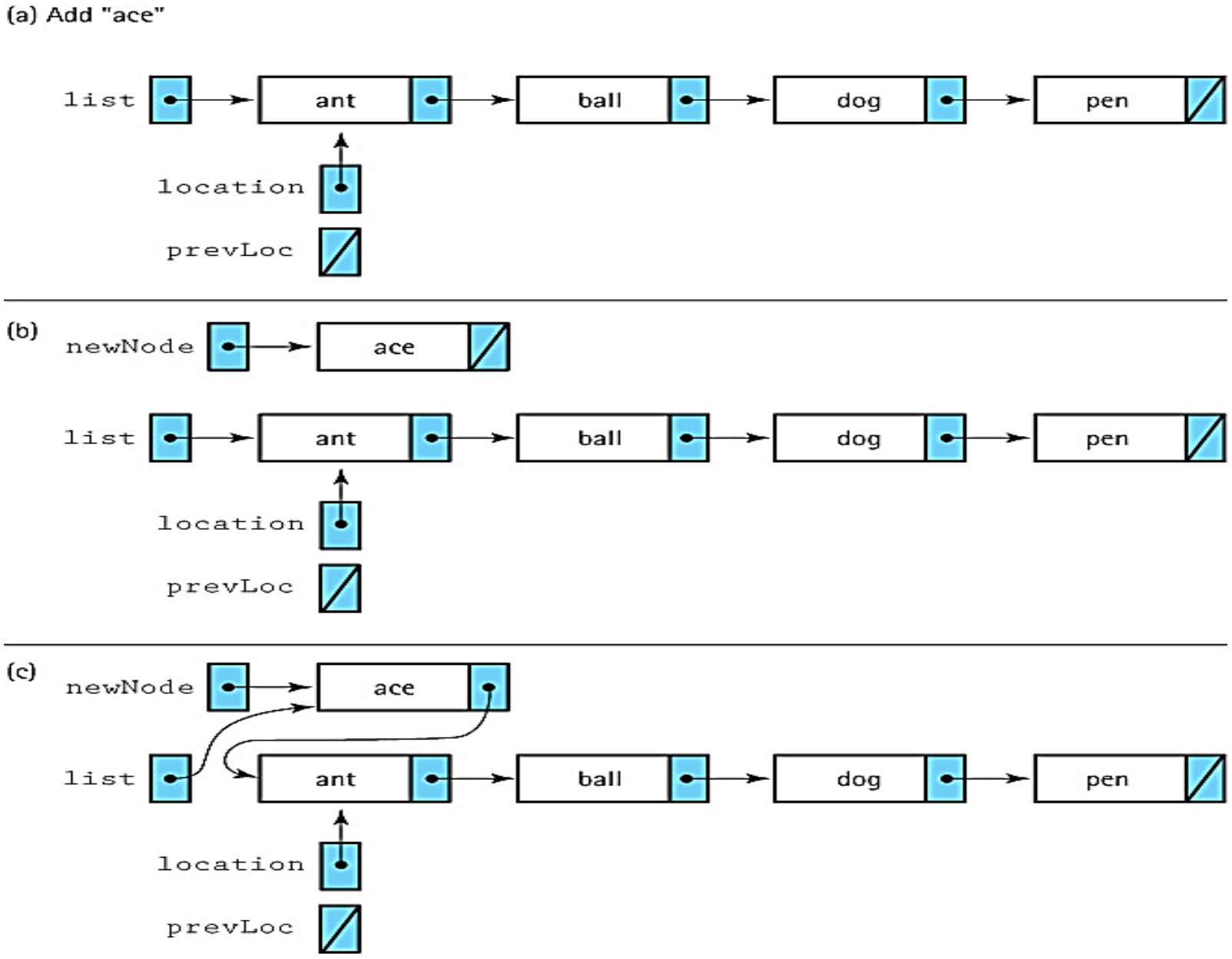


The `RefSortedList` Class

- 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:

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

The `RefSortedList` Class



Practice time

- Now, we are going to create a reference based sorted number list, with an integer as the value of each node.
- All the elements that are inserted into this list must be in ascending order.
- Removing elements should not break the order of the list.
- We need to print out the content of the list.
- As a summary, we have to implement the following:
  - creating a list from scratch
  - adding an element
  - removing an element
  - traversing the list

Action items

- Read book chapter 6.