Data Structures
Lesson 4

BSc in Computer Science
University of New York, Tirana

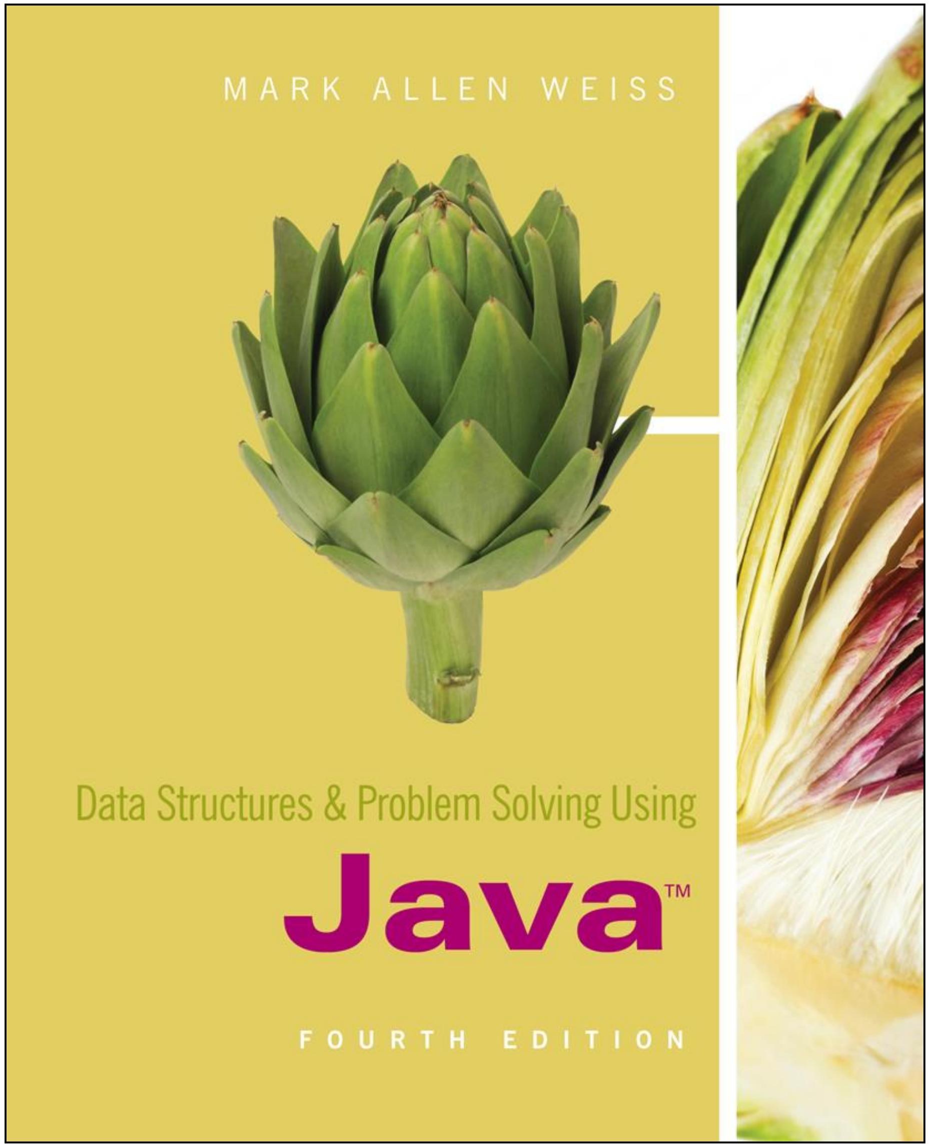
Assoc. Prof. Marenglen Biba

Outline

- Doubly Linked Lists
- Circular Doubly Linked Lists
- Exercises

Chapter 17

Linked Lists



Addison-Wesley is an imprint of PEARSON

Doubly Linked Lists

- The singly linked list does not efficiently support some important operations.
- For instance, although it is easy to go to the front of the list, it is time consuming to go to the end.

A doubly linked list

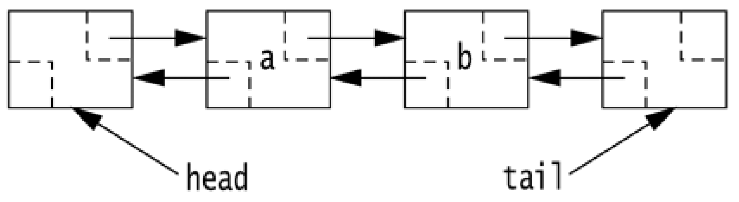


figure 17.15

A doubly linked list

Empty list

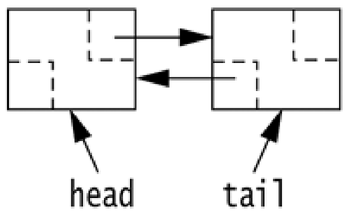


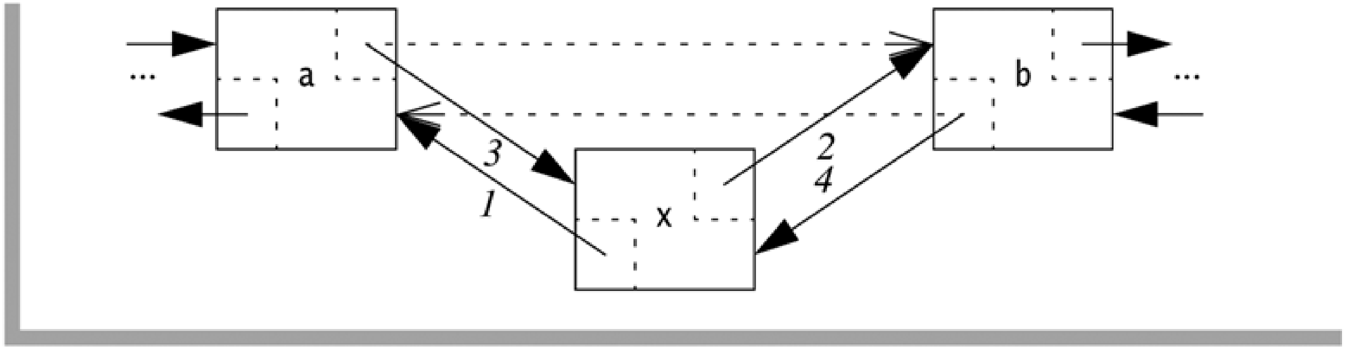
figure 17.16

An empty doubly linked list

Insertion

figure 17.17

Insertion in a doubly linked list by getting new node and then changing pointers in the order indicated



Insert after current: current is node with a

```
newNode = new DoublyLinkedListNode( x );
newNode.prev = current; // Set x's prev link
newNode.next = current.next; // Set x's next link
newNode.prev.next = newNode; // Set a's next link
newNode.next.prev = newNode; // Set b's prev link
current = newNode;
```

Circularly and doubly linked list

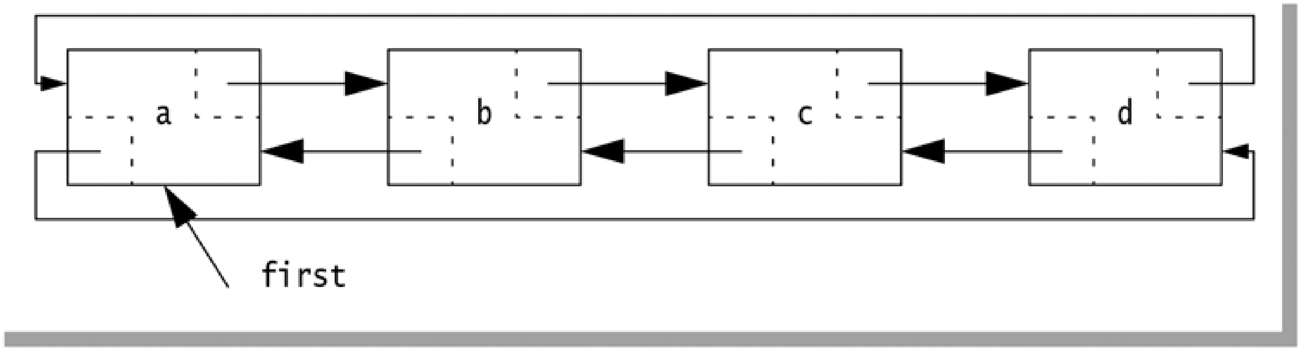


figure 17.18

A circularly and doubly linked list

Sorted linked lists

- Sometimes we want to keep the items in a linked list in sorted order, which we can do with a sorted linked list.
- The fundamental difference between a sorted linked list and an unsorted linked list is the insertion routine.
- Indeed, we can obtain a sorted list class by simply altering the insertion routine from our already written list class.

```java
package weiss.nonstandard;

// SortedLinkedList class
//
// CONSTRUCTION: with no initializer
// Access is via LinkedListIterator class
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )        --> Insert x
// void insert( x, p )     --> Insert x (ignore p)
// All other LinkedList operations
// ******************ERRORS********************************
// No special errors

public class SortedLinkedList<AnyType extends Comparable<? super AnyType>>
                              extends LinkedList<AnyType>
{
    /**
     * Insert after p.
     * @param x the item to insert.
     * @param p this parameter is ignored.
     */
    public void insert( AnyType x, LinkedListIterator<AnyType> p )
    {
        insert( x );
    }

    /**
     * Insert in sorted order.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
        LinkedListIterator<AnyType> prev = zeroth( );
        LinkedListIterator<AnyType> curr = first( );

        while( curr.isValid( ) && x.compareTo( curr.retrieve( ) ) > 0 )
        {
            prev.advance( );
            curr.advance( );
        }

        super.insert( x, prev );
    }
}
```

AnyType must implement the Comparable interface: compareTo() methods.

Insert right before larger element than x

figure 17.19

The `SortedLinkedList` class, in which insertions are restricted to sorted order

```
package weiss.util;
public class LinkedList<AnyType> extends AbstractCollection<AnyType>
                                implements List<AnyType>, Queue<AnyType>
{
    private static class Node<AnyType>
      { /* Figure 17.21 */ }
    private class LinkedListIterator<AnyType> implements ListIterator<AnyType>
      { /* Figure 17.30 */ }

    public LinkedList( )
      { /* Figure 17.22 */ }
    public LinkedList( Collection<AnyType> other )
      { /* Figure 17.22 */ }

    public int size( )
      { /* Figure 17.23 */ }
    public boolean contains( Object x )
      { /* Figure 17.23 */ }
    public boolean add( AnyType x )
      { /* Figure 17.24 */ }
    public void add( int idx, AnyType x )
      { /* Figure 17.24 */ }
    public void addFirst( AnyType x )
      { /* Figure 17.24 */ }
    public void addLast( AnyType x )
      { /* Figure 17.24 */ }
    public AnyType element( )
      { /* Added in Java 5; same as getFirst */ }
    public AnyType getFirst( )
      { /* Figure 17.25 */ }
    public AnyType getLast( )
      { /* Figure 17.25 */ }
```

figure 17.20a

Class skeleton for standard `LinkedList` class (*continues*)

figure 17.20b

Class skeleton for standard `LinkedList` class (*continued*)

```
    public AnyType remove( )
      { /* Added in Java 5; same as removeFirst */ }
    public AnyType removeFirst( )
      { /* Figure 17.27 */ }
    public AnyType removeLast( )
      { /* Figure 17.27 */ }
    public boolean remove( Object x )
      { /* Figure 17.28 */ }
    public AnyType get( int idx )
      { /* Figure 17.25 */ }
    public AnyType set( int idx, AnyType newVal )
      { /* Figure 17.25 */ }
    public AnyType remove( int idx )
      { /* Figure 17.27 */ }
    public void clear( )
      { /* Figure 17.22 */ }
    public Iterator<AnyType> iterator( )
      { /* Figure 17.29 */ }
    public ListIterator<AnyType> listIterator( int idx )
      { /* Figure 17.29 */ }

    private int theSize;
    private Node<AnyType> beginMarker;
    private Node<AnyType> endMarker;
    private int modCount = 0;

    private static final Node<AnyType> NOT_FOUND = null;
    private Node<AnyType> findPos( Object x )
      { /* Figure 17.23 */ }
    private AnyType remove( Node<AnyType> p )
      { /* Figure 17.27 */ }
    private Node<AnyType> getNode( int idx )
      { /* Figure 17.26 */}
}
```

Head
and tail

modCount is used
by the iterators to
determine if the list
has changed
while an iteration is

in progress;

```
/**
 * This is the doubly linked list node.
 */
private static class Node<AnyType>
{
    public Node( AnyType d, Node<AnyType> p, Node<AnyType> n )
    {
        data = d; prev = p; next = n;
    }

    public AnyType       data;
    public Node<AnyType> prev;
    public Node<AnyType> next;
}
```

figure 17.21

Node nested class for standard `LinkedList` class

```
    /**
     * Construct an empty LinkedList.
     */
    public LinkedList( )
    {
        clear( );
    }

    /**
     * Construct a LinkedList with same items as another Collection.
     */
    public LinkedList( Collection<AnyType> other )
    {
        clear( );
        for( AnyType val : other )
            add( val );
    }

    /**
     * Change the size of this collection to zero.
     */
    public void clear( )
    {
        beginMarker = new Node<AnyType>( null, null, null );
        endMarker = new Node<AnyType>( null, beginMarker, null );
        beginMarker.next = endMarker;

        theSize = 0;
        modCount++;
    }
```

figure 17.22

Constructors and `clear` method for standard `LinkedList` class

```
/**
 * Returns the number of items in this collection.
 * @return the number of items in this collection.
 */
public int size( )
{
    return theSize;
}

/**
 * Tests if some item is in this collection.
 * @param x any object.
 * @return true if this collection contains an item equal to x.
 */
public boolean contains( Object x )
{
    return findPos( x ) != NOT_FOUND;
}

/**
 * Returns the position of first item matching x
 * in this collection, or NOT_FOUND if not found.
 * @param x any object.
 * @return the position of first item matching x
 * in this collection, or NOT_FOUND if not found.
 */
private Node<AnyType> findPos( Object x )
{
    for( Node<AnyType> p = beginMarker.next; p != endMarker; p = p.next )
        if( x == null )
        {
            if( p.data == null )
                return p;
        }
        else if( x.equals( p.data ) )
            return p;

    return NOT_FOUND;
}
```

figure 17.23

`size` and `contains` for standard `LinkedList` class

```
/**
 * Adds an item to this collection, at the end.
 * @param x any object.
 * @return true.
 */
public boolean add( AnyType x )
{
    addLast( x );
    return true;
}

/**
 * Adds an item to this collection, at the front.
 * Other items are slid one position higher.
 * @param x any object.
 */
public void addFirst( AnyType x )
{
    add( 0, x );
}

/**
 * Adds an item to this collection, at the end.
 * @param x any object.
 */
public void addLast( AnyType x )
{
    add( size( ), x );
}

/**
 * Adds an item to this collection, at a specified position.
 * Items at or after that position are slid one position higher.
 * @param x any object.
 * @param idx position to add at.
 * @throws IndexOutOfBoundsException if idx is not
 *         between 0 and size(), inclusive.
 */
public void add( int idx, AnyType x )
{
    Node<AnyType> p = getNode( idx );
    Node<AnyType> newNode = new Node<AnyType>( x, p.prev, p );
    newNode.prev.next = newNode;
    p.prev = newNode;
    theSize++;
    modCount++;
}
```

figure 17.24

add methods for standard LinkedList class

x inserted right before node at idx

figure 17.25

get and set methods for standard LinkedList class

```
    /**
     * Returns the first item in the list.
     * @throws NoSuchElementException if the list is empty.
     */
    public AnyType getFirst( )
    {
        if( isEmpty( ) )
            throw new NoSuchElementException( );
        return getNode( 0 ).data;
    }

    /**
     * Returns the last item in the list.
     * @throws NoSuchElementException if the list is empty.
     */
    public AnyType getLast( )
    {
        if( isEmpty( ) )
            throw new NoSuchElementException( );
        return getNode( size( ) - 1 ).data;
    }

    /**
     * Returns the item at position idx.
     * @param idx the index to search in.
     * @throws IndexOutOfBoundsException if index is out of range.
     */
    public AnyType get( int idx )
    {
        return getNode( idx ).data;
    }

    /**
     * Changes the item at position idx.
     * @param idx the index to change.
     * @param newVal the new value.
     * @return the old value.
     * @throws IndexOutOfBoundsException if index is out of range.
     */
    public AnyType set( int idx, AnyType newVal )
    {
        Node<AnyType> p = getNode( idx );
        AnyType oldVal = p.data;

        p.data = newVal;
        return oldVal;
    }
```

```java
/**
 * Gets the Node at position idx, which must range from 0 to size( ).
 * @param idx index to search at.
 * @return internal node corresponding to idx.
 * @throws IndexOutOfBoundsException if idx is not
 *         between 0 and size(), inclusive.
 */
private Node<AnyType> getNode( int idx )
{
    Node<AnyType> p;

    if( idx < 0 || idx > size( ) )
        throw new IndexOutOfBoundsException( );

    if( idx < size( ) / 2 )
    {
        p = beginMarker.next;
        for( int i = 0; i < idx; i++ )
            p = p.next;
    }
    else
    {
        p = endMarker;
        for( int i = size( ); i > idx; i-- )
            p = p.prev;
    }

    return p;
}
```

Starts from head

Starts from tail

figure 17.26

Private getNode for standard LinkedList class

figure 17.27

`remove` methods for standard `LinkedList` class

```
/**
 * Removes the first item in the list.
 * @return the item was removed from the collection.
 * @throws NoSuchElementException if the list is empty.
 */
public AnyType removeFirst( )
{
    if( isEmpty( ) )
        throw new NoSuchElementException( );
    return remove( getNode( 0 ) );
}

/**
 * Removes the last item in the list.
 * @return the item was removed from the collection.
 * @throws NoSuchElementException if the list is empty.
 */
public AnyType removeLast( )
{
    if( isEmpty( ) )
        throw new NoSuchElementException( );
    return remove( getNode( size( ) - 1 ) );
}

/**
 * Removes an item from this collection.
 * @param idx the index of the object.
 * @return the item that was removed from the collection.
 */
public AnyType remove( int idx )
{
    return remove( getNode( idx ) );
}

/**
 * Removes the object contained in Node p.
 * @param p the Node containing the object.
 * @return the item that was removed from the collection.
 */
private AnyType remove( Node<AnyType> p )
{
    p.next.prev = p.prev;
    p.prev.next = p.next;
    theSize--;
    modCount++;

    return p.data;
}
```

```
/**
 * Removes an item from this collection.
 * @param x any object.
 * @return true if this item was removed from the collection.
 */
public boolean remove( Object x )
{
    Node<AnyType> pos = findPos( x );

    if( pos == NOT_FOUND )
        return false;
    else
    {
        remove( pos );
        return true;
    }
}
```

figure 17.28

Additional `remove` method for standard `LinkedList` class

Summary: Key Concepts

- Circularly linked list: A linked list in which the last cell's next link references first.
- Doubly linked list: A linked list that allows bidirectional traversal by storing two links per node.
- Header node: An extra node in a linked list that holds no data but serves to satisfy the requirement that every node have a previous node. A header node allows us to avoid special cases such as the insertion of a new first element and the removal of the first element.
- Iterator class: A class that maintains a current position in a container, such as a list. An iterator class is usually in the same package as, or an inner class of, a list class.
- Sorted linked list: A list in which items are in sorted order. A sorted linked list class can be derived from a list class.

Exercises

PrintInverse( )

- Print the LinkedList in reverse order

Find number of occurrences of an element in a list

- Scan the list and update a counter every time you find an element in the list
- Use comparator

Efficiently find an element in a sorted list

- Hint: If the book title starts with a letter greater than the middle of the alphabet then start searching from tail.

- Even better: binary search

MoveToFront

MoveToFront

- If the order that items in a list are stored is not important, you can frequently speed searching with the heuristic known as move to front:
- Whenever an item is accessed, move it to the front of the list.
- This action usually results in an improvement because frequently accessed items tend to migrate toward the front of the list, whereas less frequently accessed items tend to migrate toward the end of the list.
- Consequently, the most frequently accessed items tend to require the least searching. Implement the move-to-front heuristic for linked lists.

Remove Duplicates

- Remove duplicate elements from the list

Project: Part 1

- Write a line-based text editor.
- The command syntax is similar to the Unix line editor ed.
- The internal copy of the file is maintained as a linked list of lines.
- To be able to go up and down in the file, you have to maintain a doubly linked list. Most commands are represented by a one-character string.
- Some are two characters and require an argument (or two).

figure 17.31

Commands for editor in Exercise 17.19

Command	Function
`1`	Go to the top.
`a`	Add text after current line until `.` on its own line
`d`	Delete current line.
`dr num num`	Delete several lines.
`f name`	Change name of the current file (for next write).
`g num`	Go to a numbered line.
`h`	Get help.
`i`	Like append, but add lines before current line.
`m num`	Move current line after some other line.
`mr num num num`	Move several lines as a unit after some other line.
`n`	Toggle whether line numbers are displayed.
`p`	Print current line.
`pr num num`	Print several lines.
`q!`	Abort without write.
`r name`	Read and paste another file into the current file.
`s text text`	Substitute text with other text.
`t num`	Copy current line to after some other line.
`tr num num num`	Copy several lines to after some other line.
`w`	Write file to disk.
`x!`	Exit with write.
`$`	Go to the last line.
`-`	Go up one line.
`+`	Go down one line.
`=`	Print current line number.
`/ text`	Search forward for a pattern.
`? text`	Search backward for a pattern.
`#`	Print number of lines and characters in file.

Project Part 1 Deadline

- Zip the project into a .zip document
- Rename the file in “Name Surname Exercise Number .zip”
- Submit the file by email to: marenglenbiba@unyt.edu.al.
- Mail Subject: DSSPRING16 – Project Part 1 - Name Surname
- Deadline 01/06/2017 23:59.
- 20% penalty if the above rules are not respected as written.
- 20% penalty for each day of delay