Lab 2

Due 2 February

Handout 2
CSC 062: Spring, 2007
26 January

Compression

1 Short Answers

Bring a written answer to the following question to lab on Tuesday. Pseudo-code (english written in an algorithmic style) is fine. Don't bother to write and debug Java.

- C-3.9 on page 150 of Goodrich and Tomassia

2 Lab

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 lab 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 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:

(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 extend `DoublyLinkedList<E>` to a class `CurDoublyLinkedList<E>`. This class should have extra capability to move to any desired node of the list (called the current element) and then either add a new node after this element or to remove the current node. Your new class should support all of the old methods of the Lists as well as `first()`, `last()`, `next()`, `back()`, `isOffRight()`, `isOffLeft()`, `getCurrentValue()`, `addAfterCurrent(E value)`, and `deleteCurrent()`. Specifications for these methods can be found in the attached code.

You will also have to override existing methods that might impact the value of `Current`. For example, whenever a new value is added, its node should become the value of current. If the current element is removed, then its successor in the list should become the current element (even if it is the last element of the list - in which case the trailer will be the current element).

2. Use this to design the class `CompressedTable<ValueType>` implementing the interface `TwoDTable<ValueType>`. 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<Association<RowOrderedPosn,ValueType` (as well as other instance variables).

   The class `Association<KeyType,ValueType>` can be found in the structure library. It represents a pair of elements. For example if `KeyType` is `String` and `ValueType` is `Integer` then it might represent pairs of names and phone numbers. In our case it will represent pairs of `RowOrderedPosn` and `Color`, representing postions in the array and their associated colors. For example if 2 and 3 are the row and columns in myPosn then new Association¡RowOrderedPosn,Color¿(myPosn,Color.GREEN) represents the fact that color green is at row 2 and column 3 in the grid. Study this class to make sure you understand it, but you need not write any code for it.

I have written an Applet which allows the user to interact with a window to create and manipulate an image which will be stored using a CompressedTable. Your classes can be linked in with my program, resulting in a very efficient way of storing such images. You can run it by typing on a command line:

```
appletViewer CompressedGrid.html
```

where `CompressedGrid.html` (and all other startup files) can be found in /common/cs/cs062/labs/Lab2-Picture. (From the shortcut to cs062 on your desktop, just go into labs and Lab2-Picture.)

I 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 just type it in and expect it to work with my GridTest program – it won't and you will have real difficulty trying to determine where your errors are. Write simple test programs that don't use graphics, and print out the list so you can see what is currently contained in it after each of your operations.

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

1. Find the node of the list which encodes the position being updated.
2. If the new information is the same as that in the node then nothing needs to be done. Otherwise determine if the node represents exactly the position being updated.
3. If so, update the value of the node, otherwise add a new node representing the new position
4. 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.
5. 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.

3 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.

4 What to hand in

This is a fairly complex program. As a result you should bring a very complete design for your program to lab on Tuesday. I will be grading your designs of the two methods `deleteCurrent` of `CurDoublyLinkedList` and `updateInfo` of `CompressedTable`. While these should not be written in Java, I will need to see the complete logic of the methods. These designs will be worth 10 to 20% of your grade for this lab so do a good, careful job. Remember that you must draw pictures and look at all possible special cases in order to get these right.

Your final program will be due on Friday, February 2 at 5 p.m. As usual place all of your code in a folder and drop it off in the CS 062 dropbox folder.