Compression

Assignment 4
Due Sunday, February 24, 2013
CSC 062: Spring, 2013

Compression

1 Compressing Information

Sometimes we need to store massive amounts of information about an object. A good example is storing graphic images. To save space on disks and in transmission of information across the internet, researchers have designed algorithms to compress data. In this assignment you will learn one of these compression techniques.

A graphic image can be represented by a two dimensional array of information about the colors of various picture elements (or pixels). At high resolution the image may be composed of 1000 rows and 1000 columns of information, leading to the need to store information on 1,000,000 pixels per image. Needless to say this creates serious problems for storing and transmitting these images. However most images tend to have many contiguous groups of pixels, each of which are the same color. We can take advantage of this by trying to encode information about the entire block in a relatively efficient manner.

The basic idea of our encoding will be to represent a block of pixels with the same color by simply recording the first place where we encounter the new color and only recording information when we see a new color. For instance suppose we have the following table of information (where we will imagine each number represents a color):

2	2	2	3	3
2	3	3	3	3
3	1	1	1	3

If we imagine tracing through the table from left to right starting with the top row and going through successive rows then we notice that we only need to record the following entries:

2	-	-	3	-
2	3	-	-	-
-	1	-	-	3

Rather than recording this in a two-dimensional table, it will now generally be more efficient to keep this information in a linear list where it is assumed we sweep across an entire row before going on to the next:

(0,0)*2	(0,3)*3	(1,0)*2	(1,1)*3	(2,1)*1	(2,4)*3

In this program you are to design a class which will represent one of these compressed tables. You will do this in two stages:

1. First write a class `CurDoublyLinkedList` that extends `DoublyLinkedList` from Bailey's `structure5` package. (A listing of Bailey's `DoublyLinkedList` can be found at the end of this document.) This class will have extra capability to move to any desired element of the list (called the current element) and then either add a new node after this element or to remove the current element. Your new class should support all of the old methods of the Lists as well as `first()`, `last()`, `next()`, `back()`, `isOffRight()`, `isOffLeft()`, `getCurrent()`, `addAfterCurrent(E value)`, and `deleteCurrent()`. Specifications for these methods can be found in the startup code available on-line.

2. Use this to design the class `CompressedTable` implementing the interface `TwoDTable`. The two key public methods of this class are `updateInfo(row,col,newInfo)` and `getInfo(row,col)`. Again, specifications can be found in the attached code. This class will have an instance variable of type `CurDoublyLinkedList` (as well as other instance variables).

We have provided a class `GridTest` that will use your classes to create an application that allows the user to manipulate a grid of rectangles that can form an image. The user interacts with the application by clicking on a color button to set the current color and then clicking on rectangles in the grid to change the colors of individual rectangles. Along with the color buttons there is a button that will display the results of sending the `toString` method to the object of type CompressedTable to show the current state of the representation.

A working demo of this program can be found at
`http://www.cs.pomona.edu/classes/cs062/assignments/CompressedGrid/CompressedGrid.html`.

Create a new project in `Eclipse` with an appropriate package and copy the files over from `/common/cs/cs062/assignments/assignment4/` into the source directory of your newly created package directory. Don't forget to add the `package` command to your files.

We have provided you with a lot of code here, but you will find that much of the omitted code is quite tricky. This project will require you to be very careful in developing the code for the methods. Look carefully at the provided code and design your methods very carefully. In particular, be sure to test your code carefully *as it is developed* as you will likely make several logical errors if you are not extremely careful. Do not attempt to run the `main` program in `GridTest` until you have first carefully tested `CurDoublyLinkedList` and then `CompressedTable`. (The `main method provided includes tests of some methods, but not other. Add your own tests to make sure every method is exercised!`)

The `updateInfo` method of `CompressedTable` is probably the trickiest code to write. Here is a brief outline of the logic.

1. We have provided you with code to find the node of the list that encodes the position being updated. Of course not every position is in the list, only those representing changes to the array. If the node is not there, the method returns the node before the given position in the list. The class `RowOrderedPosn` (see the startup code) not only encodes a position, but, because it also contains information on the number of rows and columns in the table, can determine if one position would come before or after another.

2. If the new information in the table is the same as that in the node found in step 1, then nothing needs to be done. Otherwise determine if the node represents exactly the position being updated.

   If it is the same, update the value of the node, otherwise add a new node representing the new position

3. If you are not careful you may accidentally change several positions in the table to the new value. Avoid this by considering putting in a new node representing the position immediately after the position with the new value. (Draw pictures of the list so you can see what is happening!)

4. If there is already a node with this successor position then nothing needs to be done. Otherwise add a new node with the successor position and the original value. (Do you see why this is necessary? Look at the demo program to see why.)

Try this sketch of an algorithm out with several sample lists so that you can understand how it works!

Extra credit for design!!

This is your most complex program yet. As a result you should make a very complete design for your program before you sit down at a computer to program. If you email your design for the methods `removeFirst`, `removeLast`, and `removeCurrent` of `CurDoublyLinkedList` and the method `updateInfo` of `CompressedTable` to us at `kim@cs.pomona.edu` by Thursday at midnight, we will take a look at them and provide you with feedback on them. (Keep in mind that methods in `CurDoublyLinkedList` can make calls to `super` to access methods in `DoublyLinkedList` and then add your own behavior.)

While these designs should not be written in Java, you should include the complete logic of the methods. Remember that you must draw pictures and look at all possible special cases in order to get these right. You can get up to two points extra credit for submitting these via e-mail by the deadline.

More Extra Credit

As you add more information to the table, you will notice that the table is no longer as efficient in space, because several consecutive entries may have the same values. Make the representation more efficient by dropping later values if they can be subsumed by earlier ones.

For example, the list

(0,0)*2	(0,3)*3	(1,0)*3	(1,1)*3	(2,1)*1	(2,4)*3

can be replaced by the much simpler list:

(0,0)*2	(0,3)*3	(2,1)*1	(2,4)*3

For extra credit, modify the `updateInfo` method of `CompressedTable` to eliminate consecutive items with the same value. The amount of extra credit received will be proportional to the efficiency of your algorithm. Ideally this optimization will only take O(1) time each time something is inserted in the table.

What to hand in

As usual, export the entire folder from Eclipse to your desktop, change the name of the folder to something of the form "lastname-Assignment4" (where you should replace *lastname* by your last name). Make sure the folder contains both your .java and .class files. Then drag it into the dropoff folder in `/common/cs/cs062/dropbox`, where an alias for the cs062 folder should be sitting on your desktop already. Be sure that your code is clear, formatted properly and commented appropriately (using Javadoc... see the first assignment for details on what's expected for comments).

DoublyLinkedList class from structure5

```
// Implementation of lists, using doubly linked elements.
// (c) 1998, 2001 duane a. bailey
package structure5;
import java.util.Iterator;
/**
 * An implementation of lists using doubly linked elements, similar to that of
 * {@link java.util.LinkedList java.util.LinkedList}.
 * This class is a basic implementation of the {@link List} interface.
 * Operations accessing or modifying either the head or the tail of
 * the list execute in constant time.
 * Doubly linked lists are less space-efficient than singly linked lists,
 * but tail-related operations are less costly.
 * Example usage:
 *
 * To place a copy of every unique parameter passed to a program into a
 * DoublyLinkedList,  we would use the following:
 * <pre>
 * public static void main({@link java.lang.String String[]} arguments)
 * {
 *     {@link DoublyLinkedList} argList = new {@link #DoublyLinkedList()};
 *     for (int i = 0; i < arguments.length; i++){
 *         if (!argList.{@link #contains(Object) contains(arguments[i])}){
 *             argList.{@link #add(Object) add(arguments[i])};
 *         }
 *    }
 *    System.out.println(argList);
 * }
 * </pre>
 * @version $Id: DoublyLinkedList.java 31 2007-08-06 17:19:56Z bailey $
 * @author, 2001 duane a. bailey
 */
public class DoublyLinkedList<E> extends AbstractList<E>
{
    /**
     * Number of elements within list.
     */
    protected int count;
    /**
     * Reference to head of list.
     */
    protected DoublyLinkedNode<E> head;
    /**
     * Reference to tail of list.
     */
    protected DoublyLinkedNode<E> tail;

    /**
     * Constructs an empty list.
     *
     * @post constructs an empty list
     *
     */
```

```
public DoublyLinkedList()
{
    head = null;
    tail = null;
    count = 0;
}

/**
 * Add a value to head of list.
 *
 * @post adds value to beginning of list
 *
 * @param value value to be added.
 */
public void add(E value)
{
    addFirst(value);
}

/**
 * Add a value to head of list.
 *
 * @pre value is not null
 * @post adds element to head of list
 *
 * @param value value to be added.
 */
public void addFirst(E value)
{
    // construct a new element, making it head
    head = new DoublyLinkedNode<E>(value, head, null);
    // fix tail, if necessary
    if (tail == null) tail = head;
    count++;
}

/**
 * Remove a value from head of list.
 * Value is returned.
 *
 * @pre list is not empty
 * @post removes first value from list
 *
 * @return value removed from list.
 */
public E removeFirst()
{
    Assert.pre(!isEmpty(),"List is not empty.");
    DoublyLinkedNode<E> temp = head;
    head = head.next();
    if (head != null) {
        head.setPrevious(null);
    } else {
        tail = null; // remove final value
```

```
    }
    temp.setNext(null);// helps clean things up; temp is free
    count--;
    return temp.value();
}

/**
 * Add a value to tail of list.
 *
 * @pre value is not null
 * @post adds new value to tail of list
 *
 * @param value value to be added.
 */
public void addLast(E value)
{
    // construct new element
    tail = new DoublyLinkedNode<E>(value, null, tail);
    // fix up head
    if (head == null) head = tail;
    count++;
}

/**
 * Remove a value from tail of list.
 *
 * @pre list is not empty
 * @post removes value from tail of list
 *
 * @return value removed from list.
 */
public E removeLast()
{
    Assert.pre(!isEmpty(),"List is not empty.");
    DoublyLinkedNode<E> temp = tail;
    tail = tail.previous();
    if (tail == null) {
        head = null;
    } else {
        tail.setNext(null);
    }
    count--;
    return temp.value();
}
/*
public void addLast(E value)
{
    // construct new element
    tail = new DoublyLinkedNode<E>(value, null, tail);
    count++;
}

public E removeLast()
{
```

```
    Assert.pre(!isEmpty(),"List is not empty.");
    DoublyLinkedNode<E> temp = tail;
    tail = tail.previous();
    tail.setNext(null);
    count--;
    return temp.value();
}
*/

/**
 * Get a copy of first value found in list.
 *
 * @pre list is not empty
 * @post returns first value in list
 *
 * @return A reference to first value in list.
 */
public E getFirst()
{
    return head.value();
}

/**
 * Get a copy of last value found in list.
 *
 * @pre list is not empty
 * @post returns last value in list
 *
 * @return A reference to last value in list.
 */
public E getLast()
{
    return tail.value();
}

/**
 * Check to see if a value is within list.
 *
 * @pre value not null
 * @post returns true iff value is in list
 *
 * @param value A value to be found in list.
 * @return True if value is in list.
 */
public boolean contains(E value)
{
    DoublyLinkedNode<E> finger = head;
    while ((finger != null) && (!finger.value().equals(value)))
    {
        finger = finger.next();
    }
    return finger != null;
}
```

```
/**
 * Remove a value from list.  At most one value is removed.
 * Any duplicates remain.  Because comparison is done with "equals,"
 * actual value removed is returned for inspection.
 *
 * @pre value is not null.  List can be empty
 * @post first element matching value is removed from list
 *
 * @param value value to be removed.
 * @return value actually removed.
 */
public E remove(E value)
{
    DoublyLinkedNode<E> finger = head;
    while (finger != null &&
           !finger.value().equals(value))
    {
        finger = finger.next();
    }
    if (finger != null)
    {
        // fix next field of element above
        if (finger.previous() != null)
        {
            finger.previous().setNext(finger.next());
        } else {
            head = finger.next();
        }
        // fix previous field of element below
        if (finger.next() != null)
        {
            finger.next().setPrevious(finger.previous());
        } else {
            tail = finger.previous();
        }
        count--;            // fewer elements
        return finger.value();
    }
    return null;
}

/**
 * Determine number of elements in list.
 *
 * @post returns number of elements in list
 *
 * @return number of elements found in list.
 */
public int size()
{
    return count;
}

/**
```

```
 * Determine if list is empty.
 *
 * @post returns true iff list has no elements
 *
 * @return True iff list has no values.
 */
public boolean isEmpty()
{
    return size() == 0;
}

/**
 * Remove all values from list.
 *
 * @post removes all elements from list
 */
public void clear()
{
    head = tail = null;
    count = 0;
}

/**
 * Get value at location i.
 *
 * @pre 0 <= i < size()
 * @post returns object found at that location
 *
 * @param i position of value to be retrieved.
 * @return value retrieved from location i (returns null if i invalid)
 */
public E get(int i)
{
    if (i >= size()) return null;
    DoublyLinkedNode<E> finger = head;
    // search for ith element or end of list
    while (i > 0)
    {
        finger = finger.next();
        i--;
    }
    // not end of list, return value found
    return finger.value();
}

/**
 * Set value stored at location i to object o, returning old value.
 *
 * @pre 0 <= i < size()
 * @post sets ith entry of list to value o, returns old value
 * @param i location of entry to be changed.
 * @param o new value
 * @return former value of ith entry of list.
 */
```

```
public E set(int i, E o)
{
    if (i >= size()) return null;
    DoublyLinkedNode<E> finger = head;
    // search for ith element or end of list
    while (i > 0)
    {
        finger = finger.next();
        i--;
    }
    // get old value, update new value
    E result = finger.value();
    finger.setValue(o);
    return result;
}

/**
 * Insert value at location.
 *
 * @pre 0 <= i <= size()
 * @post adds ith entry of list to value o
 * @param i index of this new value
 * @param o value to be stored
 */
public void add(int i, E o)
{
    Assert.pre((0 <= i) &&
               (i <= size()), "Index in range.");
    if (i == 0) addFirst(o);
    else if (i == size()) addLast(o);
    else {
        DoublyLinkedNode<E> before = null;
        DoublyLinkedNode<E> after = head;
        // search for ith position, or end of list
        while (i > 0)
        {
            before = after;
            after = after.next();
            i--;
        }
        // create new value to insert in correct position
        DoublyLinkedNode<E> current =
            new DoublyLinkedNode<E>(o,after,before);
        count++;
        // make after and before value point to new value
        before.setNext(current);
        after.setPrevious(current);
    }
}

/**
 * Remove and return value at location i.
 *
 * @pre 0 <= i < size()
```

```
 * @post removes and returns object found at that location
 *
 * @param i position of value to be retrieved.
 * @return value retrieved from location i (returns null if i invalid)
 */
public E remove(int i)
{
    Assert.pre((0 <= i) &&
               (i < size()), "Index in range.");
    if (i == 0) return removeFirst();
    else if (i == size()-1) return removeLast();
    DoublyLinkedNode<E> previous = null;
    DoublyLinkedNode<E> finger = head;
    // search for value indexed, keep track of previous
    while (i > 0)
    {
        previous = finger;
        finger = finger.next();
        i--;
    }
    previous.setNext(finger.next());
    finger.next().setPrevious(previous);
    count--;
    // finger's value is old value, return it
    return finger.value();
}

/**
 * Determine first location of a value in list.
 *
 * @pre value is not null
 * @post returns the (0-origin) index of value,
 *   or -1 if value is not found
 *
 * @param value value sought.
 * @return index (0 is first element) of value, or -1
 */
public int indexOf(E value)
{
    int i = 0;
    DoublyLinkedNode<E> finger = head;
    // search for value or end of list, counting along way
    while (finger != null && !finger.value().equals(value))
    {
        finger = finger.next();
        i++;
    }
    // finger points to value, i is index
    if (finger == null)
    {   // value not found, return indicator
        return -1;
    } else {
        // value found, return index
        return i;
```

```
    }
}

/**
 * Determine last location of a value in list.
 *
 * @pre value is not null
 * @post returns the (0-origin) index of value,
 *   or -1 if value is not found
 *
 * @param value value sought.
 * @return index (0 is first element) of value, or -1
 */
public int lastIndexOf(E value)
{
    int i = size()-1;
    DoublyLinkedNode<E> finger = tail;
    // search for last matching value, result is desired index
    while (finger != null && !finger.value().equals(value))
    {
        finger = finger.previous();
        i--;
    }
    if (finger == null)
    {   // value not found, return indicator
        return -1;
    } else {
        // value found, return index
        return i;
    }
}

/**
 * Construct an iterator to traverse list.
 *
 * @post returns iterator that allows traversal of list
 *
 * @return An iterator that traverses list from head to tail.
 */
public Iterator<E> iterator()
{
    return new DoublyLinkedListIterator<E>(head);
}

/**
 * Construct a string representation of list.
 *
 * @post returns a string representing list
 *
 * @return A string representing elements of list.
 */
public String toString()
{
    StringBuffer s = new StringBuffer();
```

```
        s.append("<DoublyLinkedList:");
        Iterator li = iterator();
        while (li.hasNext())
        {
            s.append(" "+li.next());
        }
        s.append(">");
        return s.toString();
    }
}
```