Chapter Goals

• To learn how to use the linked lists provided in the standard library
• To be able to use iterators to traverse linked lists
• To understand the implementation of linked lists
• To distinguish between abstract and concrete data types
• To know the efficiency of fundamental operations of lists and arrays
• To become familiar with the stack and queue types
Using Linked Lists

• A linked list consists of a number of nodes, each of which has a reference to the next node

• Adding and removing elements in the middle of a linked list is efficient

• Visiting the elements of a linked list in sequential order is efficient

• Random access is not efficient
Inserting an Element into a Linked List

Figure 1  Inserting an Element into a Linked List
Java's LinkedList class

• Generic class
  • Specify type of elements in angle brackets: LinkedList<Product>

• Package: java.util

• Easy access to first and last elements with methods
  void addFirst(E obj)
  void addLast(E obj)
  E getFirst()
  E getLast()
  E removeFirst()
  E removeLast()
List Iterator

- ListIterator type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access
A List Iterator

Figure 2  A List Iterator
A Conceptual View of the List Iterator

Initial ListIterator position

After calling next

After inserting J

Figure 3   A Conceptual View of the List Iterator
List Iterator

- Think of an iterator as pointing between two elements
  - Analogy: like the cursor in a word processor points between two characters

- The `listIterator` method of the `LinkedList` class gets a list iterator

```java
LinkedList<String> employeeNames = ...;
ListIterator<String> iterator = employeeNames.listIterator();
```
List Iterator

• Initially, the iterator points before the first element
• The `next` method moves the iterator
  ```java
  iterator.next();
  ```
• `next` throws a `NoSuchElementException` if you are already past the end of the list
• `hasNext` returns true if there is a next element
  ```java
  if (iterator.hasNext())
  iterator.next();
  ```
List Iterator

• The `next` method returns the element that the iterator is passing

```java
while (iterator.hasNext()) {
    String name = iterator.next();
    Do something with name
}
```

• Shorthand:

```java
for (String name : employeeNames) {
    Do something with name
}
```

Behind the scenes, the for loop uses an iterator to visit all list elements
List Iterator

- LinkedList is a *doubly linked list*
  - Class stores *two links*:
    - One to the next element, and
    - One to the previous element

- To move the list position backwards, use:
  - hasPrevious
  - previous
Adding and Removing from a LinkedList

• The **add** method:
  - *Adds an object after the iterator*
  - *Moves the iterator position past the new element*

```java
iterator.add("Juliet");
```
Adding and Removing from a LinkedList

• The \texttt{remove} method
  • \textit{Removes and} \textit{Returns the object that was returned by the last call to next or previous}

  //Remove all names that fulfill a certain condition
  while (iterator.hasNext())
  {
    String name = iterator.next();
    if (name fulfills condition)
      iterator.remove();
  }

• Be careful when calling \texttt{remove}:
  • \textit{It can be called only once after calling next or previous}
  • \textit{You cannot call it immediately after a call to add}
  • \textit{If you call it improperly, it throws an IllegalStateException}
Implementing Linked Lists

• Previous section: Java's LinkedList class

• Now, we will look at the implementation of a simplified version of this class

• It will show you how the list operations manipulate the links as the list is modified

• To keep it simple, we will implement a singly linked list
  • Class will supply direct access only to the first list element, not the last one

• Our list will not use a type parameter
  • Store raw Object values and insert casts when retrieving them
Implementing Linked Lists

- **Node**: stores an object and a reference to the next node

- Methods of linked list class and iterator class have frequent access to the `Node` instance variables

- To make it easier to use:
  - *We do not make the instance variables private*
  - *We make Node a private inner class of LinkedList*
  - *It is safe to leave the instance variables public*
    - None of the list methods returns a `Node` object
Implementing Linked Lists

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

Implementing Linked Lists

- LinkedList class
  - Holds a reference *first* to the first node
  - Has a method to get the first element
public class LinkedList
{
    public LinkedList()
    {
        first = null;
    }
    public Object getFirst()
    {
        if (first == null)
            throw new NoSuchElementException();
        return first.data;
    }
    ...
    private Node first;
}
Adding a New First Element

• When a new node is added to the list
  • *It becomes the head of the list*
  • *The old list head becomes its next node*
### Adding a New First Element

```java
public void addFirst(Object obj) {
    Node newNode = new Node();  // 1
    newNode.data = obj;
    newNode.next = first;
    first = newNode;
}
```

**Figure 4** Adding a Node to the Head of a Linked List
Adding a New First Element

```java
public void addFirst(Object obj) {
    Node newNode = new Node();
    newNode.data = obj;
    newNode.next = first;
    first = newNode;
}
```

**Figure 4** Adding a Node to the Head of a Linked List
Adding a New First Element

```java
public void addFirst(Object obj) {
    Node newNode = new Node();
    newNode.data = obj;
    newNode.next = first;
    first = newNode;
}
```

**Figure 4** Adding a Node to the Head of a Linked List
Removing the First Element

• When the first element is removed
  • 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 will be garbage collected when there are no further references to it
Removing the First Element

public Object removeFirst()
{
    if (first == null)
        throw new NoSuchElementException();
    Object obj = first.data;
    first = first.next;
    return obj;
}

Figure 5 Removing the First Node from a Linked List
Linked List Iterator

- We define `LinkedListIterator`: private inner class of `LinkedList`
- Implements a simplified `ListIterator` interface
- Has access to the `first` field and private `Node` class
- Clients of `LinkedList` don't actually know the name of the iterator class
  - *They only know it is a class that implements the `ListIterator` interface*
LinkedListIterator

The LinkedListIterator class

```java
public class LinkedList {
    . . .
    public ListIterator listIterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements ListIterator {
        public LinkedListIterator() {
            position = null;
            previous = null;
        }
```

Continued
... 

private Node position;
private Node previous;
}
...
}
The Linked List Iterator's `next` Method

- **position**: reference to the last visited node
- Also, store a reference to the last reference before that
- `next` method: `position` reference is advanced to `position.next`
- Old position is remembered in `previous`
- If the iterator points before the first element of the list, then the old `position` is null and `position` must be set to `first`
public Object next()
{
    if (!hasNext())
        throw new NoSuchElementException();

    previous = position; // Remember for remove
    if (position == null)
        position = first;
    else position = position.next;

    return position.data;
}
The Linked List Iterator's `hasNext` Method

- The `next` method should only be called when the iterator is not at the end of the list

- The iterator is at the end
  - *if the list is empty* (`first == null`)
  - *if there is no element after the current position* (`position.next == null`)
private class LinkedListIterator implements ListIterator {
    
    public boolean hasNext() {
        if (position == null)
            return first != null;
        else
            return position.next != null;
    }
    
    . . .
}
The Linked List Iterator's `remove` Method

- If the element to be removed is the first element, call `removeFirst`

- Otherwise, the node preceding the element to be removed needs to have its `next` reference updated to skip the removed element

- If the previous reference equals `position`:
  - *this call does not immediately follow a call to next*
  - *throw an IllegalArgumentException*

- It is illegal to call `remove` twice in a row
  - *remove sets the previous reference to position*
The Linked List Iterator's \texttt{remove} Method

public void remove()
{
    if (previous == position)
        throw new IllegalStateException();
    if (position == first)
    {
        removeFirst();
    }
    else
    {
        previous.next = position.next;  \footnote{1}
    }
    position = previous;
}
The Linked List Iterator's `remove` Method (cont.)

**Figure 6** Removing a Node from the Middle of a Linked List
public void remove()
{
    If (previous == position)
        throw new IllegalStateException();
    if (position == first)
    {
        removeFirst();
    }
    else
    {
        previous.next = position.next; ①
    }
    position = previous; ②
}
The Linked List Iterator's \texttt{remove} Method (cont.)

\textbf{Figure 6} Removing a Node from the Middle of a Linked List
The Linked List Iterator's *set* Method

- Changes the data stored in the previously visited element
- The *set* method

```java
public void set(Object obj) {
    if (position == null)
        throw new NoSuchElementException();
    position.data = obj;
}
```
The Linked List Iterator's add Method

• The most complex operation is the addition of a node
  • *add inserts the new node after the current position*
  • *Sets the successor of the new node to the successor of the current position*
The Linked List Iterator's `add` Method

```java
public void add(Object obj)
{
    if (position == null)
    {
        addFirst(obj);
        position = first;
    }
    else
    {
        Node newNode = new Node();
        newNode.data = obj;
        newNode.next = position.next;  // 1
        position.next = newNode;
        position = newNode;
    }
    previous = position;
}
```

Continued
Figure 7  Adding a Node to the Middle of a Linked List
The Linked List Iterator's `add` Method

```java
public void add(Object obj)
{
    if (position == null)
    {
        addFirst(obj);
        position = first;
    }
    else
    {
        Node newNode = new Node();
        newNode.data = obj;
        newNode.next = position.next;  // 1
        position.next = newNode;       // 2
        position = newNode;
    }
    previous = position;
}
```

*Continued*
The Linked List Iterator's `add` Method (cont.)

**Figure 7** Adding a Node to the Middle of a Linked List

*Big Java* by Cay Horstmann
Copyright © 2008 by John Wiley & Sons. All rights reserved.
The Linked List Iterator's `add` Method

```java
public void add(Object obj) {
    if (position == null) {
        addFirst(obj);
        position = first;
    } else {
        Node newNode = new Node();
        newNode.data = obj;
        newNode.next = position.next;  // 1
        position.next = newNode;       // 2
        position = newNode;            // 3
        previous = position;
    }
}
Continued
```
The Linked List Iterator's `add` Method (cont.)

**Figure 7** Adding a Node to the Middle of a Linked List
Abstract and Concrete Data Types

• There are two ways of looking at a linked list
  • To think of the concrete implementation of such a list
    o Sequence of node objects with links between them
  • Think of the abstract concept of the linked list
    o Ordered sequence of data items that can be traversed with an iterator
Abstract and Concrete Data Types

Figure 8  A Concrete View of a Linked List

Figure 9  An Abstract View of a Linked List
Abstract Data Types

• Define the fundamental operations on the data
• Do not specify an implementation
Abstract and Concrete Array Type

• As with a linked list, there are two ways of looking at an array list

• Concrete implementation: a partially filled array of object references

• We don't usually think about the concrete implementation when using an array list
  • *We take the abstract point of view*

• Abstract view: ordered sequence of data items, each of which can be accessed by an integer index
Abstract and Concrete Data Types

Figure 10  A Concrete View of an Array List

Figure 11  An Abstract View of an Array List
Abstract and Concrete Data Types

• Concrete implementations of a linked list and an array list are quite different

• The abstractions seem to be similar at first glance

• To see the difference, consider the public interfaces stripped down to their minimal essentials
Fundamental Operations on Array List

public class ArrayList
{
    public Object get(int index) {. . .}  
    public void set(int index, Object value) {. . .}
    . . .
}

Fundamental Operations on Linked List

public class LinkedList
{
   public ListIterator listIterator() {}
   . . .
}

public interface ListIterator
{
   Object next();
   boolean hasNext();
   void add(Object value);
   void remove();
   void set(Object value);
   . . .
}
Efficiency of Operations for Arrays and Lists

- Adding or removing an element
  - A fixed number of node references need to be modified to add or remove a node, regardless of the size of the list
  - In big-Oh notation: $O(1)$

- Adding or removing an element
  - On average $n/2$ elements need to be moved
  - In big-Oh notation: $O(n)$
### Efficiency of Operations for Arrays and Lists

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random access</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Linear traversal step</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Add/remove an element</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Abstract Data Types

- Abstract list
  - Ordered sequence of items that can be traversed sequentially
  - Allows for insertion and removal of elements at any position

- Abstract array
  - Ordered sequence of items with random access via an integer index
Self Check 15.6

What is the advantage of viewing a type abstractly?

**Answer:** You can focus on the essential characteristics of the data type without being distracted by implementation details.
Stacks and Queues

- Stack: collection of items with "last in first out" retrieval
- Queue: collection of items with "first in first out" retrieval
Stack

- Allows insertion and removal of elements only at one end
  - *Traditionally called the top of the stack*
- New items are added to the top of the stack
- Items are removed at the top of the stack
- Called *last in, first out* or LIFO order
- Traditionally, addition and removal operations are called push and pop
- Think of a stack of books
Queue

- Add items to one end of the queue (the tail)
- Remove items from the other end of the queue (the head)
- Queues store items in a first in, first out or FIFO fashion
- Items are removed in the same order in which they have been added
- Think of people lining up
  - *People join the tail of the queue and wait until they have reached the head of the queue*
Stacks and Queues: Uses in Computer Science

• Queue
  • Event queue of all events, kept by the Java GUI system
  • Queue of print jobs

• Stack
  • Run-time stack that a processor or virtual machine keeps to organize the variables of nested methods