Engr 660: Software Engineering II
Fall Semester 2003, Final Project
Due Monday, 8 December, 2003 at 5:00 P.M.

Background

This project deals with the coordinated processing of two ordered sequences, that is, *cosequential processing*.

Consider a sequence of elements that are ordered by the values of some key attribute. A sequence is *ascending* if every element is <= all of its successors in the sequence. Successor means an element that occurs later in the sequence (i.e., away from the beginning). A sequence is *increasing* if every element is < its successors. Similarly, a sequence is *descending* or *decreasing* if every element is >= or >, respectively, its successors.

One example of cosequential processing is a program to merge two ascending sequences of some type into a single ascending sequence. In this process the smaller of the two elements is repeatedly moved to the result sequence until both input sequences are empty.

For another example, suppose an increasing sequence is used to represent a set of values. Then cosequential processing can be used to implement the set operations such as union and intersection. A union of two sets is the sequence that includes all elements in both sequences with no duplicates. An intersection would contain all elements that match in the two sequences.

Cosequential processing can also be used to update a master file with data from a transaction file. Suppose one sequential file contains bank account numbers, in increasing order, paired with the month-end balances of those accounts. (This is the master file.) Suppose a second sequential file contains transactions on the bank accounts. That is, it contains deposits and withdrawals ordered first in ascending order by account number and within that in ascending order by date. (This is called the transaction file.) Then a cosequential processing can be used to apply the transaction file to the master file to create and updated master file of account balances. For each matching pair of account numbers, the transactions are applied to the matching account balance.

Cosequential processing is basically the generalized merge of two sequences. Using the typical technique, a general merge function must examine the "heads" of its two input sequences, take an appropriate action (e.g., compute an appropriate value for the initial segment of the result sequence), advance the input sequences appropriately, and then repeat the process on the remaining portions of the input sequences to generate the remainder of the output sequence.

We can represent our cosequential processing model in terms of a general merge process defined in pseudocode as follows:

```
gmerge(list1, list2)

    while (list1 and list2 both not empty)

        head1 = first element of list1
        key1  = key of head1
        head2 = first element of list2
        key2  = key of head2

        if (key1 < key2)
            do "list1 is less" action
            do "advance list1" action
        else if (key1 = key2)
            do "both equal" action
            do "advance when both equal" action
        else
            do "list1 is greater" action
            do "advance list2" action

    while (list1 is not empty)
        head1  = first element of list1
        do "list1 is less" action
        do "advance list1" action

    while (list2 is not empty)
        head2  = first element of list2
        do "list1  is greater" action
        do "advance list2" action

    return result
```

Assignment

This is an individual assignment!

Analyze the cosequential processing problem and design an appropriate software framework, using the above pseudocode as a guide. The framework should consist of frozen spots (design and code featurs that are common for all applications in the family) and hot spots (parts of the framework that are designed for customization with client code). It should be possible and convenient to build a variety of cosequential processing applications using the framework.

Issues to consider:

- The framework should be able to handle implementations of the examples described above such as the set operations, merging lists, and the master-transaction file update process.
- The framework should be able to be connected to lists that are stored on disk files or in memory or wherever.
- The frameworks should be able to accommodate different kinds of user-defined list elements, keys, comparison operators, etc.
- You should identify other hot spots as appropriate.

Turn in the following items:

- A document that (a) describes the design and (b) describes how use the framework to build an application.
- Printed and electronic copies of the framework source code, appropriately documented.
- Printed and electronic copies of the source code for applications of the framework to two significantly different problems.