Week 15

Basic Data Structures

(Chapter 16)

Chapter Goals

© andrea laurita/iStockphoto.

- To understand the implementation of linked lists and array lists
- To analyze the efficiency of fundamental operations of lists and arrays
- To implement the stack and queue data types
- To implement a hash table and understand the efficiency of its operations

Implementing Linked Lists - The Node Class

- We will implement a simplified, singly-linked list.
- A linked list stores elements in a sequence of nodes.
- A `Node` object stores an element and a reference to the next node.
  - private inner class
  - public instance variables

```
public class LinkedList
{
   . . .
   class Node
   {
      public Object data;
      public Node next;
   }
}
```

Implementing Linked Lists - The Node Class

- A linked list object holds a reference to the first node:
  - Each node holds a reference to the next node.

```
public class LinkedList
{
  private Node first;

  public LinkedList() { first = null; }


  public Object getFirst()
  {
    if (first == null) { throw new NoSuchElementException(); }
      return first.data;
  }
}
```

Implementing Linked Lists - Adding and Removing the First Element

- When adding or removing the first element, the reference to the first node must be updated.

```
public class LinkedList
{
   . . .
   public void addFirst(Object element)
   {
      Node newNode = new Node();1
      newNode.data = element;
      newNode.next = first; 2
      first = newNode;3
   }
   . . .
}
```

Implementing Linked Lists - Adding the First Element

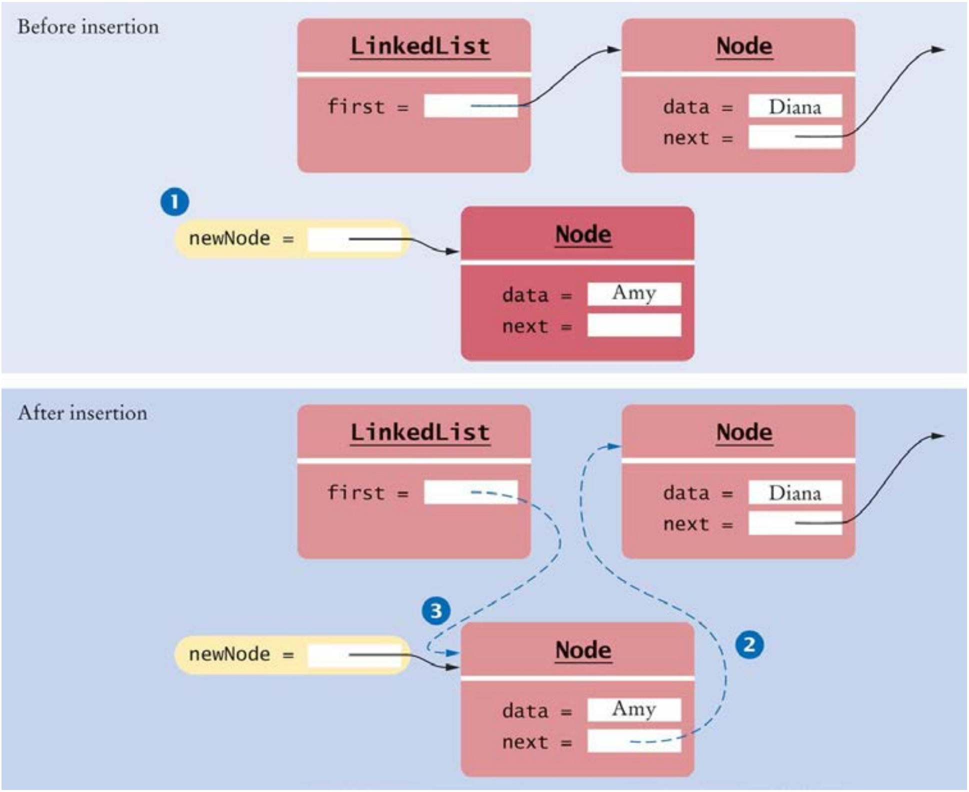

Figure 1 Adding a Node to the Head of a Linked List

Implementing Linked Lists - Removing the First Element

- The data of the first node are saved and later returned as the method result.
- The successor of the first node becomes the first node of the shorter list.
    - The old node is eventually recycled by the garbage collector.

```
public class LinkedList
{
  . . .
  public Object removeFirst()
  {
    if (first == null) { throw new NoSuchElementException(); }
    Object element = first.data;
    first = first.next;
    return element;
  }
  . . .
}
```

Implementing Linked Lists - Removing the First Element

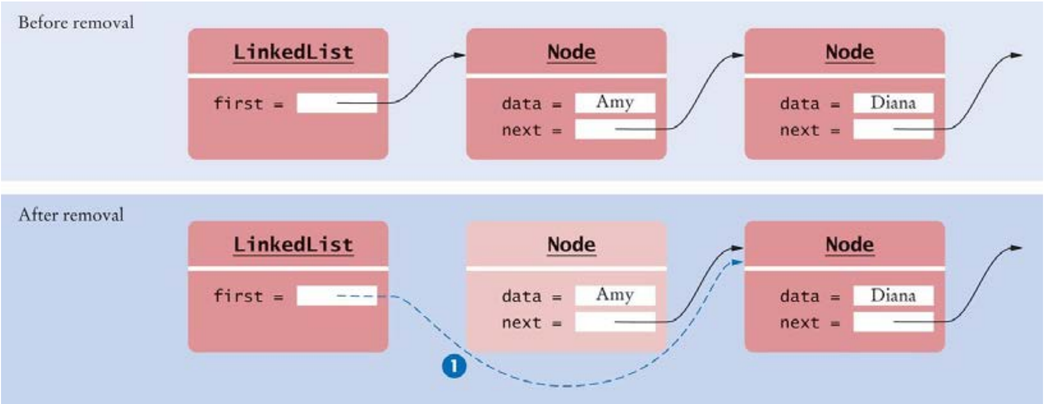

Figure 2 Removing the First Node from a LinkedList

The Iterator Class

- Our simplified `ListIterator` interface has methods: `next`, `hasNext`, `remove`, `add`, and `set`.
- Our `LinkedList` class declares a private inner class `LinkedListIterator`.
  - `LinkedListIterator` implements our simplified `ListIterator` interface.
  - As an inner class `LinkedListIterator` has access to
    - The instance variable `first`
    - The private `Node` class.
- A list iterator object has:
  - A reference to the the currently visited node, `position`
  - A reference to the last node before that, `previous`
  - A `isAfterNext` flag to track when the `next` method has been called.

The Iterator Class

- The `LinkedListIterator` class:

```
public class LinkedList
{
   . . .
   public ListIterator listIterator()
   {
      return new LinkedListIterator();
   }
   class LinkedListIterator implements ListIterator
   {
      private Node position;
      private Node previous;
      private boolean isAfterNext;
      public LinkedListIterator()
      {
         position = null;
         previous = null;
         isAfterNext = false;
      }
      . . .
   }
}
```

Advancing an Iterator

- To advance an iterator:
  - Update the position
  - Remember the old position for the `remove` method.
- The `next` method:

```
class LinkedListIterator implements ListIterator
{
                  . . .
  public Object next()
  {
    if (!hasNext()) { throw new NoSuchElementException(); }
    previous = position; // Remember for remove
    isAfterNext = true;

    if (position == null)
    { position = first; }
    else
    { position = position.next; }
    return position.data;
  }
                  . . .
}
```

Advancing an Iterator

- The iterator is at the end
  - if the list is empty (`first == null`) or
  - if there is no element after the current position (`position.next == null`).
- The `hasNext` method:

```
class LinkedListIterator implements ListIterator
{
         . . .
   public boolean hasNext()
   {
      if (position == null)
      {
         return first != null;
      }
      else
      {
         return position.next != null;
      }
   }
         . . .
}
```

Removing an Element

- If this is the first element:
  - Call `removeFirst`
  - Otherwise, update the `next` reference of the previous node
- Update `isAfterNext` to disallow another call to `remove`.
- The `remove` method:

```
class LinkedListIterator implements ListIterator
{
                . . .
  public void remove()
  {
    if (!isAfterNext) { throw new IllegalStateException(); }
    if (position == first)
    {
      removeFirst();
    }
    else
    {
      previous.next = position.next;  1
    }
    position = previous;  2
    isAfterNext = false;  3
  }
                . . .
}
```

Removing an Element

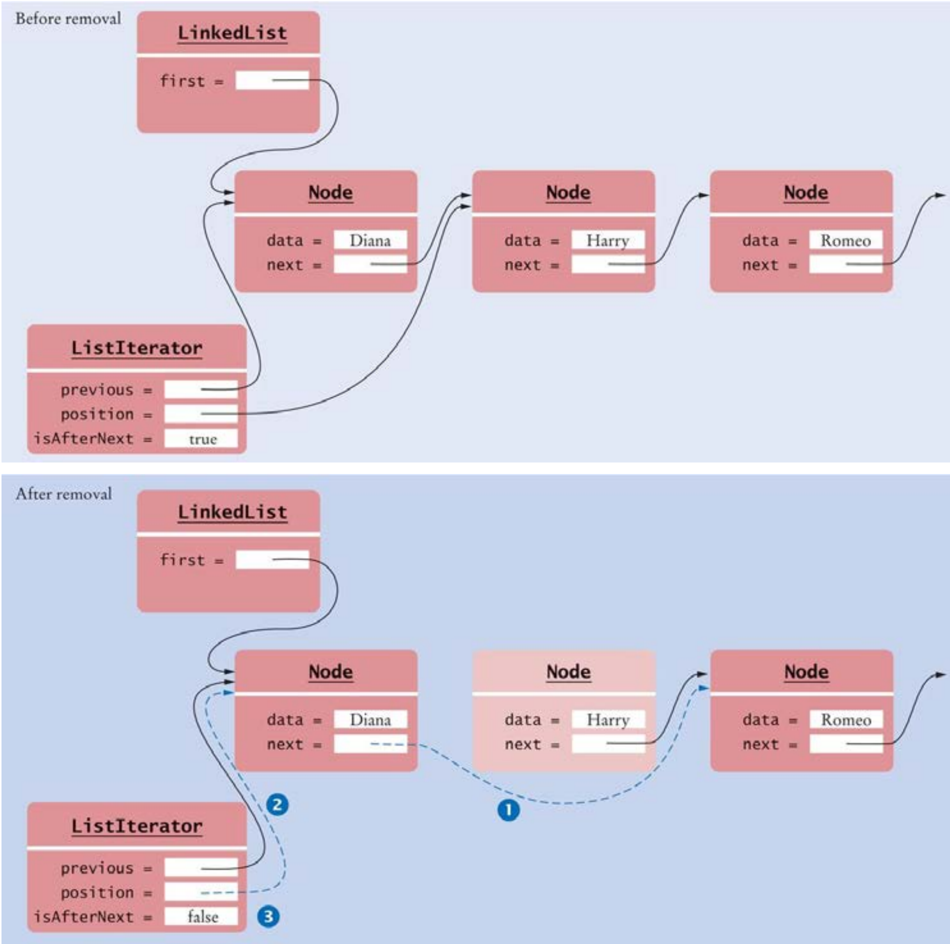

Figure 3 Removing a Node from the Middle of a Linked List

Adding an Element

- After adding the new element
  - set the `isAfterNext` flag to false to disallow a subsequent call to the `remove` or `set` method
- The add method:

```
class LinkedListIterator implements ListIterator
{
          . . .
   public void add(Object element)
   {
      if (position == null)
      {
         addFirst(element);
         position = first;
      }
      else
      {
         Node newNode = new Node();
         newNode.data = element;
         newNode.next = position.next;   1
         position.next = newNode;   2
         position = newNode;   3
      }
      isAfterNext = false;   4
   }
          . . .
}
```

Adding an Element

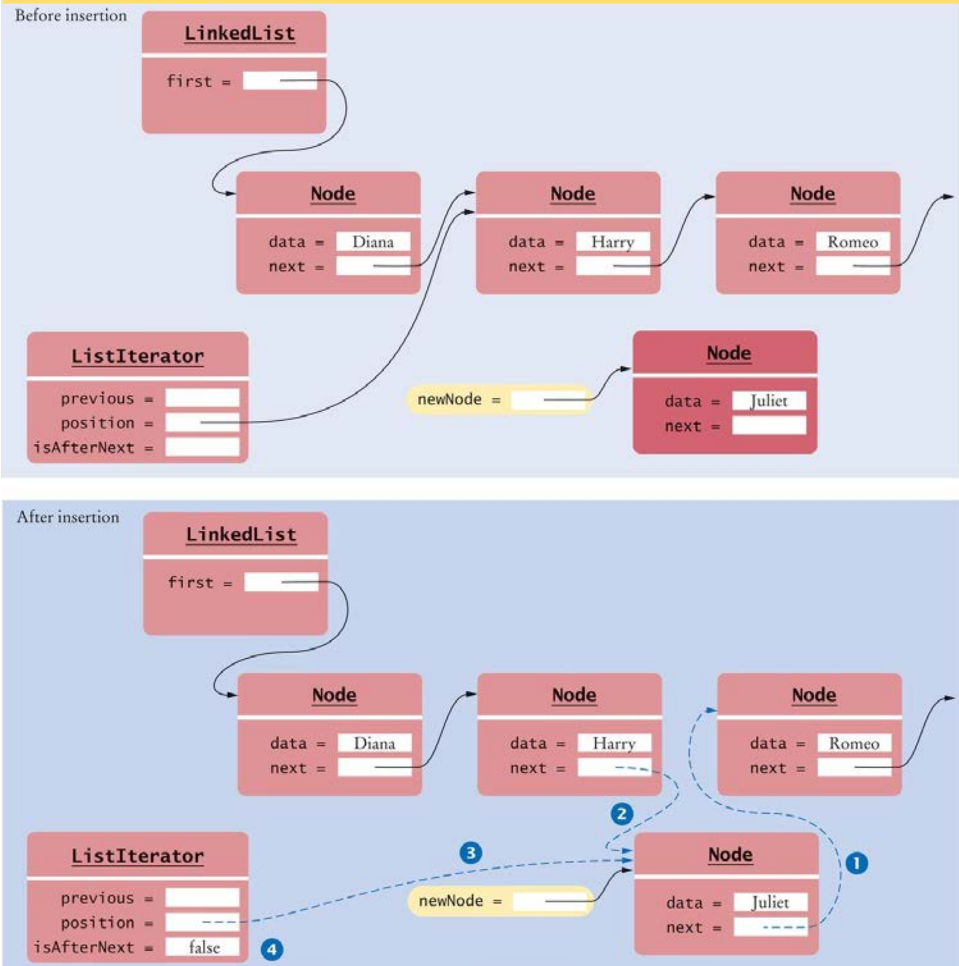

Figure 4 Adding a Node to the Middle of a Linked List

Setting an Element to a Different Value

- `Set` method changes the data in the previously visited element.
- Must follow a call to `next`.
- The `set` method:

```
public void set(Object element)
{
  if (!isAfterNext) { throw new IllegalStateException(); }
  position.data = element;
}
```

Efficiency of Linked List Operations

- To get the k^{th} element of a linked list, you start at the beginning of the list and advance the iterator *k* times.
- To get to the k^{th} node of a linked list, one must skip over the preceding nodes.

Efficiency of Linked List Operations

- When adding or removing an element, we update a couple of references in a constant number of steps.
- **Adding and removing an element at the iterator position in a linked list takes $O(1)$ time.**

Efficiency of Linked List Operations

- To add an element at the end of the list
  - Must get to the end - an O(n) operation
  - Add the element O(1) operation
- Adding to the end of a linked list in our implementation takes $O(n)$ time

  If the linked list keeps a reference to `last` as well as `first`
  - The time is reduced to constant time: $O(1)$

```
public class LinkedList
{
   private Node first;
   private Node last;
   . . .
}
```

- **We will conclude that adding to the end of a linked list is *O*(1).**

Efficiency of Linked List Operations

- To remove an element from the end of the list:
  - Need a reference to the next-to-last element so that we can set its `next` reference to null
  - Takes n-1 iterations
- Removing an element from the end of the list is $O(n)$.

Efficiency of Linked List Operations

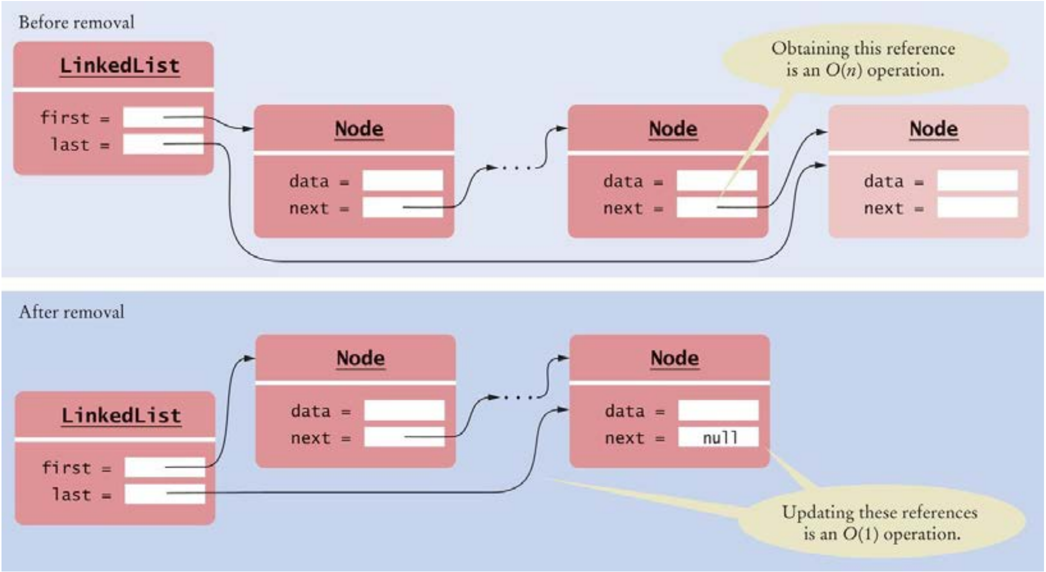

Figure 5 Removing the Last Element of a Singly-Linked List

Efficiency of Linked List Operations

- In a doubly-linked list, each node has a reference to the previous node in addition to the next one.

```
public class LinkedList
{
   . . .
   class Node
   {
      public Object data;
      public Node next;
      public Node previous;
   }
}
```

Efficiency of Linked List Operations

- In a doubly-linked list, removal of the last element takes a constant number of steps.

```
last = last.previous; 1
last.next = null; 2
```

Efficiency of Linked List Operations

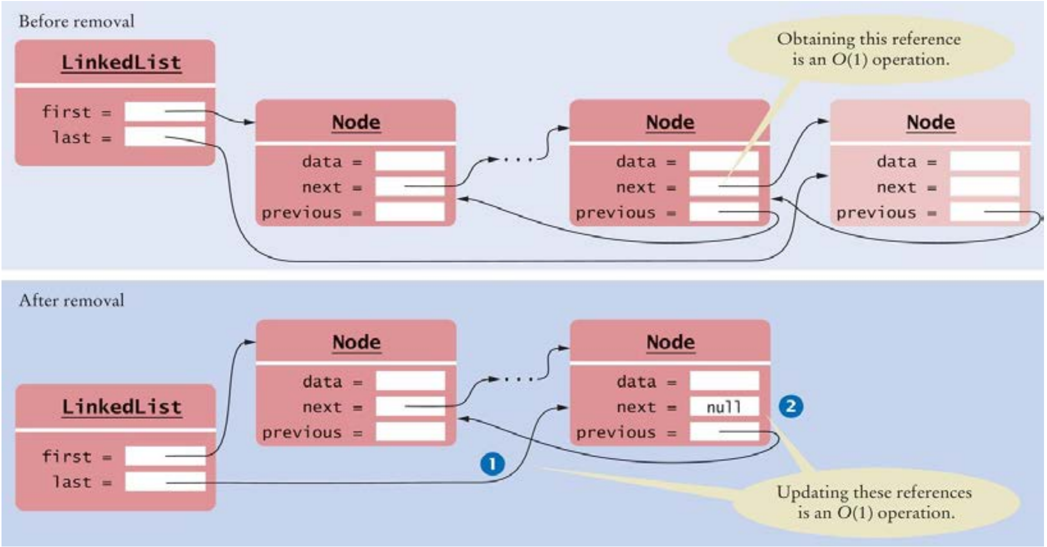

Figure 6 Removing the Last Element of a Doubly-Linked List

Efficiency of Linked List Operations

Table 1 Efficiency of Linked List Operations

Operation	Singly-Linked List	Doubly-Linked List
Access an element.	$O(n)$	$O(n)$
Add/remove at an iterator position.	$O(1)$	$O(1)$
Add/remove first element.	$O(1)$	$O(1)$
Add last element.	$O(1)$	$O(1)$
Remove last element.	$O(n)$	$O(1)$

section_1/LinkedList.java

```java
import java.util.NoSuchElementException;

/**
    A linked list is a sequence of nodes with efficient
    element insertion and removal. This class
    contains a subset of the methods of the standard
    java.util.LinkedList class.
*/
public class LinkedList
{
   private Node first;

   /**
      Constructs an empty linked list.
   */
   public LinkedList()
   {
      first = null;
   }

   /**
      Returns the first element in the linked list.
      @return the first element in the linked list
   */
   public Object getFirst()
   {
      if (first == null) { throw new NoSuchElementException();
      }  return first.data;
   }

   /**
      Removes the first element in the linked list.
      @return the removed element
   */
```

section_1/ListIterator.java

```
/**
    A list iterator allows access of a position in a linked list.
    This interface contains a subset of the methods of the
    standard java.util.ListIterator interface. The methods for
    backward traversal are not included.
*/
public interface ListIterator
{
   /**
      Moves the iterator past the next element.
      @return the traversed element
   */
   Object next();

   /**
      Tests if there is an element after the iterator position.
      @return true if there is an element after the iterator position
   */
   boolean hasNext();

   /**
      Adds an element before the iterator position
      and moves the iterator past the inserted element.
      @param element the element to add
   */
   void add(Object element);

   /**
      Removes the last traversed element. This method may
      only be called after a call to the next() method.
   */
   void remove();
```

Static Classes

- Every object of an inner class has a reference to the outer class.
  - It can access the instance variables and methods of the outer class.
- If an inner class does not need to access the data of the outer class,
  - It does not need a reference.
  - Declare it static to save the cost of the reference.
- Example: Declare the `Node` class of the `LinkedList` class as `static`:

```
public class LinkedList
{
   . . .
   static class Node
   {
      . . .
   }
}
```

Implementing Array Lists

- An array list maintains a reference to an array of elements.
- The array is large enough to hold all elements in the collection.
- When the array gets full, it is replaced by a larger one.
- An array list has an instance field that stores the current number of elements.

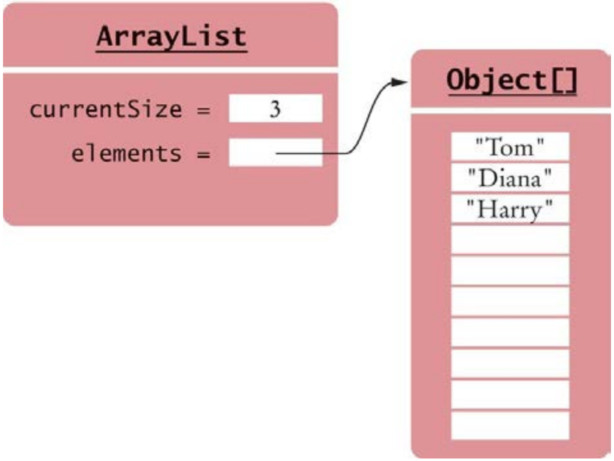

Figure 7 An Array List Stores its Elements in an Array

Implementing Array Lists

- Our `ArrayList` implementation will manage elements of type `Object`:

```
public class ArrayList
{
   private Object[] elements;
   private int currentSize;

   public ArrayList()
   {
      final int INITIAL_SIZE = 10;
      elements = new Object[INITIAL_SIZE];
      currentSize = 0;
   }

   public int size() { return currentSize; }
   . . .
}
```

Implementing Array Lists - Getting and Setting Elements

- Providing `get` and `set` methods:
  - Check for valid positions
  - Access the internal array at the given position
- Helper method to check bounds:

```
private void checkBounds(int n)
{
   if (n < 0 || n >= currentSize)
   {
      throw new IndexOutOfBoundsException();
   }
}
```

Implementing Array Lists - Getting and Setting Elements

- The `get` method:

```
public Object get(int pos)
{
   checkBounds(pos);
   return element[pos];
}
```

- The `set` method:

```
public void set(int pos, Object element)
{
   checkBounds(pos);
   elements[pos] = element;
}
```

- Getting and setting an element can be carried out with a bounded set of instructions, independent of the size of the array list.
- These are $O(1)$ operations.

Removing or Adding Elements

- To remove an element at position *k*, move the elements with higher index values.
- The `remove` method:

```
public Object remove(int pos)
{
   checkBounds(pos);
   Object removed = elements[pos];
   for (int i = pos + 1; i < currentSize; i++)
   {
      elements[i - 1] = elements[i];
   }
   currentSize--;
   return removed;
}
```

- On average, n / 2 elements need to move.
- Inserting a element also requires moving, on average, n /2 elements.
- Inserting or removing an array list element is an $O(n)$ operation.

Removing or Adding Elements

- Exception: adding an element after the last element
  - Store the element in the array
  - Increment size
- An $O(1)$ operation
- A the `addLast` method:

```
public boolean addLast(Object newElement)
{
   growIfNecessary();
   currentSize++;
   elements[currentSize - 1] = newElement;
   return true;
}
```

Removing or Adding Elements

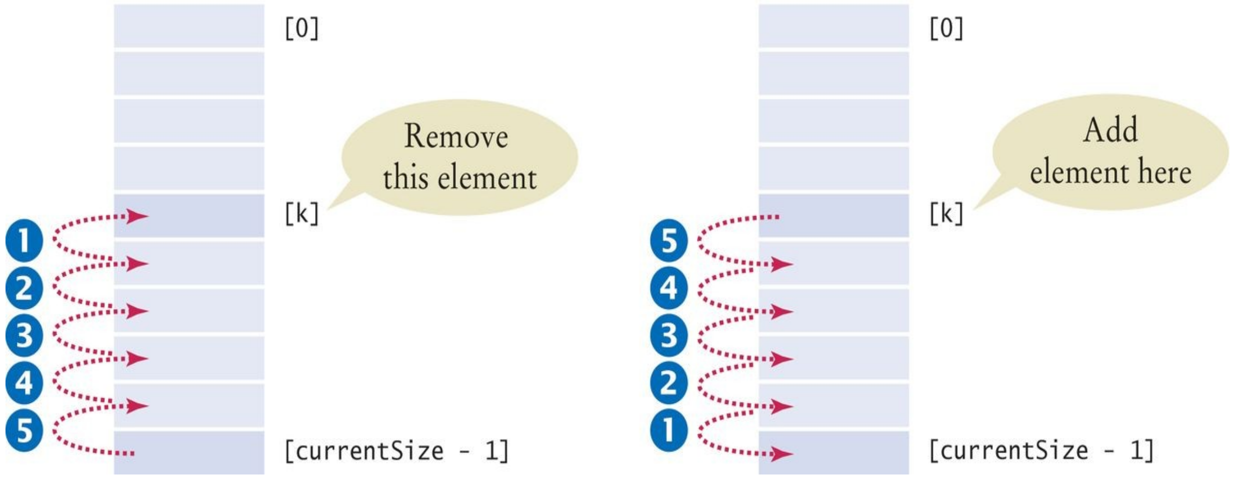

Figure 8 Removing and Adding Elements

Growing the Internal Array

© Craig Dingle/iStockphoto.

When an array list is completely full, we must move the contents to a larger array.

Growing the Internal Array

- When the array is full:
  - Create a bigger array
  - Copy the elements to the new array
  - New array replaces old
- Reallocation is *O*(*n*).
- The `growIfNecessary` method:

```
private void growIfNecessary()
{
  if (currentSize == elements.length)
  {
    Object[] newElements = new Object[2 * elements.length]; 1
    for (int i = 0; i < elements.length; i++)
    {
      newElements[i] = elements[i];   2
    }

    elements = newElements;  3
  }
}
```

Growing the Internal Array

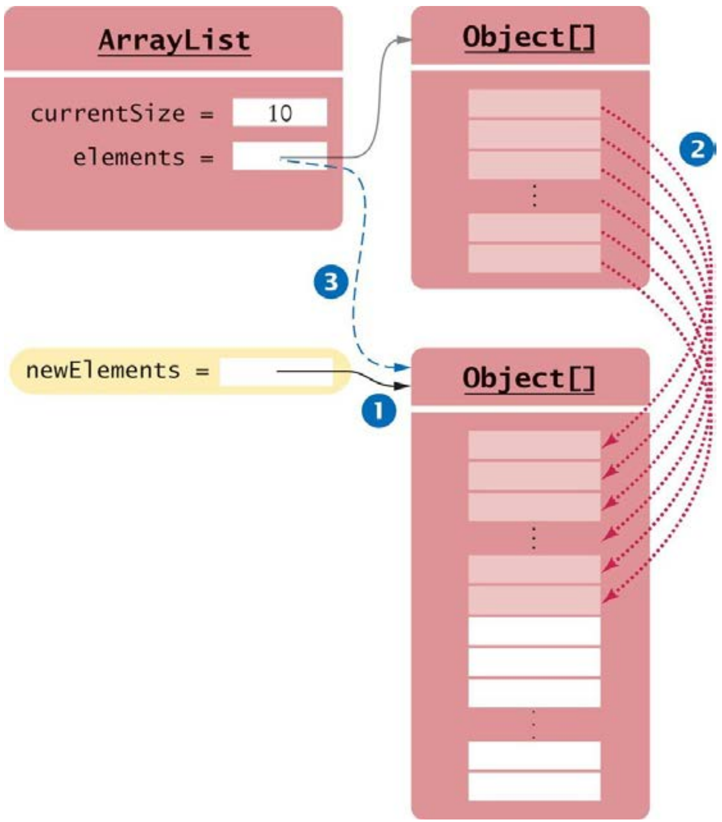

Figure 9 Reallocating the Internal Array

Growing the Internal Array

- Reallocation seldom happens.
- We amortize the cost of the reallocation over all the insertion or removals.
- Adding or removing the last element in an array list takes amortized $O(1)$ time. Written O(1)+

Efficiency of Array List and Linked List Operations

Table 2 Efficiency of Array List and Linked List Operations

Operation	Array List	Doubly-Linked List
Add/remove element at end.	$O(1)+$	$O(1)$
Add/remove element in the middle.	$O(n)$	$O(1)$
Get kth element.	$O(1)$	$O(k)$

Implementing Stacks and Queues

- Stacks and queues are abstract data types.
- We specify how operations must behave.
- We do not specify the implementation.
- Many different implementations are possible.

Stacks as Linked Lists

- A stack can be implemented as a sequence of nodes.
- New elements are “pushed” to one end of the sequence, and they are “popped” from the same end.
- Push and pop from the least expensive end - the front.
- The `push` and `pop` operations are identical to the `addFirst` and `removeFirst` operations of the linked list.

Stacks as Linked Lists

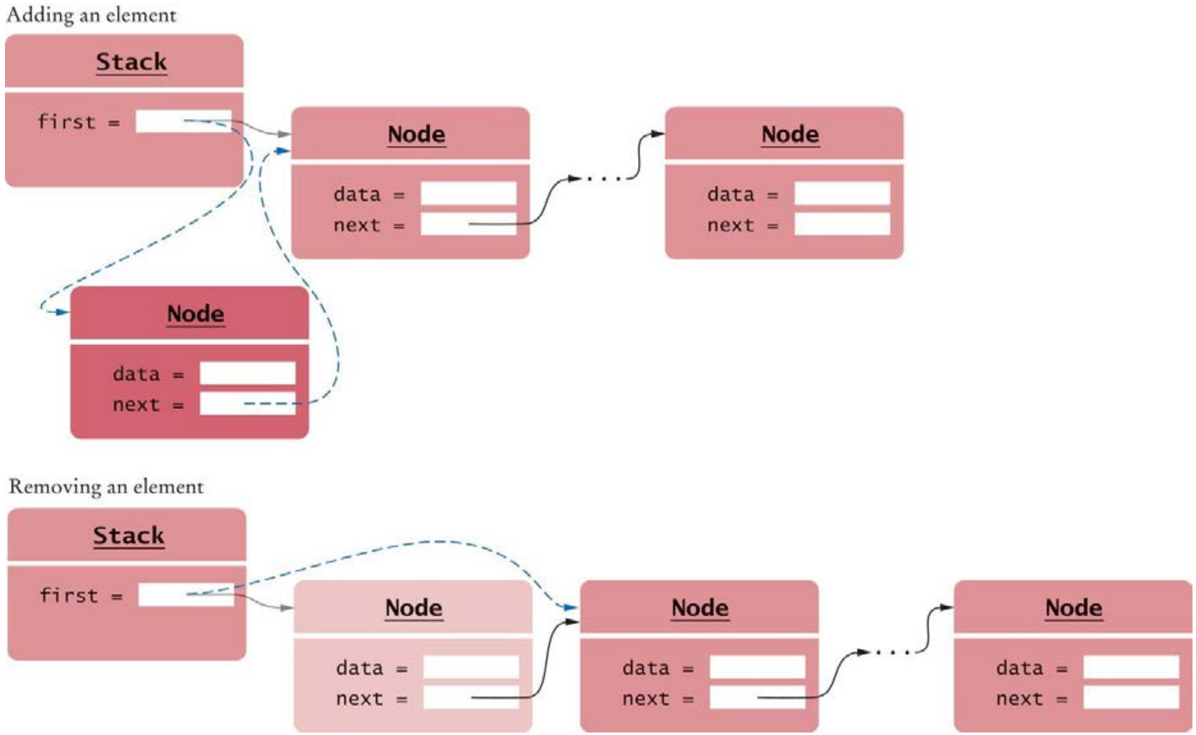

Figure 10 Push and Pop for a Stack Implemented as a Linked List

section_3_1/LinkedListStack.java

```java
import java.util.NoSuchElementException;

/**
   An implementation of a stack as a sequence of nodes.
*/
public class LinkedListStack
{
   private Node first;

   /**
      Constructs an empty stack.
   */
   public LinkedListStack()
   {
      first = null;
   }

   /**
      Adds an element to the top of the stack.
      @param element the element to add
   */
   public void push(Object element)
   {
      Node newNode = new Node();
      newNode.data = element;
      newNode.next = first;
      first = newNode;
   }

   /**
      Removes the element from the top of the stack.
      @return the removed element
   */
   public Object pop()
   {
```

Stacks as Arrays

- A stack can be implemented as an array.
- Push and pop from the least expensive end - the back.

  The array must grow when it gets full.
- The `push` and `pop` operations are identical to the `addLast` and `removeLast`
- operations of an array list.
- `push` and `pop` are $O(1)+$ operations.

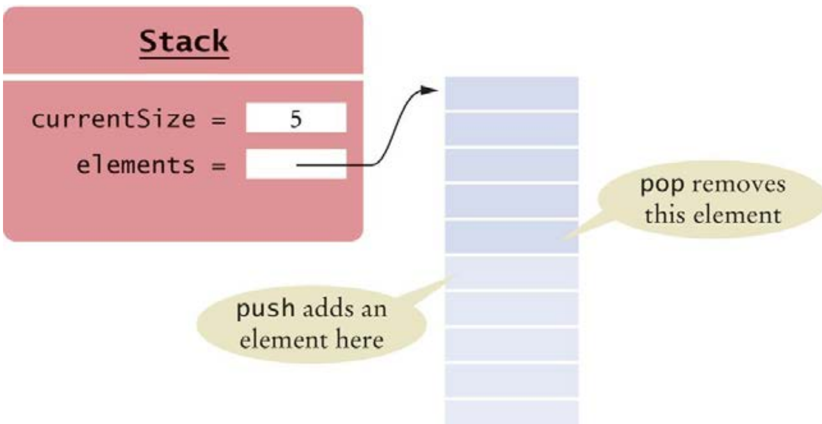

Figure 11 A Stack Implemented as an Array

Queues as Linked Lists

- A queue can be implemented as a linked list:
  - Add elements at the back.
  - Remove elements at the front.
  - Keep a reference to last element.
- The `add` and `remove` operations are $O(1)$ operations.

Queues as Linked Lists

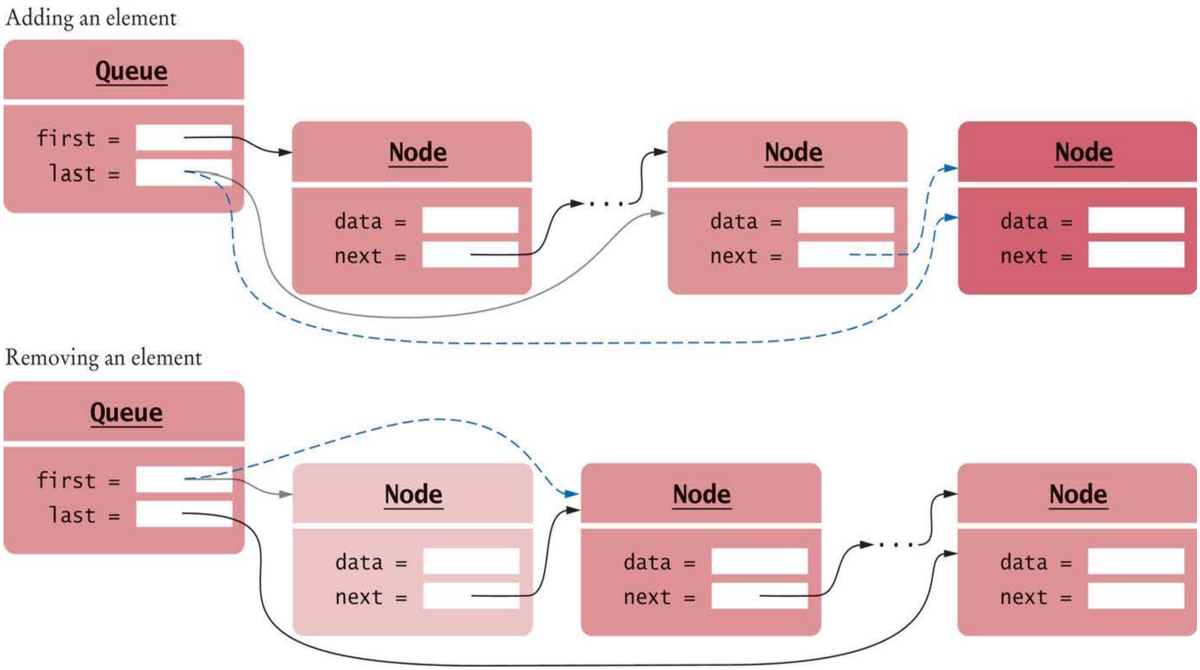

Figure 12 A Queue Implemented as a Linked List

Queues as Circular Arrays

- In a circular array, we wrap around to the beginning after the last element.

© ihsanyildizli/iStockphoto.

- When removing elements of a circular array,
  - increment the index at which the head of the queue is located.
- When the last element of the array is filled,
  - Wrap around and start storing at index 0
  - If elements have been removed there is room Else reallocate.
- All operations except reallocating are independent of the queue size
  - O(1)
- Reallocation is amortized constant time
  - O(1)+

Queues as Circular Arrays

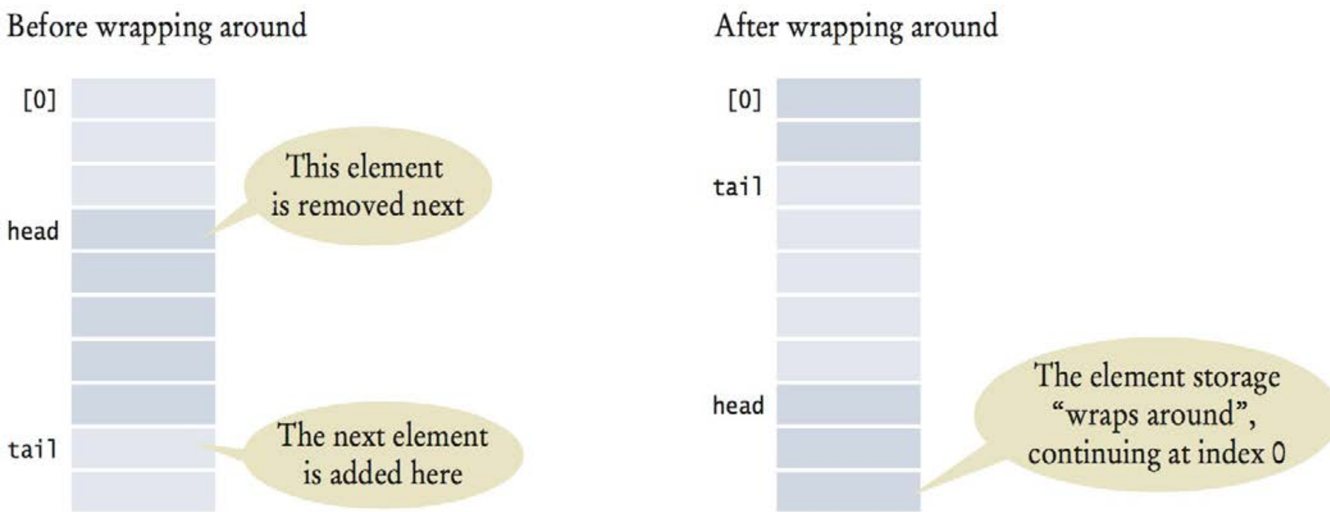

Figure 13 Queue Elements in a Circular Array

Queues as Circular Arrays

Table 3 Efficiency of Stack and Queue Operations

	Stack as Linked List	Stack as Array	Queue as Linked List	Queue as Circular Array
Add an element.	$O(1)$	$O(1)+$	$O(1)$	$O(1)+$
Remove an element.	$O(1)$	$O(1)+$	$O(1)$	$O(1)+$

section_3_4/CircularArrayQueue.java

```
import java.util.NoSuchElementException;

/**
   An implementation of a queue as a circular array.
*/
public class CircularArrayQueue
{
   private Object[] elements;
   private int currentSize;
   private int head;
   private int tail;

   /**
      Constructs an empty queue.
   */
   public CircularArrayQueue()
   {
      final int INITIAL_SIZE = 10;
      elements = new
      Object[INITIAL_SIZE];  currentSize
      = 0;
      head = 0;
      tail = 0;
   }
   /**
      Checks whether this queue is empty.
      @return true if this queue is empty
   */
   public boolean empty() { return currentSize == 0;
   }
   /**
      Adds an element to the tail of this queue.
      @param newElement the element to add
   */
   public void add(Object newElement)
```

Implementing a Hash Table

- In the Java library sets are implemented as hash sets and tree sets.
- **Hashing**: place items into an array at an index determined from the element.
- **Hash code**: an integer value that is computed from an object,
  - in such a way that different objects are likely to yield different hash codes.
- Collision: when two or more distinct objects have the same hash code. A good hash function minimizes collisions.
- A hash table uses the hash code to determine where to store each element.

Implementing a Hash Table

Table 4 Sample Strings and Their Hash Codes

String	Hash Code	String	Hash Code
"Adam"	2035631	"Juliet"	-2065036585
"Eve"	70068	"Katherine"	2079199209
"Harry"	69496448	"Sue"	83491
"Jim"	74478	"Ugh"	84982
"Joe"	74656	"VII"	84982

Hash Tables

- **Hash table**: An array that stores the set elements.
- **Hash code**: used as an array index into a hash table.
- Simplistic implementation
  - Very large array
  - Each object at its hashcode location
  - Simple to locate an element
  - But not practical

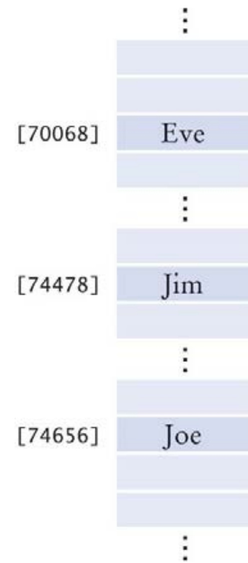

Figure 14 A Simplistic Implementation of a Hash Table

Hash Tables - Realistic Implementation

- A reasonable size array.
- Use the remainder operator to calculate the position.

```
int h = x.hashCode();
if (h < 0) { h = -h; }
position = h % arrayLength;
```

- Use **separate chaining** to handle collisions:
  - All colliding elements are collected in a linked list of elements with the same position value.
  - The lists are called buckets.
- Each entry of the hash table points to a sequence of nodes containing elements with the same hash code.
- A hash table can be implemented as an array of buckets—sequences of nodes that hold elements with the same hash code.

Hash Tables - Realistic Implementation

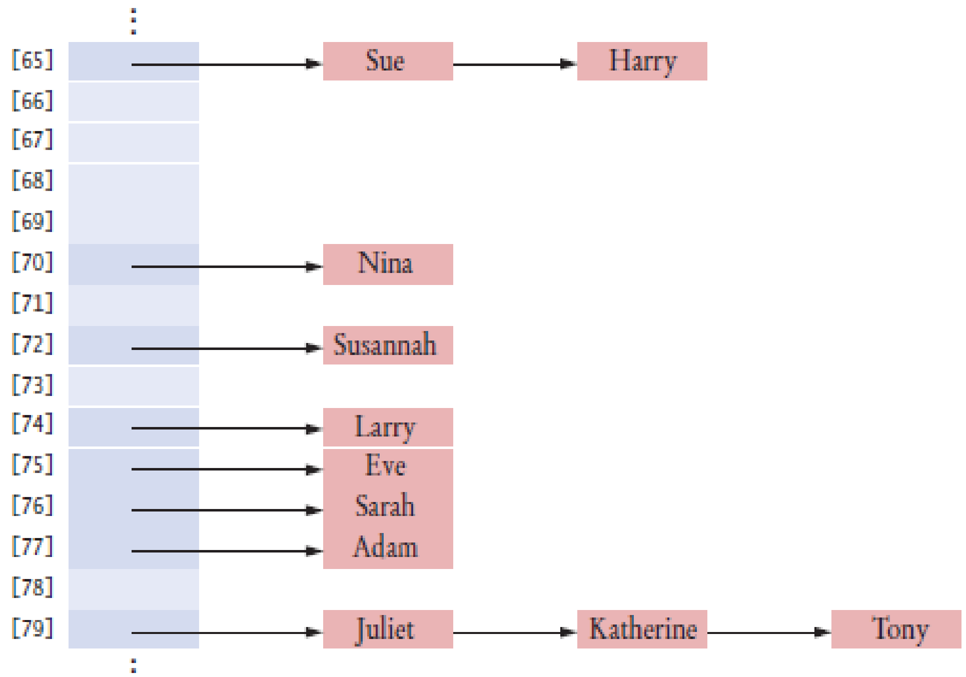

Figure 15 A Hash Table with Buckets to Store Elements with the Same Hash Code

Hash Tables

Elements with the same hash code are placed in the same bucket.

Implementing a Hash Table - Finding an Element

- Algorithm to find an element, `obj`
  - Compute the hash code and compress it.
    - Gives an index h into the hash table.
  - Iterate through the elements of the bucket at position h.
    - Check element is equal to `obj`.
  - If a match is found among the elements of that bucket,
    - `obj` is in the set. Otherwise, it is not.
- If there are no or only a few collision:
  - adding, locating, and removing hash table elements takes O(1) time.

Adding and Removing Elements

- Algorithm to add an element:
  - Compute the compressed hash code h.
  - Iterate through the elements of the bucket at position h.
    - For each element of the bucket, check whether it is equal to `obj`.
  - If a match is found among the elements of that bucket, then exit.
  - Otherwise, add a node containing obj to the beginning of the node sequence. If the load factor exceeds a fixed threshold, reallocate the table.
- Load factor: a measure of how full the table is.
  - The number of elements in the table divided by the table length.
- Adding an element to a hash table is $O(1)+$

Adding and Removing Elements

- Algorithm to remove an element:
  - Compute the hash code to find the bucket that should contain the object.
  - Try to find the element.
  - If it is present:
    - o remove it.
    - o otherwise, do nothing.
  - Shrink the table if it becomes too sparse.
- Removing an element from a hash table is $O(1)+$

Iterating over a Hash Table

- When iterator points to the middle of a node chain,
  - easy to get the next element.
- When the iterator is at the end of a node chain,
  - Skip over empty buckets.
  - Advance the iterator to the first node of the first non-empty bucket.
- Iterator needs to store the bucket number and a reference to the current node in the node chain.

```
if (current != null && current.next != null) {
  current = current.next; // Move to next element in bucket
}
else // Move to next bucket
{
  do
  {
    bucketIndex++;
    if (bucketIndex == buckets.length)
    { throw new NoSuchElementException(); }
    current = buckets[bucketIndex];
  }
  while (current == null);
}
```

Iterating over a Hash Table

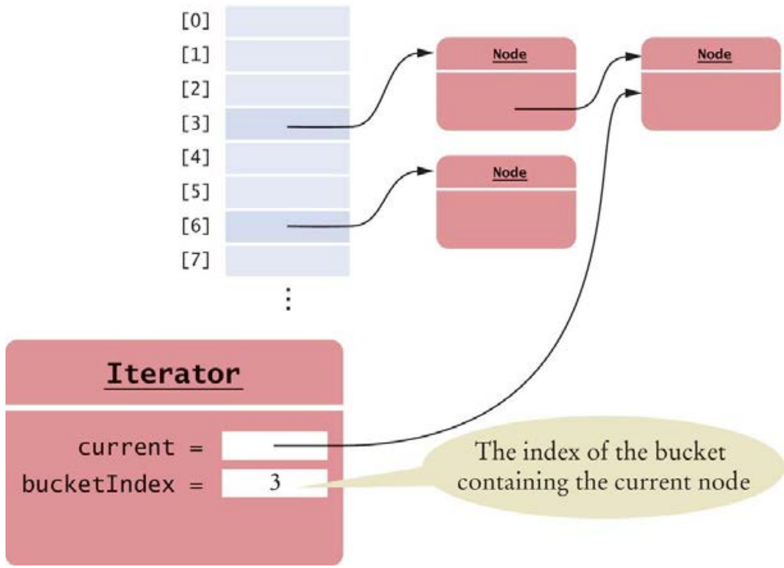

Figure 16 An Iterator to a Hash Table

Hash Table Efficiency

- The cost of iterating over all elements of a hash table:
  - Is proportional to the table length
  - Not the number of elements in the table
- Shrink the table when the load factor gets too small.

  One iteration is $O(1)$.
- Iterating over the entire table is $O(n)$.

Table 5 Hash Table Efficiency

Operation	Hash Table
Find an element.	$O(1)$
Add/remove an element.	$O(1)+$
Iterate through all elements.	$O(n)$

section_4/HashSet.java

```
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
    This class implements a hash set using separate chaining.
*/
public class HashSet
{
   private Node[] buckets;
   private int currentSize;

   /**
      Constructs a hash table.
      @param bucketsLength the length of the buckets array
   */
   public HashSet(int bucketsLength)
   {
      buckets = new Node[bucketsLength];
      currentSize = 0;
   }

   /**
      Tests for set membership.
      @param x an object
      @return true if x is an element of this set
   */
   public boolean contains(Object x)
   {
      int h = x.hashCode();
      if (h < 0) { h = -h; }
      h = h % buckets.length;

      Node current = buckets[h];
      while (current != null)
      {
```

section_4/HashSetDemo.java

```
import java.util.Iterator;

/**
   This program demonstrates the hash set class.
*/
public class HashSetDemo
{
   public static void main(String[] args)
   {
      HashSet names = new HashSet(101);


      names.add("Harry");
      names.add("Sue");
      names.add("Nina");
      names.add("Susannah")
      ;
      names.add("Larry");
      names.add("Eve");
      names.add("Sarah");
      names.add("Adam");
      names.add("Tony");
      names.add("Katherine")
      ;
      names.add("Juliet");
      names.add("Romeo");
      names.remove("Romeo");
      names.remove("George")
      ;
      Iterator iter = names.iterator();
      while (iter.hasNext())
      {
         System.out.println(iter.next())
         ;
}  }  }
```

Program Run:

```
Harry
Sue
Nina
Susannah
Larry
Eve
Sarah
Adam
Juliet
Katherine
Tony
```