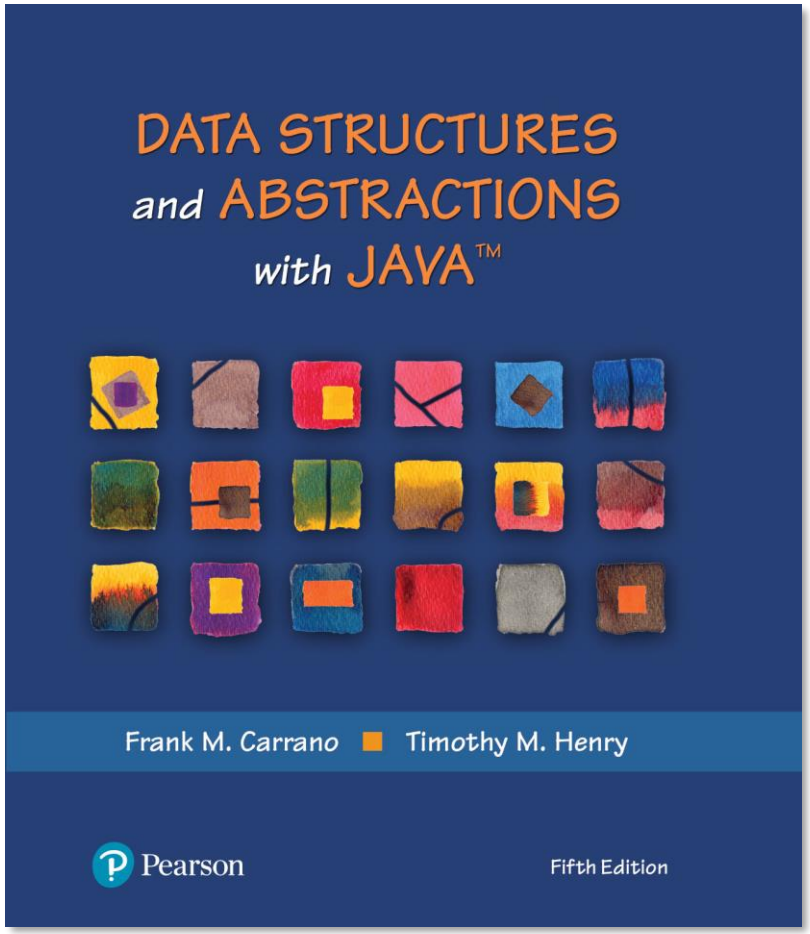
Data Structures and Abstractions with Java™

5th Edition



Chapter 3

A Bag Implementation That Links Data

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

Problems with Array Implementation

- Array has fixed size
- May become full
- Alternatively may have wasted space
- Resizing is possible but requires overhead of time

Analogy

- Empty classroom
- Numbered desks stored in hallway
  - Number on back of desk is the “address”
- Number on desktop references another desk in chain of desks
- Desks are linked by the numbers

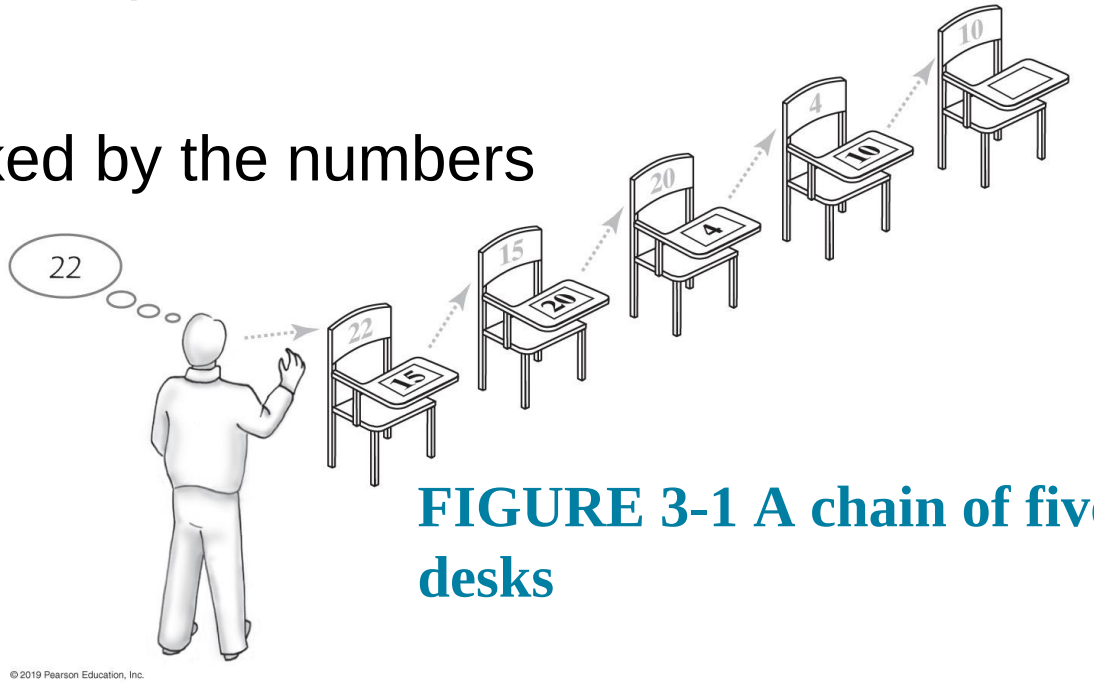


FIGURE 3-1 A chain of five desks

Forming a Chain by Adding to Its Beginning

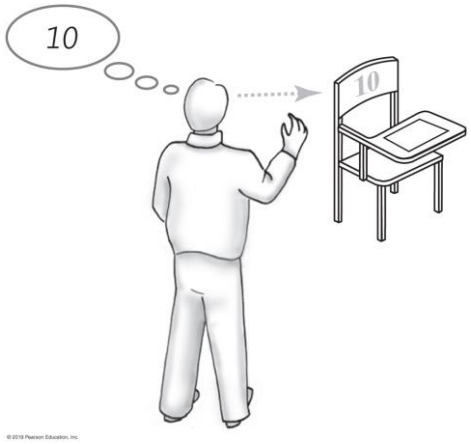


FIGURE 3-2
One desk in the room

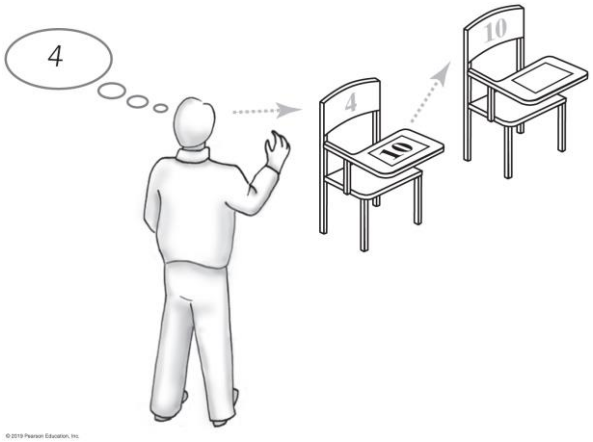


FIGURE 3-3
Two linked desks, with the newest desk first

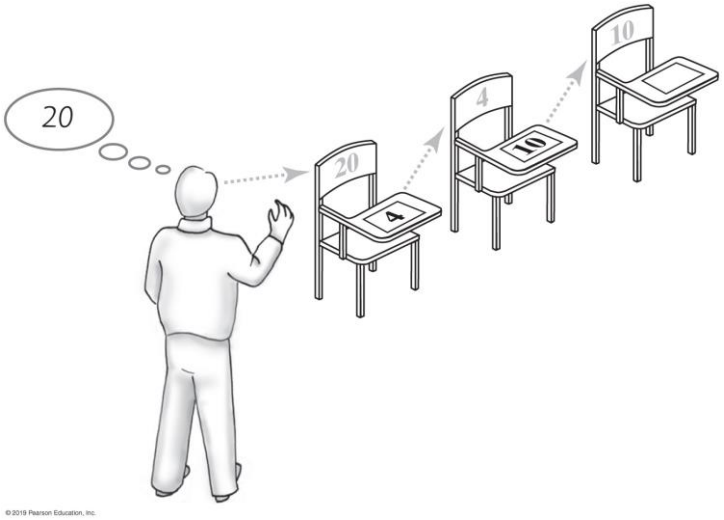


FIGURE 3-4
Three linked desks, with the newest desk first

Forming a Chain by Adding to Its Beginning

```
//Process the first student
newDesk represents the new student's desk New student sits at newDesk
Instructor memorizes the address of newDesk
// Process the remaining students
while (students arrive)
{
    newDesk represents the new student's desk New student sits at newDesk
    Write the instructor's memorized address on newDesk
    Instructor memorizes the address of newDesk
}
```

Pseudocode detailing steps taken to form a chain of desks

The Private Class Node

```
private class Node
{
  private T    data; // Entry in bag
  private Node next; // Link to next node

private Node(T dataPortion)
{
      this(dataPortion, null);
} // end constructor

private Node(T dataPortion, Node nextNode)
{
      data = dataPortion;
      next = nextNode;
} // end constructor
} // end Node
```

LISTING 3-1 The private inner class Node

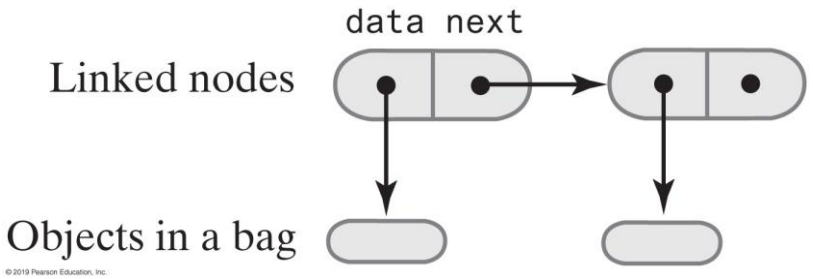


FIGURE 3-5
Two linked nodes that each reference object data

An Outline of the Class LinkedBag (Part 1)

```
/** OUTLINE
      A class of bags whose entries are stored in a chain of linked nodes.
   The bag is never full. */
public class LinkedBag<T> implements BagInterface<T>
{
     private Node firstNode;       // reference to first node
     private int numberOfEntries;

     public LinkedBag()
     {
     firstNode = null;
    numberOfEntries = 0;
     } // end default constructor

  // . . .
```

LISTING 3-2 An outline of the class LinkedBag

An Outline of the Class LinkedBag (Part 2)

```
    private class Node
    {
      private T    data; // Entry in bag
      private Node next; // Link to next node

    private Node(T dataPortion)
    {
        this(dataPortion, null);
    } // end constructor

    private Node(T dataPortion, Node nextNode)
    {
        data = dataPortion;
        next = nextNode;
    } // end constructor
    } // end Node
} // end LinkedBag
```

LISTING 3-2 An outline of the class LinkedBag

Beginning a Chain of Nodes

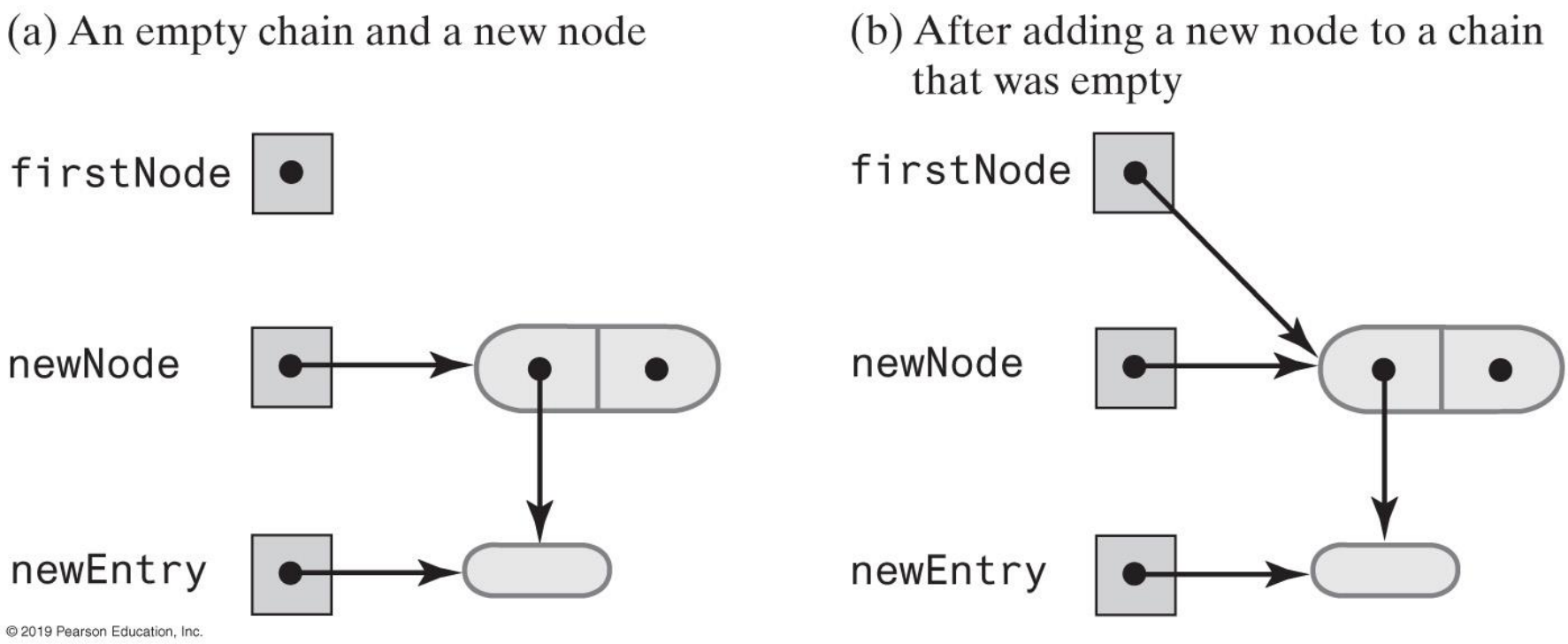


FIGURE 3-6 Adding a new node to an empty chain

Beginning a Chain of Nodes

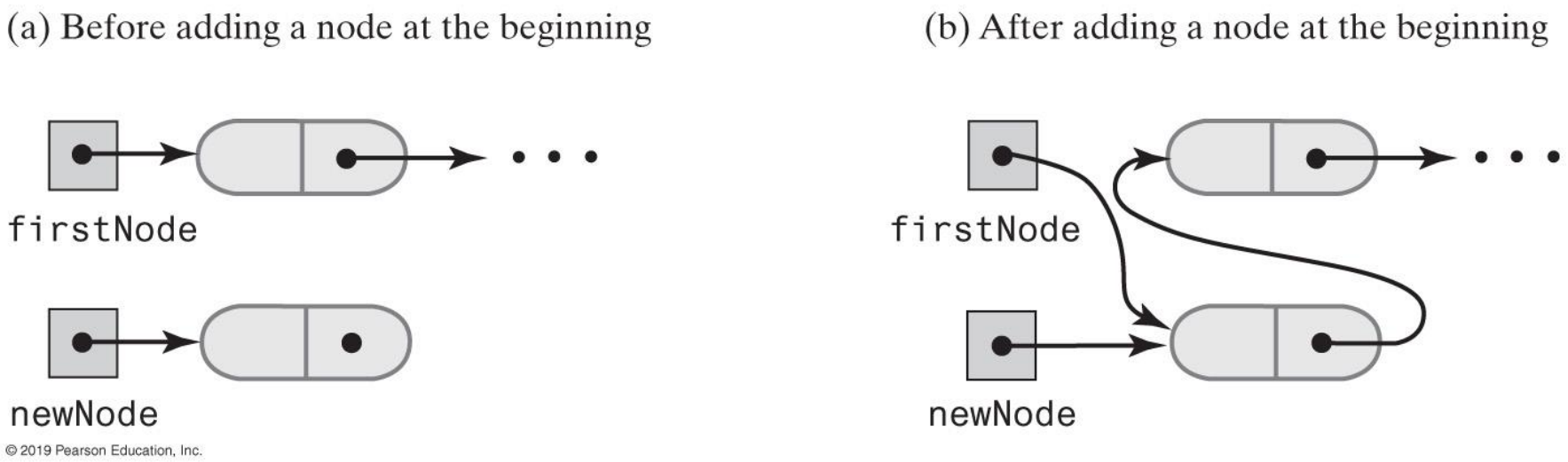


FIGURE 3-7 A chain of nodes just before and just after adding a node at the beginning

Beginning a Chain of Nodes

```
 /** Adds a new entry to this bag.
    @param newEntry  The object to be added as a new entry.
    @return  True. */
 public boolean add(T newEntry) // OutOfMemoryError possible
 {
// Add to beginning of chain:
 Node newNode = new Node(newEntry);
 newNode.next = firstNode;  // Make new node reference rest of chain
                  // (firstNode is null if chain is empty)
firstNode = newNode;      // New node is at beginning of chain
 numberOfEntries++;

 return true;
 } // end add
```

The method add

Method toArray

```
 /** Retrieves all entries that are in this bag.
    @return  A newly allocated array of all the entries in the bag. */
 public T[] toArray()
 {
// The cast is safe because the new array contains null entries
@SuppressWarnings("unchecked")
T[] result = (T[])new Object[numberOfEntries]; // Unchecked cast

int index = 0;
Node currentNode = firstNode;
while ((index < numberOfEntries) && (currentNode != null))
{
  result[index] = currentNode.data;
  index++;
  currentNode = currentNode.next;
} // end while

 return result;
 } // end toArray
```

The method toArray returns an array of the entries currently in a bag

LinkedBag Test Program (Part 1)

```
/** A test of the methods add, toArray, isEmpty, and getCurrentSize,
   as defined in the first draft of the class LinkedBag.   */
public class LinkedBagDemo1
{
      public static void main(String[] args)
      {
     System.out.println("Creating an empty bag.");
     BagInterface<String> aBag = new LinkedBag1<>();
     testIsEmpty(aBag, true);
      displayBag(aBag);

     String[] contentsOfBag = {"A", "D", "B", "A", "C", "A", "D"};
      testAdd(aBag, contentsOfBag);
      testIsEmpty(aBag, false);
      } // end main
```

LISTING 3-3 A sample program that tests some methods in the class LinkedBag

LinkedBag Test Program (Part 2)

```
// Tests the method isEmpty.
// Precondition: If bag is empty, the parameter empty should be true;
// otherwise, it should be false.
    private static void testIsEmpty(BagInterface<String> bag,
                                                                boolean empty)
{
  System.out.print("\nTesting isEmpty with ");
  if (empty)
    System.out.println("an empty bag:");
  else
    System.out.println("a bag that is not empty:");

  System.out.print("isEmpty finds the bag ");
  if (empty && bag.isEmpty())
        System.out.println("empty: OK.");
   else if (empty)
        System.out.println("not empty, but it is: ERROR.");
   else if (!empty && bag.isEmpty())
        System.out.println("empty, but it is not empty: ERROR.");
   else
        System.out.println("not empty: OK.");
  } // end testIsEmpty
```

LISTING 3-3 A sample program that tests some methods in the class LinkedBag

LinkedBag Test Program (Part 3)

```
// Tests the method add.
private static void testAdd(BagInterface<String> aBag,
                                                    String[] content)
{
  System.out.print("Adding the following strings to the bag: ");
  for (int index = 0; index < content.length; index++)
  {
    if (aBag.add(content[index]))
      System.out.print(content[index] + " ");
    else
      System.out.print("\nUnable to add " + content[index] +
                       " to the bag.");
  } // end for
  System.out.println();

  displayBag(aBag);
} // end testAdd
```

LISTING 3-3 A sample program that tests some methods in the class LinkedBag

LinkedBag Test Program (Part 4)

```
// Tests the method toArray while displaying the bag.
 private static void displayBag(BagInterface<String> aBag)
 {
   System.out.println("The bag contains the following string(s):");
   Object[] bagArray = aBag.toArray();
   for (int index = 0; index < bagArray.length; index++)
   {
     System.out.print(bagArray[index] + " ");
   } // end for

   System.out.println();
 } // end displayBag

} // end LinkedBagDemo1
```

Program Output

Creating an empty bag.

Testing isEmpty with an empty bag:
isEmpty finds the bag empty: OK.
The bag contains the following string(s):

Adding the following strings to the bag: A D B A C A D
The bag contains the following string(s):
D A C A B D A

Testing isEmpty with a bag that is not empty:
isEmpty finds the bag not empty: OK.

LISTING 3-3 A sample program that tests some methods in the class LinkedBag

Method getFrequencyOf

```
  /** Counts the number of times a given entry appears in this bag.
   @param anEntry  The entry to be counted.
   @return  The number of times anEntry appears in the bag. */
  public int getFrequencyOf(T anEntry)
  {
  int frequency = 0;
int loopCounter = 0;
Node currentNode = firstNode;

while ((loopCounter < numberOfEntries) && (currentNode != null))
{
   if (anEntry.equals(currentNode.data))
   {
     frequency++;
   } // end if

   loopCounter++;
   currentNode = currentNode.next;
} // end while

  return frequency;
  } // end getFrequencyOf
```

Counts the number of times a given entry appears

Method contains

```
/** Tests whether this bag contains a given entry.
    @param anEntry  The entry to locate.
   @return  True if the bag contains anEntry, or false otherwise */

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

  while (!found && (currentNode != null))
  {
    if (anEntry.equals(currentNode.data))
      found = true;
    else
      currentNode = currentNode.next;
  } // end while
  return found;
} // end contains
```

Determine whether a bag contains a given entry

Removing an Item from a Linked Chain

- **Case 1:**
  - Your desk is first in the chain of desks.
- **Case 2:**
  - Your desk is not first in the chain of desks.

Removing an Item from a Linked Chain

- **Case 1**
  - Locate first desk by asking instructor for its address.
  - Give address written on the first desk to instructor.
    - This is address of second desk in chain.
  - Return first desk to hallway.

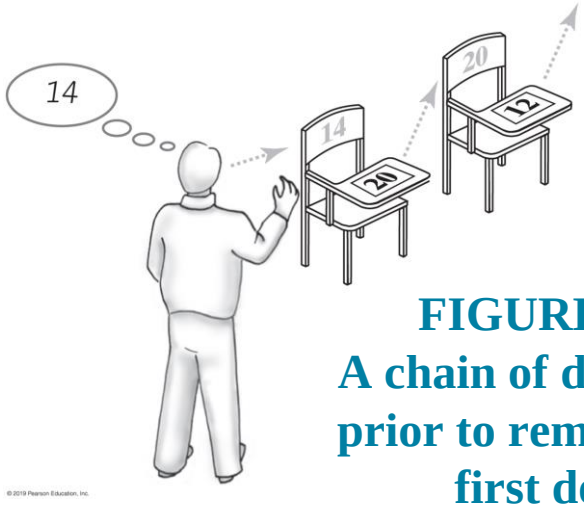


FIGURE 3-8
A chain of desks just prior to removing its first desk

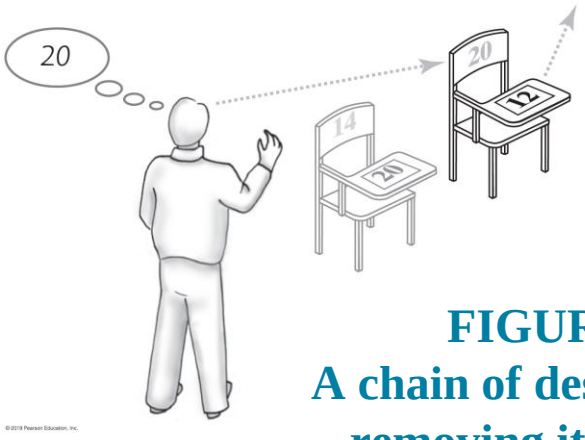


FIGURE 3-9
A chain of desks just after removing its first desk

Removing an Item from a Linked Chain

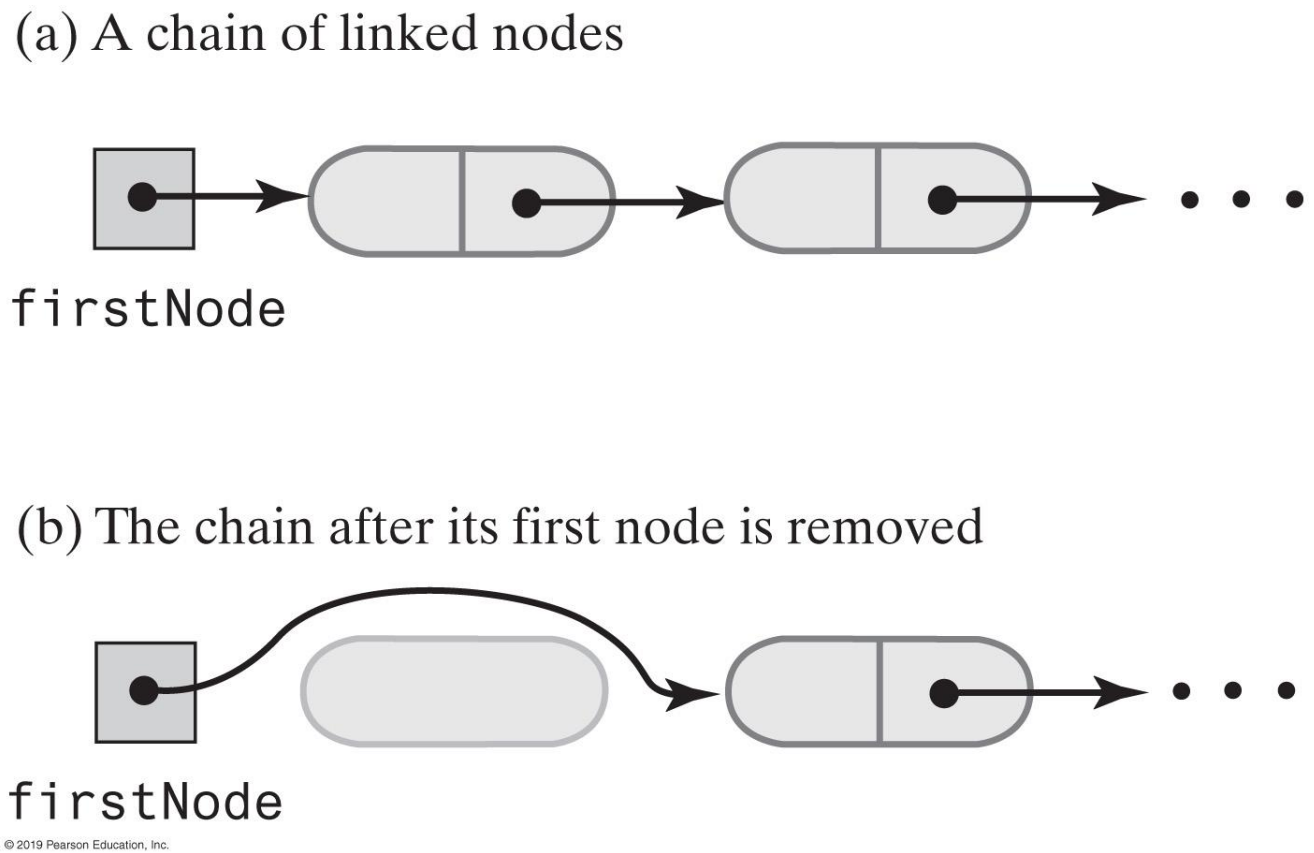


FIGURE 3-10 A chain of nodes just before and just after its first node is removed

Removing an Item from a Linked Chain

- **Case 2**
  - Move the student in the first desk to your former desk.
  - Remove the first desk using the steps described for Case 1.

Method remove

```
// Locates a given entry within this bag.
// Returns a reference to the node containing the entry, if located,
// or null otherwise.
private Node getReferenceTo(T anEntry)
{
boolean found = false;
Node currentNode = firstNode;

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

return currentNode;
} // end getReferenceTo
```

Private helper method getReferenceTo

Method remove

```
/** Removes one unspecified entry from this bag, if possible.
@return  Either the removed entry,
                        if the removal was successful, or null */
public T remove()
{
T result = null;
if (firstNode != null)
{
  result = firstNode.data;
  firstNode = firstNode.next; // Remove first node from chain
  numberOfEntries--;
} // end if

return result;
} // end remove
```

Uses private helper method `getReferenceTo`

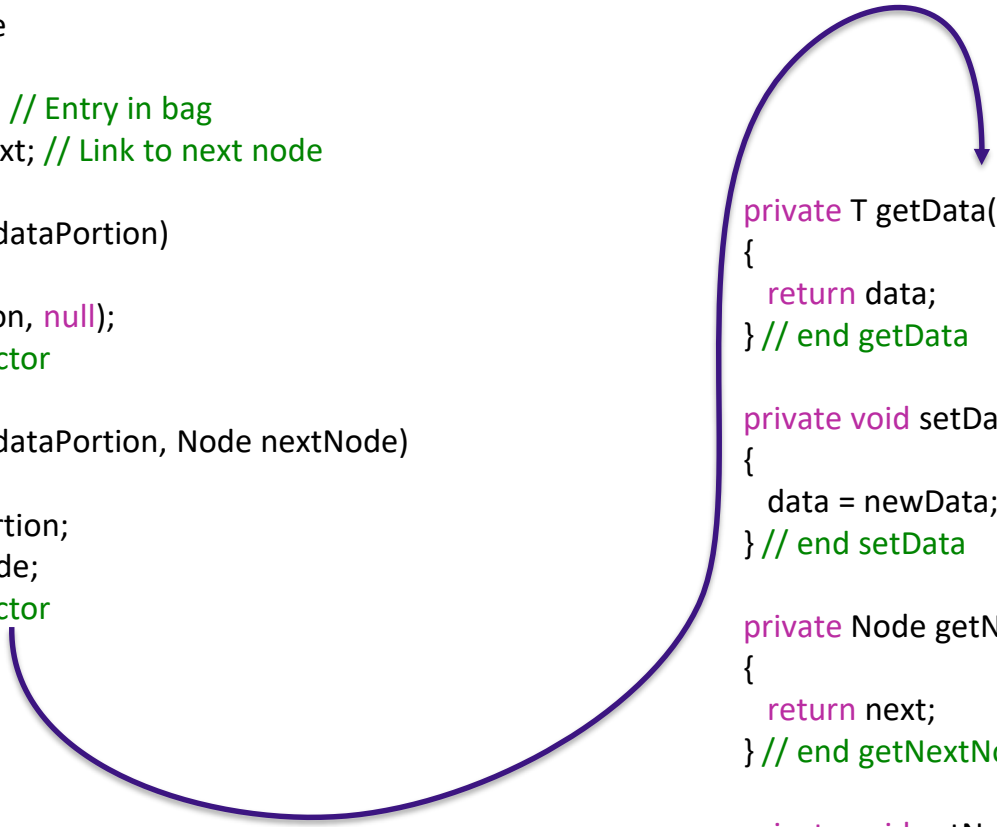
Method clear

```
/** Removes all entries from this bag.  */
    public void clear()
    {
    while (!isEmpty())
      remove();
    } // end clear
```

As in previous implementations, uses `isEmpty` and `remove`

Class Node That Has Set and Get Methods

```
private class Node
{
  private T data;    // Entry in bag
  private Node next; // Link to next node

  private Node(T dataPortion)
  {
    this(dataPortion, null);
  } // end constructor

  private Node(T dataPortion, Node nextNode)
  {
    data = dataPortion;
    next = nextNode;
  } // end constructor

  private T getData()
  {
    return data;
  } // end getData

  private void setData(T newData)
  {
    data = newData;
  } // end setData

  private Node getNextNode()
  {
    return next;
  } // end getNextNode

  private void setNextNode(Node nextNode)
  {
    next = nextNode;
  } // end setNextNode
} // end Node
```



LISTING 3-4 The inner class Node with set and get methods

A Class within A Package

```
package BagPackage;
class Node<T>
{
  private T      data;
  private Node<T> next;

  Node(T dataPortion)
  {
    this(dataPortion, null);
  } // end constructor

  Node(T dataPortion, Node<T> nextNode)
  {
    data = dataPortion;
    next = nextNode;
  } // end constructor

  T getData()
  {
    return data;
  } // end getData

  void setData(T newData)
  {
    data = newData;
  } // end setData

  Node<T> getNextNode()
  {
    return next;
  } // end getNextNode

  void setNextNode(Node<T> nextNode)
  {
    next = nextNode;
  } // end setNextNode
} // end Node
```

LISTING 3-5 The class `Node` with package access

When Node Is in Same Package

```
package BagPackage;
public class LinkedBag<T> implements BagInterface<T>
{
  private Node<T> firstNode;

  public boolean add(T newEntry)
  {
    Node<T> newNode = new Node<T>(newEntry);
    newNode.setNextNode(firstNode);
    firstNode = newNode;
    numberOfEntries++;

    return true;
  } // end add

  // . . .
} // end LinkedBag
```

LISTING 3-6 The class LinkedBag when Node is in the same package

Pros of Using a Chain

- Bag can grow and shrink in size as necessary.
- Remove and recycle nodes that are no longer needed
- Adding new entry to end of array or to beginning of chain both relatively simple
- Similar for removal

Cons of Using a Chain

- Removing specific entry requires search of array or chain
- Chain requires more memory than array of same length

End

Chapter 3