Flexichain: An editable sequence and its gap-buffer implementation

Robert Strandh (LaBRI*), Matthieu Villeneuve, Tim Moore (LaBRI)

2004-04-05

Abstract

Flexichain is an API for editable sequences. Its primary use is in end-user applications that edit sequences of objects such as text editors (characters), word processors (characters, paragraphs, sections, etc), score editors (notes, clusters, measures, etc), though it can also be used as a stack and a double-ended queue.

We also describe an efficient implementation of the API in the form of a circular gap buffer. Circularity avoids a common worst case in most implementations, makes queue operations efficient, and makes worst-case performance twice as good as that of ordinary implementations

1 Introduction

Editable sequences are useful, in particular in interactive applications such as text editors, word processors, score editors, and more. In such applications, it is highly likely that an editing operation is close to the previous one, measured as the difference in positions in the sequence. This statistical behavior makes it feasible to implement the editable sequence as a gap buffer.

The basic idea is to store objects in a vector that is usually longer than the number of elements stored in it. For a sequence of N elements where editing is required at

*Laboratoire Bordelais de Recherche en Informatique, Bordeaux, France

index i, elements 0 through i are stored at the beginning of the vector, and elements $i+1$ through $N-1$ are stored at the end of the vector. When the vector is longer N, this storage leaves a *gap*. Editing operations always result in modifications at the beginning or at the end of the gap.

Occasionally, the gap has to be moved, or rather, some elements have to be moved so as to leave the gap where the next editing operation is desired. In the worst case, i.e., that of an alternating sequence of editing operations at the beginning and at the end of the sequence, every element needs to be moved. While it can be argued that this case does not happen very frequently, it unfortunately corresponds to operations that might be reasonable in some clients, namely *rotation* of the elements or the use of the sequence as a *queue*.

The kind of worst-case behavior described in the previous paragraph can be avoided by the use of a doubly-linked list rather than a gap buffer to implement the sequence. Unfortunately, the doubly-linked list has unacceptable storage overhead for small objects such as characters (a factor 8-16 according to the word size of the machine and the implementation of the memory allocator) or bits (a factor 128-256).

2 Previous work

We are not the first ones to consider the problem of editable sequences for interactive applications.

Multics Emacs [Gre96] [Gre80] used a doubly-linked list of lines of text. Each line was a vector of characters. A new vector type was added to Multics Maclisp for the purpose. The new type used complex instructions of the underlying architecture that allowed an arbitrary sequence of bytes to be moved with a single instruction. While this implementation was fine for most text editing, it was painfully slow for editing files with few newlines, since a considerable number of bytes had to be moved for each editing operation. Today's hardware is 1000 times faster than what Multics ran on, so this implementation might be acceptable today, even though most processors might not have the specialized instructions required so it would have to be implemented in software.

GNU Emacs [LLS02] stores the entire buffer as a big gap buffer as described in the introduction. This implementation avoids bad worst-case behavior for long lines of text. On the other hand, it introduces a different, potentially more serious, worst-

case requiring every single character in the buffer to be moved for the alternating sequence of editing operations described in the introduction.

Hemlock [CM89] (the Emacs-like editor distributed with CMUCL) uses a sequence of lines. Lines can be stored in different places as compact strings, and one of the lines (the open-line) is represented as a gap buffer. This implementation largely avoids the worst case of GNU Emacs, at least for ordinary text with lines of relatively modest length.

Goatee (the Emacs-like editor of McCLIM [SM02]) uses a doubly-linked list of lines, each line being a gap buffer.

Gsharp [Str02], the interactive editor for music scores, currently uses ordinary singly-linked Lisp lists, since the score is divided into relatively few smaller units corresponding to musical phrases. Still, this implementation introduces some serious worst-case behavior that we would like to avoid.

It is interesting to notice that the sequence of lines used in Goatee (and in Hemlock and probably elsewhere as well) is just a version of an editable sequence with the implementation exposed.

3 Flexichain: an API for editable sequences

The Flexichain API grew out of the need for code factoring between different applications (especially Goatee and Gsharp, but hopefully Hemlock as well), and sometimes between different parts of one application.

To provide maximum flexibility for potential clients, we decided to divide the API in two different layers: Flexichain and Cursorchain. The Flexichain layer provides editing operations based on *positions* represented as integers, whereas the Cursorchain layer introduces the possibility of an arbitrary number of *cursors* into the editable sequence.

3.1 The basic layer: Flexichain

The basic layer provides editing operations based on the concept of a *position*.

For an insert operation, a position is an integer between 0 and N inclusive, where N is the length of the sequence. In general, a value of i indicates the position

before element number i in the sequence, except of course when $i = N$ and there is no element i. In this case, the position indicates the end of the sequence. For reasons that will be explained in the next section, we actually provide two insert operations `insert<*` and `insert>*` that are entirely equivalent when only the basic layer is used.

For a delete operation, a position is an integer between 0 and $N - 1$ and indicates the element number of the element to be deleted.

The basic layer also provides operations for accessing and replacing an element at an arbitrary position in the sequence, as well as operations that treats the sequence as a stack, a linear double-ended queue, or a circular queue.

Some relatively simple applications can use the basic layer directly. The main inconvenience of the basic layer is that the position of an element changes as a result of editing operations at lower positions in the sequence.

3.2 The second layer: Cursorchain

Complex applications such as multi-window text editors need to manage several positions in the sequence such that these positions refer to the same element independently of any editing operations in other places in the sequence.

For that reason, the second layer introduces the concept of a *cursor*. A cursor is similar to the *point* or a *mark* of Emacs. It is positioned either at the beginning of the sequence, at the end of the sequence, or between two element of the sequence.

All the operations of the basic layer can be used on instances of cursorchain.

While it is straightforward to determine what happens to a cursor when an element is deleted, it is not clear what happens when an element is inserted at a position occupied by one or more cursors. There are actually two possibilities: either the element is inserted *before* the cursors (i.e., between the cursors and the element that precedes them) so that the cursors end up at a position *after* the newly inserted element, or the element is inserted *after* the cursors (i.e., between the cursors and the element that succeeds them) so that the cursors end up at a position *before* the newly inserted element.

Different applications might want different behavior with respect to insertion, and some applications (this is the case with Gsharp) might want one behavior in one part of the buffer representation and another behavior in a different part. For that

reason, we provide two different insert operations: `insert<` for the first case and `insert>` for the second case. As indicated in the previous section, there are two insert operations in the basic layer as well, simply because these operations might be used on a cursorchain and the behavior of potential cursors at the insertion position must be specified.

Since cursors are never conceptually positioned on a particular element, we provide two different delete operations to delete elements before and after the cursor, and two different operations for accessing and replacing an element with respect to the cursor (before it and after it).

We also provide operations to move the cursor forward and backward by an arbitrary number of positions, to translate between a cursor and its position, and to determine whether the cursor is at the beginning or at the end of the sequence.

4 Implementation of the API

In order to avoid the worst-case behavior of a buffer implementation of the type used by GNU Emacs, we use a *circular gap buffer*. Thus, the first and the last element of the underlying vector are considered contiguous, and the first element of the sequence is not necessarily the first element of the underlying vector. We keep track of the first element by introducing another slot in the class that represents the Flexichain.

4.1 Implementing the basic layer

There are two main considerations with regard to the implementation of the basic layer, namely when and how to change the size of the vector that holds the sequence, and how to move the gap.

4.2 Changing the size of the vector

Whenever and insert operation is issued on a Flexichain that is *full* (i.e., the size of the vector holding the sequence has the same length as the sequence itself), its underlying vector must be extended. In order to maintain linear worst-case complexity of a sequence of editing operations, we must then multiply the size of

a vector by a constant (called the *expand factor* rather than adding a fixed number of elements.

Each resize operation requires all the elements to be moved. It is therefore desirable to avoid resize operations as much as possible. For that reason, it is advantageous to have a large expand factor. On the other hand, in order to avoid wasted space, it is desirable to have a small expand factor.

We use a default expand factor of 1.5 with the possibility for client code to alter it. Applications that manipulate sequences that vary little in length can use a small expand factor to minimize overhead, while applications that use relatively small sequences the length of which vary a lot can use a larger expand factor.

The vector has to be expanded as a result of an insert operation on a full Flexichain. It is particularly easy to move the gap in this case (no elements need to be moved). For that reason, in this case we first move the gap and then expand the vector.

To avoid too much overhead when the number of elements in the sequence decreases, we occasionally have to shrink the vector. In order to avoid having to immediately expand it again in case of more insert operations being issued, we only shrink the vector when the ratio between the length of the vector and the length of the sequence is greater than the square of the expand factor.

Shrinking the vector preserves the position of the gap.

4.3 Moving the gap

Perhaps the most complex part of implementing the basic layer is moving the gap when an editing operation (insert or delete) is issued at a position other than that of the gap.

There are three different possible configurations of the gap with respect to the data. Figure 1 shows the case where both the gap and the data are contiguous. Figure 2 shows the case where the data is not contiguous. Finally, figure 3 shows the case where the gap is not contiguous.

We make sure we always move the minimum number of elements required by moving the gap in either of the two directions possible.

It turns out that there are five different cases of combinations between the configurations of the gap and the position of the editing operation that need to be taken into account. Two of the five cases need a single call to the Lisp function `replace`,

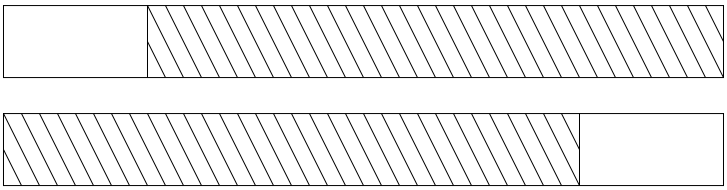

Figure 1: Gap and data are both contiguous

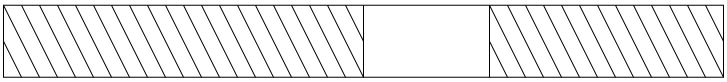

Figure 2: Data is not contiguous

two more require two calls, and one case requires three calls.

4.4 Implementing the second layer

The main difficulty in implementing the second layer lies in the way cursors are managed. It is necessary for the implementation to access all cursors in order to be able to update their corresponding positions as the sequence is altered. To avoid memory leaks, we use weak references to store the cursors, so that when client code no longer refers to a cursor, we can detect that.

Perhaps the most natural implementation of cursors would be to store the corresponding position in the sequence, and update that position whenever an editing operation with a smaller position is issued. However, such an implementation would make the complexity of editing operations proportional to the number of cursors into the sequence which is not desirable.

Instead, we store *indexes* into the underlying vector. This way, we can split the cursors into three sets according to whether they are positioned before, at, or after

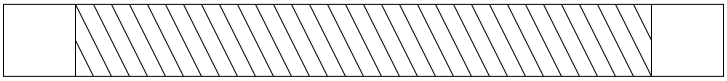

Figure 3: Gap is not contiguous

the gap (although we have not implemented this possibility yet). Only cursors that are at the gap potentially need to be altered after an editing operation. All other cursors remain unchanged. Instead, cursors are updated as a result of moving the gap.

Cursor updates are implemented as `:before`, `:after`, and `:around`, methods on the generic functions in the basic layer that handle moving the gap and changing the size of the vector. These operations constitute an internal protocol of the Flexichain library and is not part of external API.

5 Conclusions and future work

We believe we have a good API and a very implementation of it. We would be interested in seeing our code used in a variety of existing projects, in particularly Goatee and Gsharp, but also in similar projects such as Portable Hemlock and others.

A typical text editor such as Goatee or Hemlock could use Flexichains (or rather Cursorchains) to implement both the sequence of lines of text and the sequence of characters within each line.

Gsharp will use Flexichains for all the levels (currently 6) of its buffer protocol, which will both simplify the code and improve performance considerably.

The Flexichain API and its implementation are both well documented, making it easier for potential clients to take advantage of it.

References

[CM89] Bill Chiles and Rob MacLachlan. Hemlock Command Implementor's Manual. Technical Report CMU-CS-89-134-R1, School of Computer Science, Carnegie Mellon University, 1989.

[Gre80] Bernard S Greenberg. Prose and CONS: A Commercial Text-Processing System in Lisp. In *Proceedings of the 1980 Lisp Conference*, 1980.

[Gre96] Bernard S Greenberg. Multics Emacs: The History, Design and Implementation. Technical report, http://www.multicians.org/mepap.html, 1996.

[LLS02] Bil Lewis, Dan LaLiberte, and Richard Stallman. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, January 2002.

[SM02] Robert Strandh and Tim Moore. A Free Implementation of CLIM. In *Proceedings of the International Lisp Conference*, October 2002.

[Str02] Robert Strandh. Gsharp, an Extensible, Interactive Score Editor. In *Proceedings of the International Lisp Conference*, October 2002.