

# Chapter 3

## Linked Structures

# Chapter Objectives

- Describe the use of references to create linked structures
- Compare linked structures to array-based structures
- Explore the techniques for managing a linked list
- Discuss the need for a separate node to form linked structures
- Implement a bag collection using a linked list

# References as Links

- There are many ways to implement a collection
- In chapter 2 we explored an array-based implementation of a bag collection
- *A linked structure* uses object reference variables to link one object to another
- Recall that an object reference variable stores the address of an object
- In that sense, an object reference is a *pointer* to an object

**figure 3.1** An object reference variable pointing to an object



# Self-Referential Objects

- A `Person` object, for instance, could contain a reference variable to another `Person` object:

```
public class Person
{
    private String name;
    private String address;

    private Person next; // a link to another Person
                        object

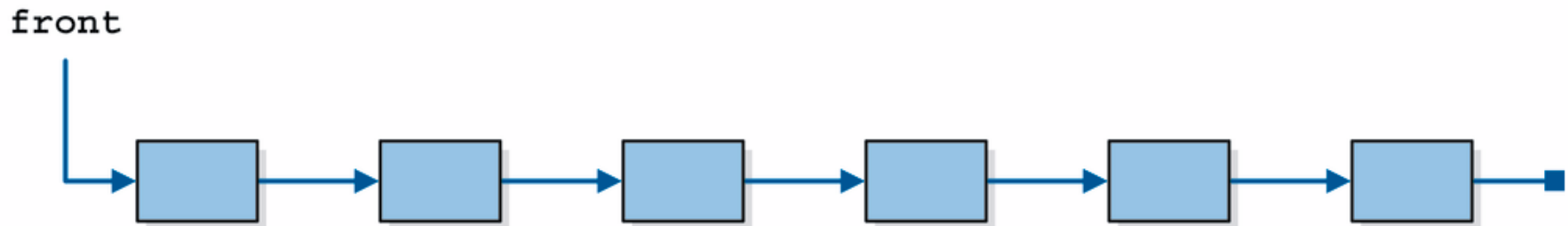
    // whatever else

}
```

# Linked Lists

- This type of reference can be used to form a *linked list*, in which one object refers to the next, which refers to the next, etc.
- Each object in a list is often generically called a *node*
- A linked list is a *dynamic* data structure in that its size grows and shrinks as needed, unlike an array, whose size is fixed
- Java objects are created dynamically when they are instantiated

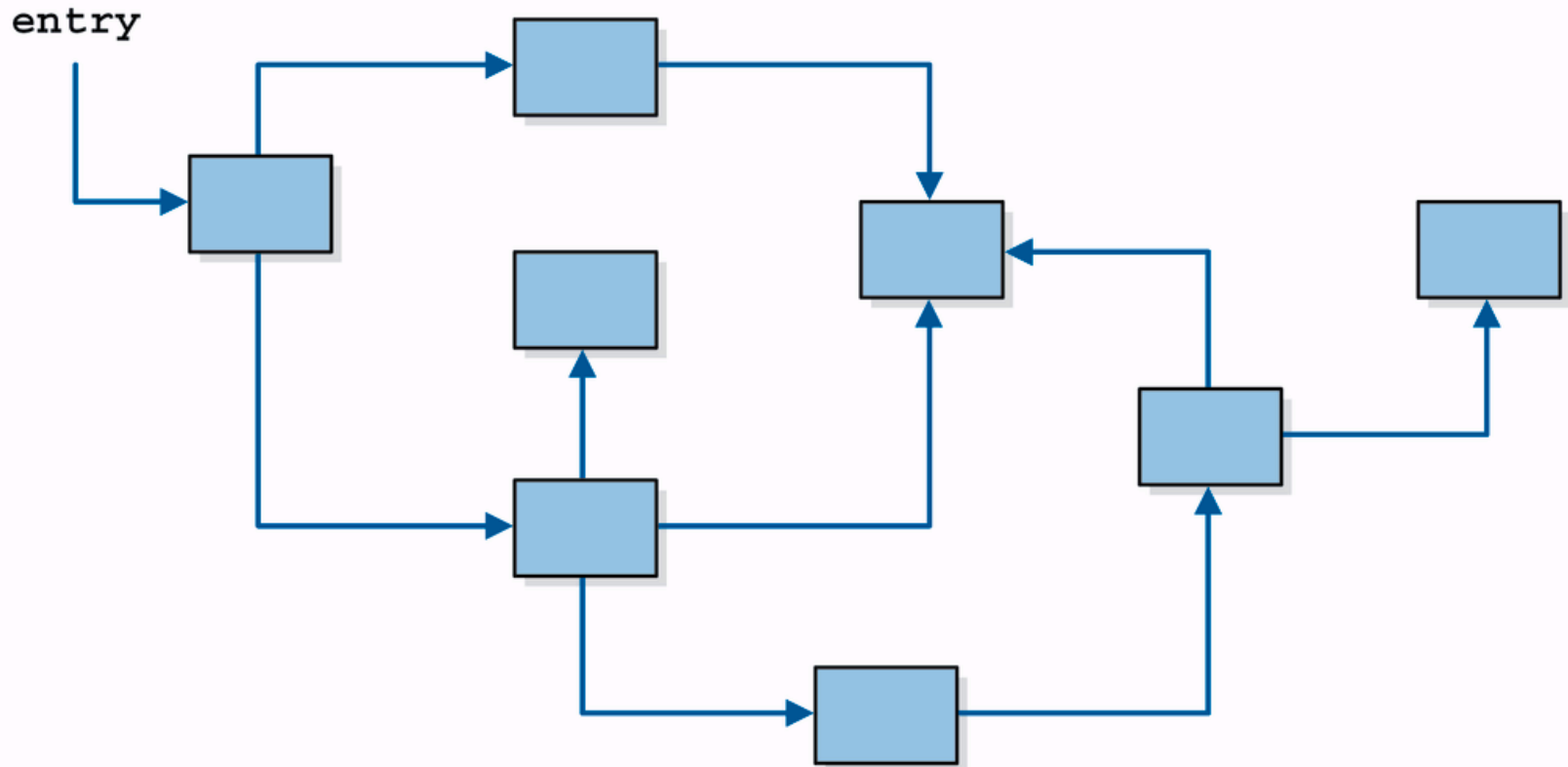
## figure 3.2 A linked list



# Non-linear Structures

- A linked list, as the name implies, is a linear structure
- Object references also allow us to create non-linear structures such as hierarchies and graphs

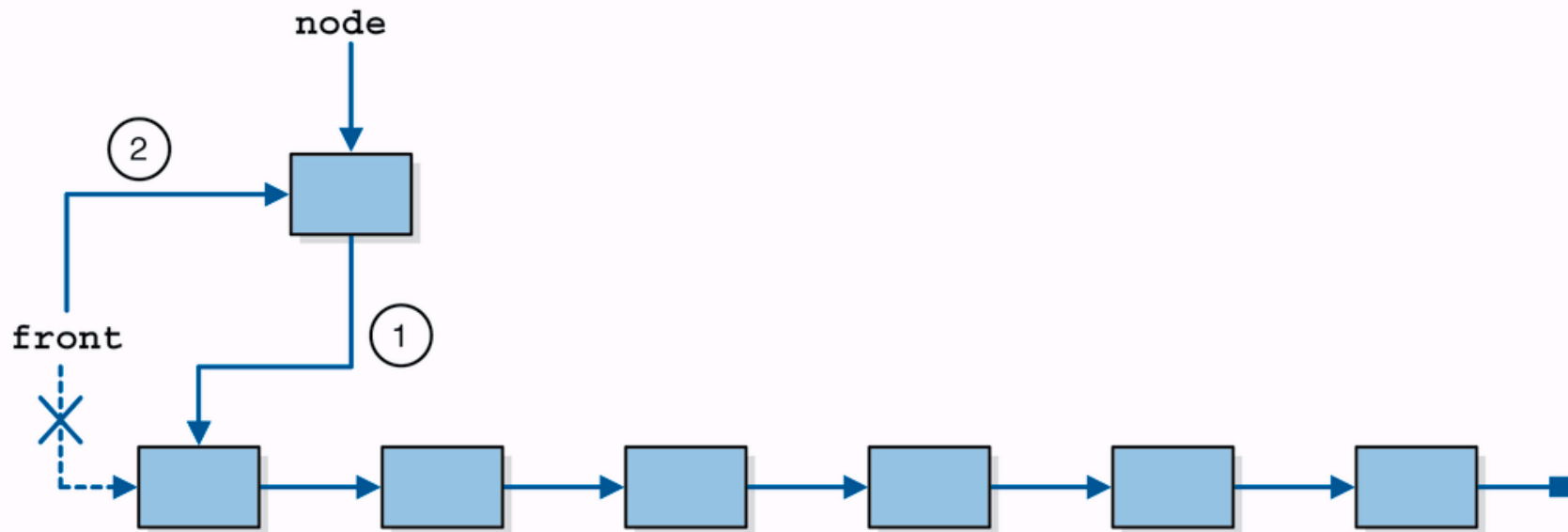
**figure 3.3** A complex linked structure



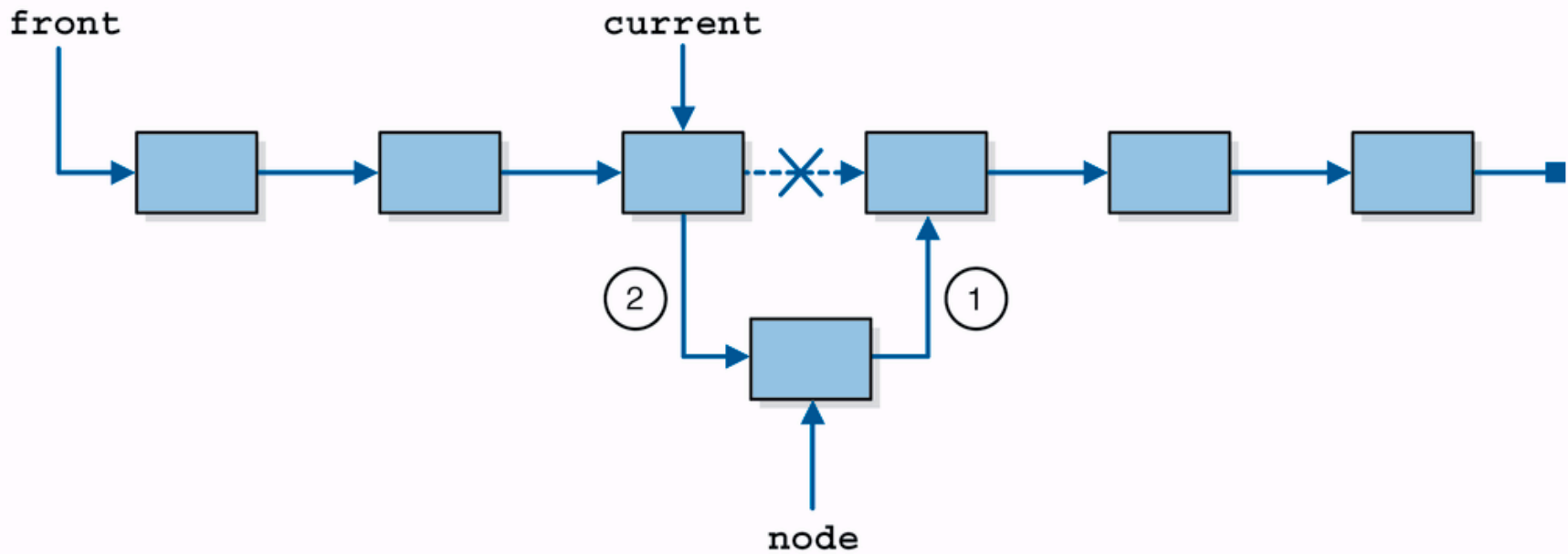
# Managing Linked Lists

- The references in a linked list must be carefully managed to maintain the integrity of the structure
- Special care must be taken to ensure that the entry point into the list is maintained properly
- The order in which certain steps are taken is important
- Consider inserting and deleting nodes in various positions within the list

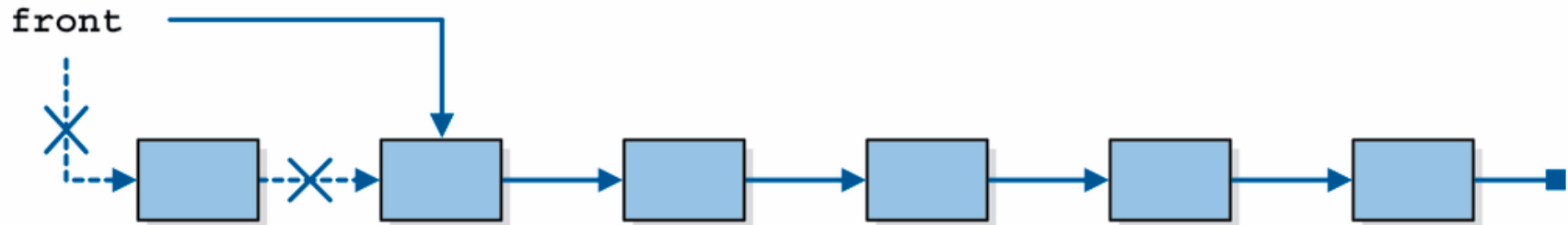
## figure 3.4 Inserting a node at the front of a linked list



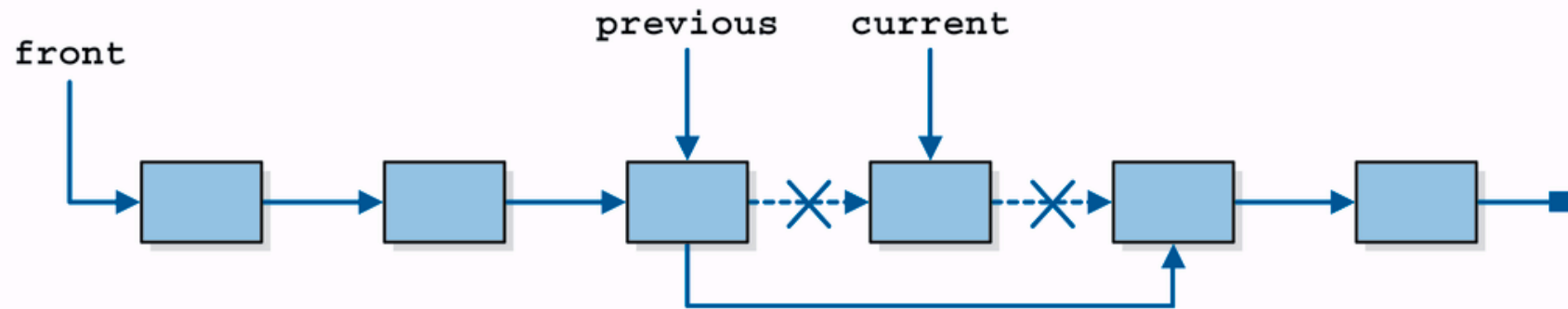
## figure 3.5 Inserting a node in the middle of a linked list



## figure 3.6 Deleting the first node in a linked list



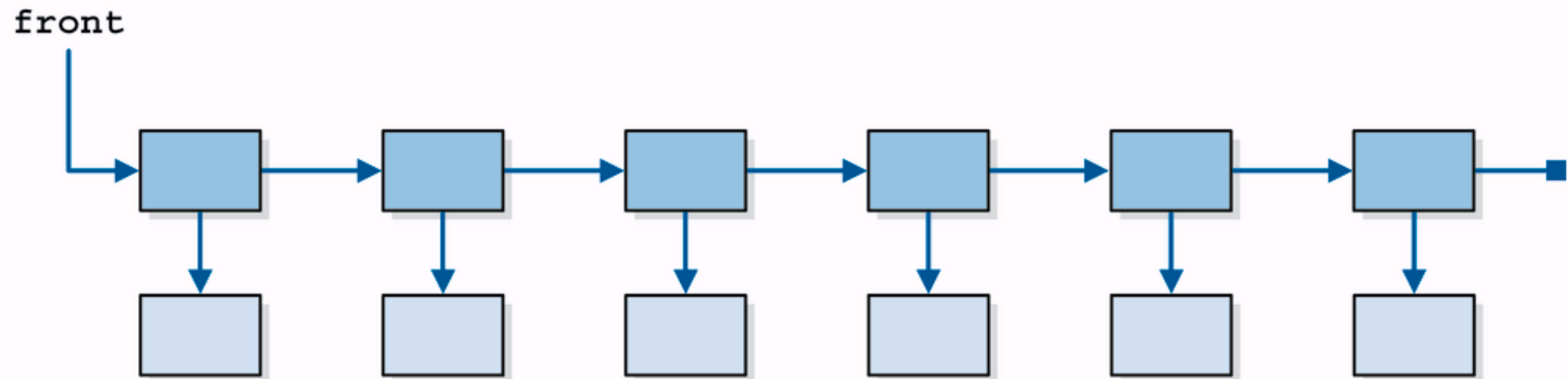
## figure 3.7 Deleting an interior node from a linked list



# Elements without Links

- The problem with self-referential objects is that they "know" they are part of a list
- A better approach is to manage a separate list of nodes that also reference the objects stored in the list
- The list is still managed using the same techniques
- The objects stored in the list need no special implementation to be part of the list
- A generic list collection can be used to store any kind of object

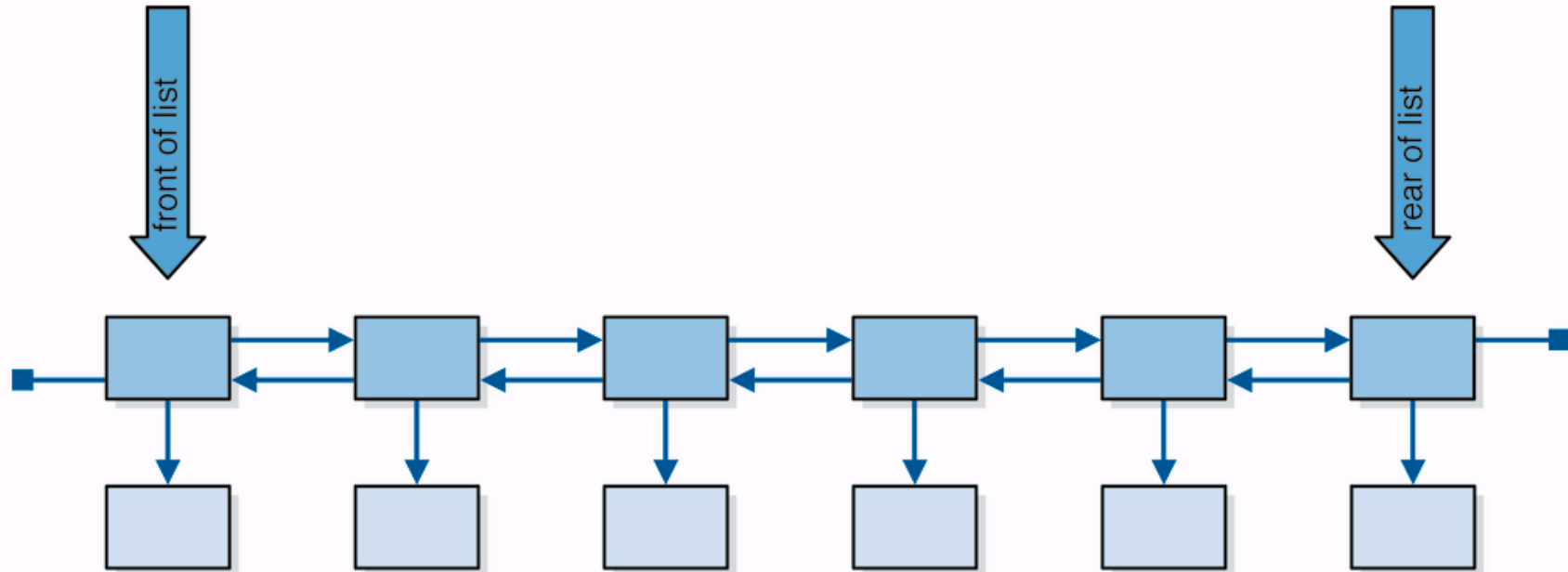
## figure 3.8 Using separate node objects to store and link elements



# Doubly Linked Lists

- There are variations on the implementation of linked lists that may be useful in particular situations
- For example, in a *doubly linked list* each node has a reference to both the next and previous nodes in the list
- This makes traversing the list easier

**figure 3.9** A doubly linked list



# Another Bag Implementation

- Let's explore a linked implementation of a bag collection
- The collection has the same purpose, and can be used to solve the same problems
- It will implement the same interface (`BagADT`) as the array-based implementation
- Only its underlying structure changes

# The `LinkedListBag` Class

- The elements of the bag are stored as nodes of the linked list
- We will maintain a `count` of the current number of elements in the bag
- Instead of declaring an array, we declare only a reference to the first element in the list (the `contents` of the bag)
- The nodes of the list are defined by the `LinearNode` class
- See [LinearNode.java](#) (page 78)

# LinearNode Class

```
public class LinearNode
{
    private LinearNode next;
    private Object element;

    //-----
    //  Creates an empty node.
    //-----
    public LinearNode()
    {
        next = null;
        element = null;
    }

    //-----
    //  Creates a node storing the specified element.
    //-----
    public LinearNode (Object elem)
    {
        next = null;
        element = elem;
    }
}
```

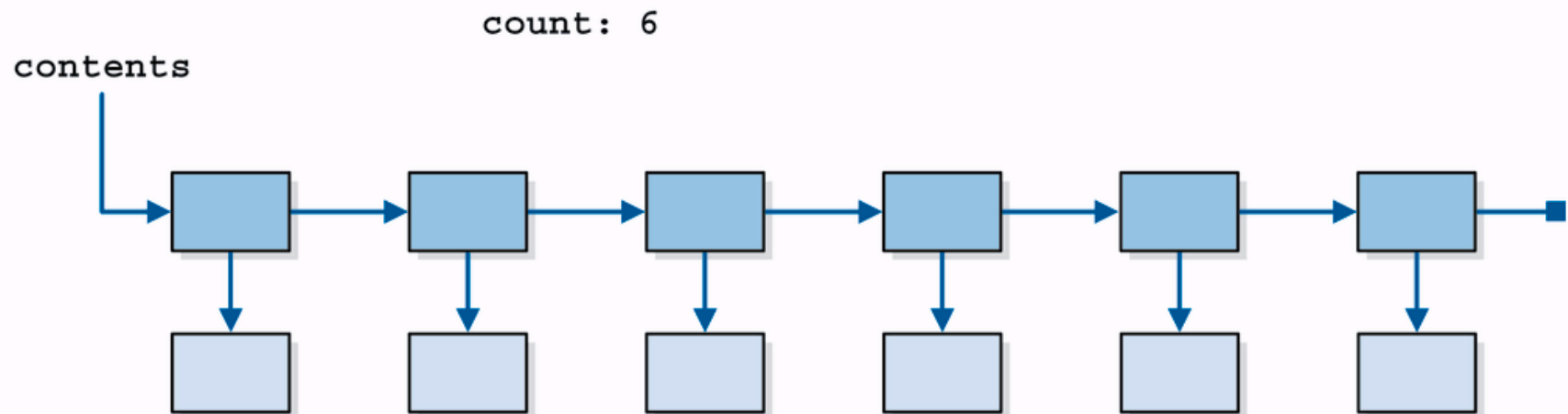
# LinearNode Class

```
//-----  
// Returns the node that follows this one.  
//-----  
public LinearNode getNext()  
{  
    return next;  
}  
  
//-----  
// Sets the node that follows this one.  
//-----  
public void setNext (LinearNode node)  
{  
    next = node;  
}
```

# LinearNode Class

```
//-----  
// Returns the element stored in this node.  
//-----  
public Object getElement()  
{  
    return element;  
}  
  
//-----  
// Sets the element stored in this node.  
//-----  
public void setElement (Object elem)  
{  
    element = elem;  
}  
}
```

# figure 3.10 A linked implementation of a bag collection



# LinkedList Constructor

```
//-----  
//  Creates an empty bag.  
//-----  
public LinkedList()  
{  
    count = 0;  
    contents = null;  
}
```

# The add Operation

```
//-----  
//  Adds the specified element to the bag.  
//-----  
public void add (Object element)  
{  
    LinearNode node = new LinearNode (element);  
    node.setNext(contents);  
    contents = node;  
    count++;  
}
```

# The removeRandom Operation

```
//-----  
// Removes a random element from the bag and returns it. Throws  
// an EmptyBagException if the bag is empty.  
//-----  
public Object removeRandom() throws EmptyBagException  
{  
    LinearNode previous, current;  
    Object result = null;  
  
    if (isEmpty())  
        throw new EmptyBagException();  
  
    int choice = rand.nextInt(count) + 1;  
  
    if (choice == 1)  
    {  
        result = contents.getElement();  
        contents = contents.getNext();  
    }  
}
```

# removeRandom continued

```
else
{
    previous = contents;
    for (int skip=2; skip < choice; skip++)
        previous = previous.getNext();
    current = previous.getNext();
    result = current.getElement();
    previous.setNext(current.getNext());
}

count--;

return result;
}
```

# The remove Operation

```
//-----  
// Removes one occurrence of the specified element from the bag  
// and returns it. Throws an EmptyBagException if the bag is  
// empty and a NoSuchElementException if the target is not in  
// the bag.  
//-----  
public Object remove (Object target) throws EmptyBagException,  
                                                             NoSuchElementException  
{  
    boolean found = false;  
    LinearNode previous, current;  
    Object result = null;  
  
    if (isEmpty())  
        throw new EmptyBagException();  
  
    if (contents.getElement().equals(target))  
    {  
        result = contents.getElement();  
        contents = contents.getNext();  
    }  
}
```

# remove continued

```
else
{
    previous = contents;
    current = contents.getNext();
    for (int look=0; look < count && !found; look++)
        if (current.getElement().equals(target)
            found = true;
        else
        {
            previous = current;
            current = current.getNext();
        }

    if (!found)
        throw new NoSuchElementException();
    result = current.getElement();
    previous.setNext(current.getNext());
}

count--;
return result;
}
```

# The `iterator` Operation

```
//-----  
// Returns an iterator for the elements currently in this bag.  
//-----  
public Iterator iterator()  
{  
    return new LinkedIterator (contents, count);  
}
```

See [LinkedIterator.java](#) (page 84)

# Linked iterator Operations

```
//-----  
// Returns true if this iterator has at least one more element  
// to deliver in the iteration.  
//-----  
public boolean hasNext()  
{  
    return (current != null);  
}
```

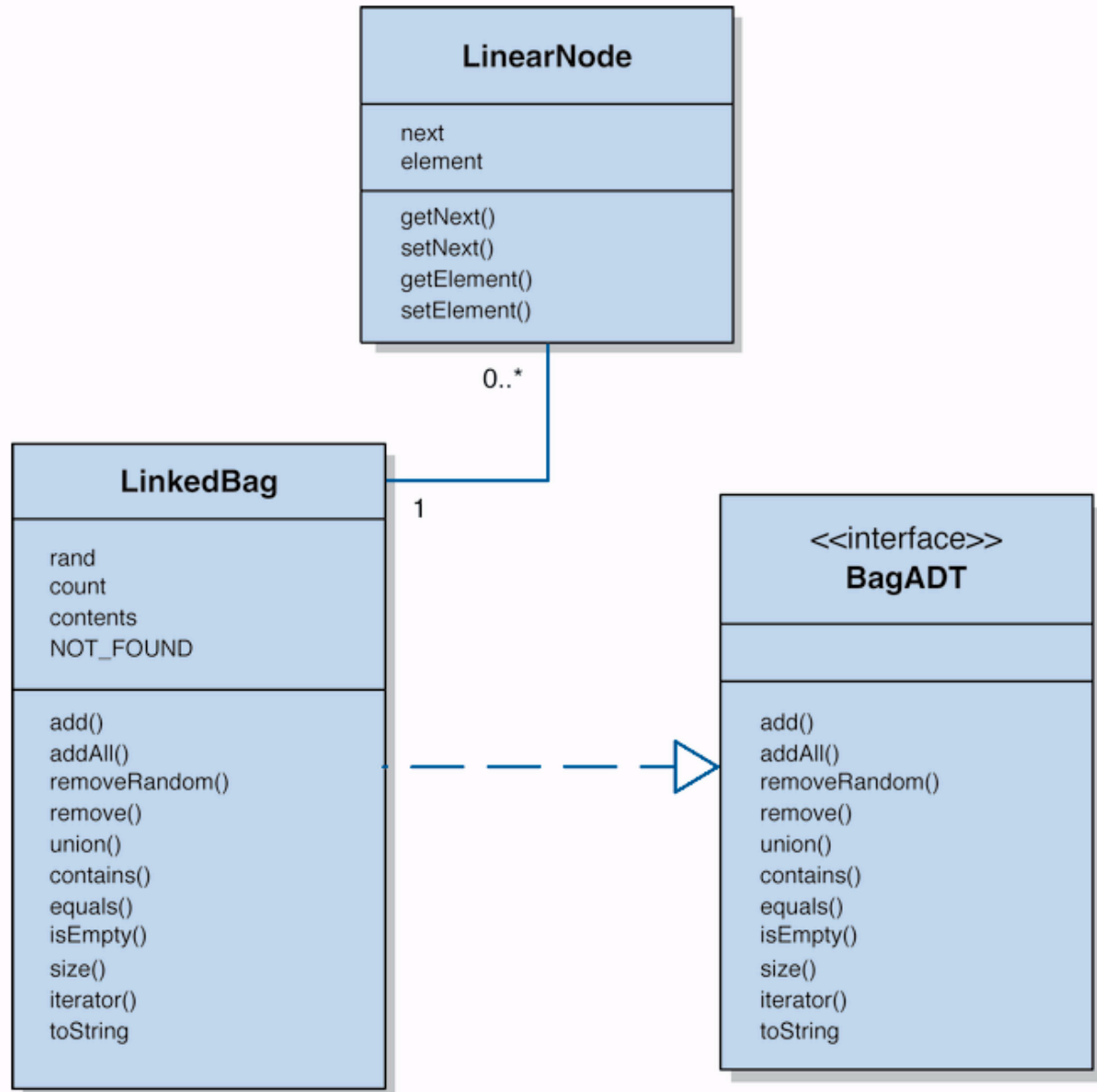
# Linked iterator Operations

```
//-----  
// Returns the next element in the iteration. If there are no  
// more elements in this iteration, a NoSuchElementException is  
// thrown.  
//-----  
  
    public Object next()  
    {  
        if (! hasNext())  
            throw new NoSuchElementException();  
  
        Object result = current.getElement();  
        current = current.getNext();  
        return result;  
    }
```

# The `iterator` Operation

```
//-----  
// The remove operation is not supported.  
//-----  
public void remove() throws UnsupportedOperationException  
{  
    throw new UnsupportedOperationException();  
}
```

**figure 3.11**  
UML  
description  
of the  
LinkedBag  
class



# Analysis of Linked Operations

- Since the order is irrelevant, and there is no capacity to expand, adding an element to the bag is  $O(1)$
- Removing a particular element, because it must be found, is  $O(n)$
- Removing a random element requires a traversal of the list, and therefore is  $O(n)$