Data Structures for Text Sequences

Charles Crowley
University of New Mexico*

June 10, 1998

Abstract

The data structure used ot maintain the sequence of characters is an important part of a text editor. This paper investigates and evaluates the range of possible data structures for text sequences. The ADT interface to the text sequence component of a text editor is examined. Six common sequence data structures (array, gap, list, line pointers, fixed size buffers and piece tables) are examined and then a general model of sequence data structures that encompasses all six structures is presented. The piece table method is explained in detail and its advantages are presented. The design space of sequence data structures is examined and several variations on the ones listed above are presented. These sequence data structures are compared experimentally and evaluated based on a number of criteria. The experimental comparison is done by implementing each data structure in an editing simulator and testing it using a synthetic load of many thousands of edits. We also report on experiments on the senstivity of the results to variations in the parameters used to generate the synthetic editing load.

1 Introduction

The central data structure in a text editor is the one that manages the sequence of characters that represents the current state of the file that is being edited. Every text editor requires such a data structure but books on data structures do not cover data structures for text sequences. Articles on the design of text editors often discuss the data structure they use [1, 3, 6, 8, 11, 12] but they do not cover the area in a general way. This article is concerned with such data structures.

Figure 1 shows where sequence data structures fit in with other data structures. Some ordered sets are ordered by something intrinsic in the items in the sets (e.g., the value of an integer, the lexicographic position of a string) and the position of an inserted item depends on its value and the values of the items already in the set. Such data structures are mainly concerned with fast searching. Data structures for this type of ordered set have been studied extensively.

The other possibility is for the order to be determined by where the items are placed when they are inserted into the set. If insert and delete is restricted to the two ends of the ordering then you have a deque. For deques, the two basic data structures are an array (used circularly) and a linked list. Nothing beyond this is necessary due to the simplicity of the ADT interface to deques. If you can insert and delete items from anywhere in the ordering you have a sequence. An important subclass

*Author's address: Computer Science Department, University of New Mexico, Albuquerque, New Mexico 87131, office: 505-277-5446, messages: 505-277-3112, fax: 505-277-0813, email: crowley@unmvax.cs.unm.edu

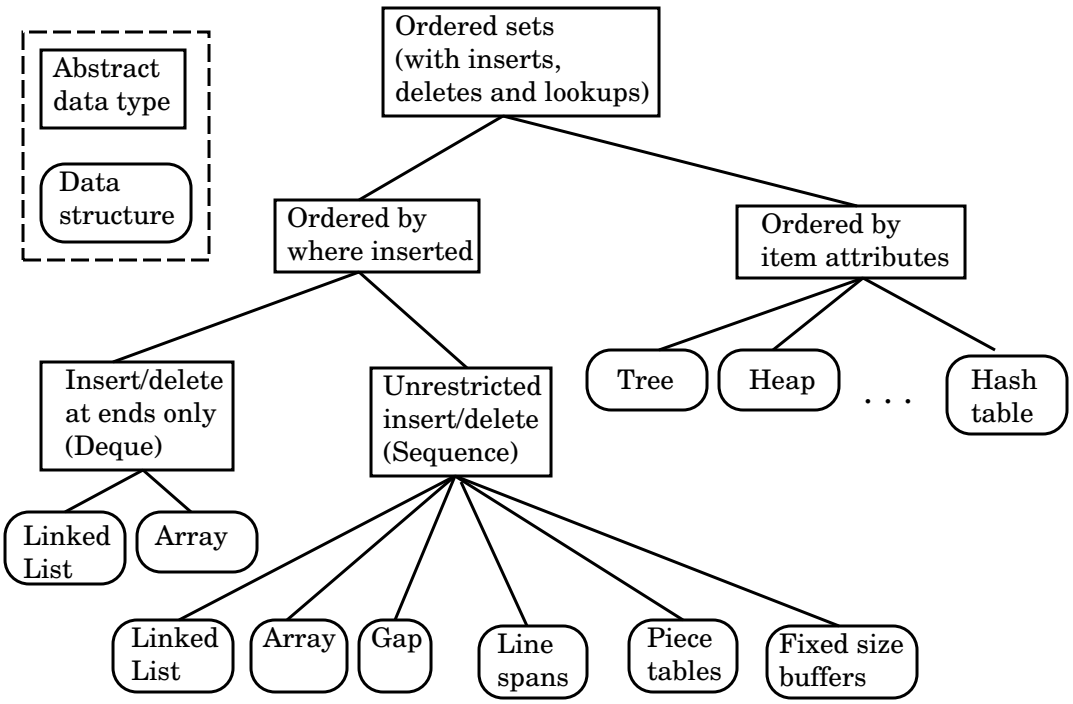

Figure 1: Ordered sets

is sequences where reading an item in the sequence (by position number) is extremely localized. This is the case for text editors and it is this subclass that is examined in this paper.

A linked list and an array are the two obvious data structures for a sequence. Neither is suitable for a general purpose text editor (a linked list takes up too much memory and an array is too slow because it requires too much data movement) but they provide useful base cases on which to build more complex sequence data structures. The gap method is a simple extension of an array, it is simply an array with a gap in the middle where characters are inserted and deleted. Many text editors use the gap method since it is simple and quite efficient but the demands on a modern text editor (multiple files, very large files, structured text, sophisticated undo, virtual memory, etc.) encourage the investigation of more complicated data structures which might handle these things more effectively.

The more sophisticated sequence data structures keep the sequence as a recursive sequence of spans of text. The line span method keeps each line together and keeps an array or a linked list of line pointers. The fixed buffer method keeps a linked list of fixed size buffers each of which is partially full of text from the sequence. Both the line span method and the fixed buffer method have been used for many text editors.

A less commonly used method is the piece table method which keeps the text as a sequence of "pieces" of text from either the original file and an "added text" file. This method has many advantages and these will become clear as the methods are presented in detail and analyzed. A major purpose of this paper is to describe the piece table method and explain why it is a good data structure for text sequences.

Looking at methods in detail suggests a general model of sequence data structures that subsumes them all. Based on examination of this general model I will propose several new sequence data structures that do not appear to have been tried before.

It is difficult to analyze these algorithms mathematically so an experimental approach was taken. I have implemented a text editor simulator and a number of sequence data structures. Using this as an experimental text bed I have compared the performance of each of these sequence data structures under a variety of conditions. Based on these experiments and other considerations I conclude with recommendations on which sequence data structures are best in various situations. In almost all cases, either the gap method or the piece table method is the best data structure.

2 Sequence Interfaces

It is useful to start out with a definition of a sequence and the interface to it. Since the more sophisticated text sequence data structures are recursive in that they require a component data structure that maintains a sequence of pointers, I will formulate the sequence of a general sequence of "items" rather than as a sequence of characters. This supports discussion of the recursive sequence data structures better.

2.1 The Sequence Abstract Data Type

A text editor maintains a sequence of characters, by implementing some variant of the abstract date type Sequence. One definition of the Sequence abstract data type is:

Domains:

- ITEM — the data type that this is a sequence of (in most cases it will be an ASCII character).
- SEQUENCE — an ordered set of ITEMS.
- POSITION — an index into the SEQUENCE which identifies the ITEMS (in this case it will be a natural number from 0 to the length of the SEQUENCE minus one).

Syntax:

- $Empty$: $\rightarrow$ SEQUENCE
- $Insert$: SEQUENCE $\times$ POSITION $\times$ ITEM $\rightarrow$ SEQUENCE
- $Delete$: SEQUENCE $\times$ POSITION $\rightarrow$ SEQUENCE
- $ItemAt$: SEQUENCE $\times$ POSITION $\rightarrow$ ITEM $\cup$ $\{EndOfFile\}$

Types:

- s : SEQUENCE;
- i : ITEM;

- p, p_1, p_2 : POSITION;

Axioms:

1. $Delete(Empty, p) = Empty$
2. $Delete(Insert(s, p_1, i), p_2) =$

   if $p_1 < p_2$ **then** $Insert(Delete(s, p_2 - 1), p_1, i)$

   if $p_1 = p_2$ **then** s

   if $p_1 > p_2$ **then** $Insert(Delete(s, p_2), p_1 - 1, i)$
3. $ItemAt(Empty, p) = EndOfFile$
4. $ItemAt(Insert(s, p_1, i), p_2) =$

   if $p_1 < p_2$ **then** $ItemAt(s, p_2 - 1)$

   if $p_1 = p_2$ **then** i

   if $p_1 > p_2$ **then** $ItemAt(s, p_2)$

The definition of a SEQUENCE is relatively simple. Axiom 1 says that deleting from an *Empty* SEQUENCE is a no-op. This could be considered an error. Axiom 2 allows the reduction of a SEQUENCE of *Insert*s and *Delete*s to a SEQUENCE containing only *Insert*s. This defines a canonical form of a SEQUENCE which is a SEQUENCE of *Insert*s on a initial *Empty* SEQUENCE. Axiom 3 implies that reading outside the SEQUENCE returns a special *EndOfFile* item. This also could have been an error. Axiom 4 defines the semantics of a SEQUENCE by defining what is at each position of a canonical SEQUENCE.[1]

2.2 The C Interface

How would this translate into C? First some type definitions:

```
typedef ReturnCode int; /* 1 for success, zero or negative for failure */

typedef Position int; /* a position in the sequence */
    /* the first item in the sequence is at position 0 */

typedef Item unsigned char; /* they are sequences of eight bit bytes */

typedef struct {
    /* To be determined */
    /* Whatever information we need for the data structures we choose */
    } Sequence;
```

In this interface the only operations that change the Sequence are *Insert* and *Delete*.

[1] I am ignoring the error of inserting beyond the end of the existing sequence.

- *Sequence Empty( );*
- *ReturnCode Insert(Sequence *sequence, Position position, Item ch);*
- *ReturnCode Delete(Sequence *sequence, Position position);*
- *Item ItemAt(Sequence *sequence, Position position);* — This does not actually require a pointer to a Sequence since no change to the sequence is being made but we expect that they will be large structures and should not be passing them around. I am ignoring error returns (e.g., position out of range) for the purposes of this discussion. These are easily added if desired.
- *ReturnCode Close(Sequence *sequence);*

Many variations are possible. The next few paragraphs discuss some of them.

Any practical interface would allow the sequence to be initialized with the contexts of a file. In theory this is just the *Empty* operation followed by an *Insert* operation for each character in the initializing file. Of course, this is too inefficient for a real text editor.[2] Instead we would have a *NewSequence* operation:

- *Sequence NewSequence(char * file_name);* — The sequence is initialized with the contents of the file whose name is contained in 'file_name'.

Usually the *Delete* operation will delete any logically contiguous subsequence

- *ReturnCode Delete(Sequence *sequence, Position beginPosition, Position endPosition);*

Sometimes the *Insert* operation will insert a subsequence instead of just a single character.

- *ReturnCode Insert(Sequence *sequence, Position position, Sequence sequenceToInsert);*

Sometimes *Copy* and *Move* are separate operations (instead of being composed of *Inserts* and *Deletes*).

- *ReturnCode Copy(Sequence *sequence, Position fromBegin, Position fromEnd, Position toPosition);*
- *ReturnCode Move(Sequence *sequence, Position fromBegin, Position fromEnd, Position toPosition);*

A *Replace* operation that subsumes *Insert* and *Delete* in another possibility.

- *ReturnCode Replace(Sequence *sequence, Position fromBegin, Position fromEnd, Sequence sequenceToReplaceItWith);*

Finally the *ItemAt* procedure could retrieve a subsequence.

[2] Although this is the method I use in my text editor simulator described later.

- *ReturnCode SequenceAt(Sequence *sequence, Position fromBegin, Position fromEnd, Sequence *returnedSequence);*

These variations will not affect the basic character of the data structure used to implement the sequence or the comparisons between them that follow. Therefore I will assume the first interface (Empty, Insert, Delete, IntemAt, and Close).

3 Comparing sequence data structures

In order to compare sequence data structures it is necessary to know the relative frequency of the five basic operations in typical editing.

The *NewSequence* operation is infrequent. Most sequence data structures require the *NewSequence* operation to scan the entire file and convert it to some internal form. This can be a problem with very large files. To look through a very large file, it must be read in any case but if the sequence data structure does not require sequence preprocessing it can interleave the file reading with text editor use rather than requiring the user to wait for the entire file to be read before editing can begin. This is an advantage and means the user does not have to worry about how many files are loaded or how large they are, since the cost of reading the large file is amortized over its use.

The *Close* operation is also infrequent and not normally an important factor in comparing sequence data structures.

The *ItemAt* operation will, of course, be very frequent, but it will also be very localized as well because characters in a text editor are almost always accessed sequentially. When the screen is drawn the characters on the screen are accessed sequentially beginning with the first character on the screen. If the user pages forward or backwards the characters are accessed sequentially. Searches normally begin at the current position and proceed forward or backward sequentially. Overall, there is almost no random access in a text editor. This means that, although the *ItemAt* operation is very frequent, its efficiency in the general case is not significant. Only the case where a character is requested that is sequentially before or after the last character accessed needs to be optimized.

To test this hypothesis I instrumented a text editor and looked at the *ItemAt* operations in a variety of text editing situations. The number of *ItemAt* calls that were sequential (one away from the last *ItemAt*) was always above 97% and usually above 99%. The number of *ItemAts* that were within 20 items of the previous *ItemAt* was always over 99%. The average number of characters from one *ItemAt* to the next was typically around 2 but sometimes would go up to as high as 10 or 15. Overall these experiments verify that the positions of *ItemAt* calls are extremely localized.

Thus sequence data structures should be optimized for edits (*inserts* and *deletes*) and sequential character access. Since caching can be used with most data structures to make sequential access fast, this means that the efficiency of inserts and deletes is paramount. With regard to *Inserts* and *Deletes* one would assume that there would be more *Inserts* than *Deletes* but that there would be a mix between them in typical text editing.

In all sequence data structures, *ItemAt* is much faster than *Inserts* and *Deletes*. If we consider typical text editing, the display is changed after each *Insert* and *Delete* and this requires some

number of *ItemAts*, often just a few characters around the edit but possibly the whole rest of the window (if a newline in inserted or deleted).

The criteria used for comparing sequence data structures are:

- The time taken by each operation
- The paging behavior of each operation
- The amount of space used by the sequence data structure
- How easily it fits in with typical file and IO systems
- The complexity (and space taken by) the implementation

Later I will present timings comparing the basic operations for a range of sequence data structures. These timings will be taken from example implementations of the data data structures and a text editor simulator that calls these implementations.

4 Definitions

An *item* is the basic element. Usually it will be a character. A *sequence* is an ordered set of items. Sequential items in a sequence are said to be *logically contiguous*. The sequence data structure will keep the items of the sequence in *buffers*. A buffer is a set of sequentially addressed memory locations. A buffer contains items from the sequence but not necessarily in the same order as they appear logically in the sequence. Sequentially addressed items in a buffer are *physically contiguous*. When a string of items is physically contiguous in a buffer and is also logically contiguous in the sequence we call them a *span*. A *descriptor* is a pointer to a span. In some cases the buffer is actually part of the descriptor and so no pointer is necessary. This variation is not important to the design of the data structures but is more a memory management issue.

Sequence data structures keep spans in buffers and keep enough information (in terms of descriptors and sequences of descriptors) to piece together the spans to form the sequence. Buffers can be kept in memory but most sequence data structures allow buffers to get as large as necessary or allow an unlimited number of buffers. Thus it is necessary to keep the buffers on disk in *disk files*. Many sequence data structures use buffers of unlimited size, that is, their size is determined by the file contents. This requires the buffer to be a disk file. With enough disk block caching this can be made as fast as necessary.

The concepts of buffers, spans and descriptors can be found in almost every sequence data structure. Sequence data structures vary in terms of how these concepts are used.

If a sequence data structures uses a variable number of descriptors it requires a recursive sequence data structure to keep track of the sequence of descriptors. In section 5 we will look at three sequence data structures that use a fixed number of descriptors and in section 6 we will look at three sequence data structures that use a variable number of descriptors. Section 7 will present a general model of a sequence data structure that encompasses all these data structures.

5 Basic Sequence Data Structures

All three of the sequence data structures in this section use a fixed number (either one, two or three) of external descriptors.[3]

5.1 The array method (one span in one buffer)

This is the most obvious sequence data structure. In this method there is one span and one buffer. Two descriptors are needed: one for the span and one for the buffer. (See Figure 2[4]) The buffer

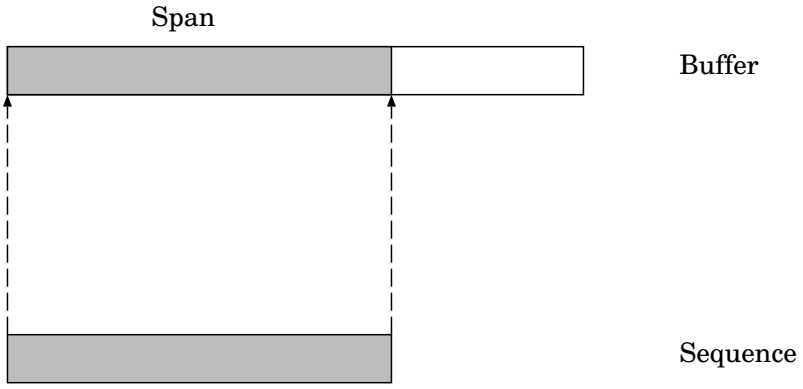

Figure 2: The array method

contains the items of the sequence in physically contiguous order. *Deletes* are handled by moving all the items following the deleted item to fill in the hole left by the deleted item. *Inserts* are handled by moving all the items that will follow the item to be inserted in order to make room for the new item. *ItemAt* is an array reference. The buffer can be extended as much as necessary to hold the data.

Clearly this would not be an efficient data structure if a lot of editing was to be done are large files. It is a useful base case and is a reasonable choice in situations where few inserts and deletes are made (e.g., a read-only sequence) or the sequences are relatively small (e.g., a one-line text editor). This data structure is sometimes used to hold the sequence of descriptors in the more complex sequence data structure, for example, an array of line pointers (see section 6).

5.2 The gap method (two spans in one buffer)

The gap method is only a little more complex than the array method but it is more much efficient. In this method you have one large buffer that holds all the text but there is a gap in the middle of the buffer that does not contain valid text. (See Figure 3) Three descriptors are needed: one for each span and one for the gap. The gap will be at the text "cursor" or "point" where all text

[3] That is, descriptors that are not kept in other descriptors.

[4] In all these figures, each pair of dashed arrows pointing from the (logical) sequence to the buffers represents one descriptor.

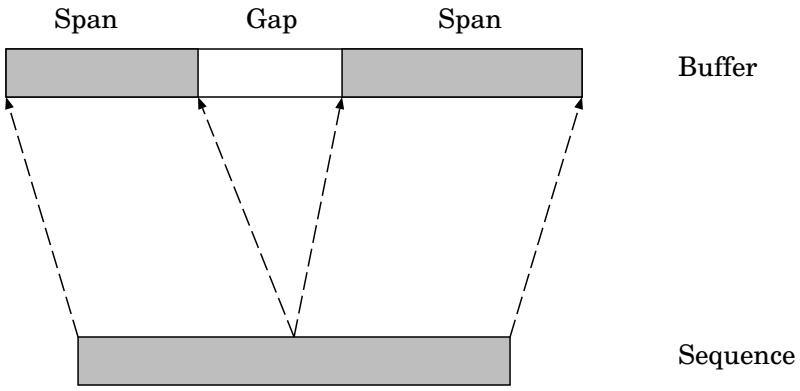

Figure 3: The gap (or two span) method

editing operations take place. *Inserts* are handled by using up one of the places in the gap and incrementing the length of the first descriptor (or decrementing the begin pointer of the second descriptor). *Deletes* are handled by decrementing the length of the first descriptor (or incrementing the begin pointer of the second descriptor). *ItemAt* is a test (first or second span?) and an array reference.

When the cursor moves the gap is also moved so if the cursor moves 100 items forward then 100 items have to be moved from the second span to the first span (and the descriptors adjusted). Since most cursor moves are local, not that many items have to be moved in most cases.

Actually the gap does not need to move every time the cursor is moved. When an editing operation is requested then the gap is moved. This way moving the cursor around the file while paging or searching will not cause unnecessary gap moves.

If the gap fills up, the second span is moved in the buffer to create a new gap. There must be an algorithm to determine the new gap size. As in the array case, the buffer can be extended to any length. In practice, it is usually increased in size by some fixed increment or by some fixed percentage of the current buffer size and this becomes the size of the new gap. With virtual memory we can make the buffer so large that it is unlikely to fill up. And with some help from the operating system, we can expand the gap without actually moving any data. This method does use up large quantities of virtual address space, however.

This method is simple and surprisingly efficient. The gap method is also called the *split buffer* method and is discussed in [9].

5.3 The linked list method

At the other extreme is the linked list method which has a span for each item in the sequence and the span is kept in the descriptor. This requires a large (and file content dependent) number of descriptors which must be sequenced. If we link the descriptors together into a linked list then we really only have to remember one descriptor, the head of the list. (See Figure 4)

Inserts and *Deletes* are fast and easy (just move some pointers) but *ItemAt* is more expensive than the array or gap methods since several pointers may have to be followed.

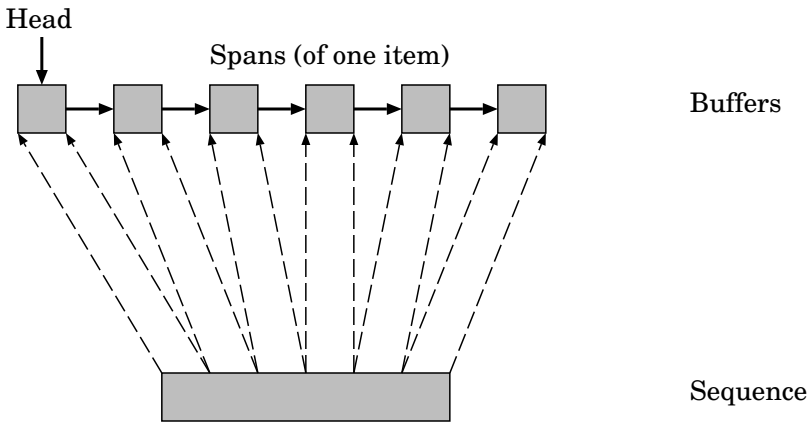

Figure 4: The linked list method

The linked list method uses a lot of extra space and so is not appropriate for a large sequence but is frequently used as a way of implementing a sequence of descriptors required in the more complex sequence data structures. In fact, it is the most common method for that.

One could think of the array method as a special case of the linked list method. An array is really a linked list with the links implicit, that is, the link is computed to be the next physically sequential address after the current item. In this view, the linked list method the only primitive sequence data type. The array method is a special case if linked list method and the gap method is a variation on the array method.

6 Recursive Sequence Data Structures

In this section three more sequence data structures are presented. Each of these methods requires a variable number of descriptors and so must recursively use a (usually simpler) sequence data structure to implement the sequence of descriptors.

6.1 Determination of span and buffer sizes

The basic cases either used a fixed number of spans (one for the array method and two for the gap method) or spans of size 1 (the linked list method). The recursive methods described in this section use a variable number of spans. How is the span size determined? There are two possibilities: either the spans are determined by the contents of the file or by the text editor independent of the sequence contents. The main example of content determined spans is to have one span per logical line of text.

If the editor is allowed to determine span sizes to its own best advantage the second issue is the size of the buffers. Again there are two possibilities: fixed size or variable size. Fixed size buffers are easier to manage and are often chosen to be some (low) multiple of the page size or the disk block size. A fixed size buffer implies a maximum span size (that is, the buffer size).

This table show the possibilities:

	Fixed size buffers	*Variable size buffers*
Content determined spans	Line size is limited	Line spans
Editor determined spans	Fixed size buffers	Piece tables

These are the three methods that will be examined in this section.

6.2 The line span method (one span per line)

Since most text editors do many operations on lines it seems reasonable to use a data structure that is line oriented so line operations can be handled efficiently. The line span method uses a descriptor for each line. Often one large buffer is used to hold all the line spans. New lines are appended to the end of the used part of the buffer. See Figure 5.

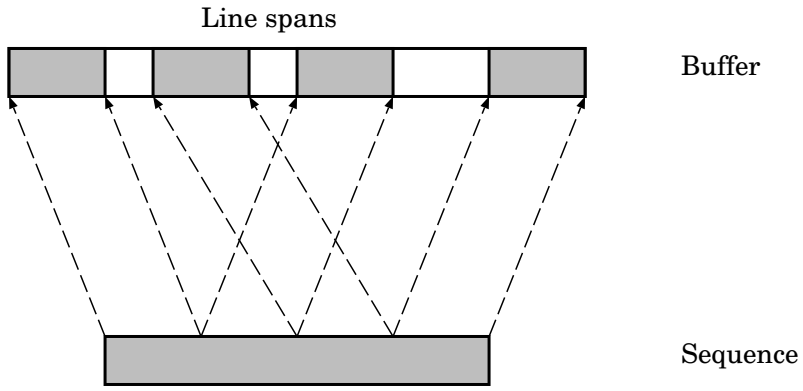

Figure 5: The line spans method

Line deletes are handled by deleting the line descriptor. Deleting characters within a line involves moving the rest of the characters in the line up to fill the gap. Since any one line is probably not that long this is reasonably efficient.

Line inserts are handled by adding a line descriptor. Inserting characters in a line involves copying the initial part (before the insert) of the line buffer to new space allocated for the line, adding the characters to be inserted, copying the rest of the line and pointing the descriptor at the new line. Multiple line inserts and deletes are combinations of these operations. Caching can make this all fairly efficient.

Usually new space is allocated at the end of the buffer and the space occupied by deleted or changed lines is not reused since the effort of dynamic memory allocation is not worth the trouble. A disk file the continues to grow at the end can be handled quite efficiently by most file systems.

This method uses as many descriptors as there are lines in the file, that is, a variable number of descriptors hence there is a recursive problem of keeping these descriptors in a sequence. Typically one of the basic methods described in section 5 is used to maintain the sequence of line descriptors. The linked list method can be used (as in Ved [6]) or the array method (as in Godot [11], Gina [1] and ed [3]).

NOTE: reference to SW Tools and SW Tools Sampler here. For linked lists of lines.

These simpler methods are acceptable since the number of line descriptors is much smaller than the number of characters in the file being edited. The linked list method allows efficient insertions and deletions but requires the management of list nodes.

This method is acceptable for a line oriented text editor but is not as common these days since strict line orientation is seen as too restrictive. It does require preprocessing of the entire buffer before you can begin editing since the line descriptors have to be set up.

6.3 Fixed size buffers

In the line spans method the partitioning of the sequence into spans is determined by the contents of the text file, in particular the line structure. Another approach is for the text editor to decide the partitioning of the sequence into spans based on the efficiency of buffer use. Since fixed size blocks are more efficient to deal with the file can be divided into a sequence of fixed size buffers. If the buffers were required to be always full, a great deal of time would be spent rearranging data in the buffers, therefore, each block has a maximum size but will usually contain fewer actual items from the sequence so there is room for inserts and deletes, which will usually affect only one buffer. (See Figure 6.)

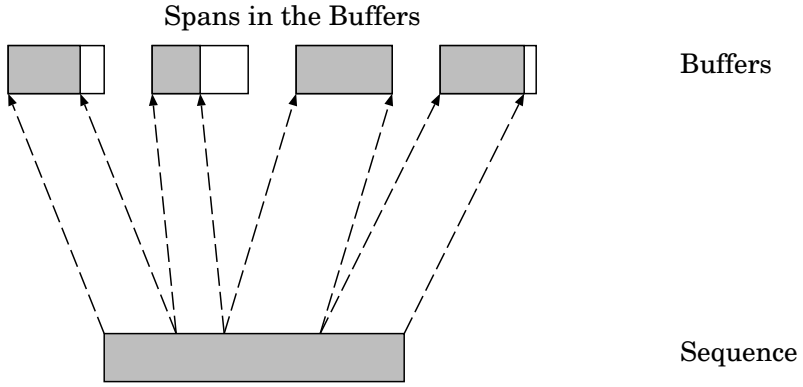

Figure 6: Fixed size buffers

The disk block size (or some multiple of the disk block size) is usually the most efficient choice for the fixed size buffers since then the editor can do its own disk management more easily and not depend on the virtual memory system or the file system for efficient use of the disk.

Usually a lower bound on the number of items in a buffer is set (half the buffer size is a common choice). This requires moving items between buffers and occasionally merging two buffers to prevent the accumulation of large numbers of buffers. There are four problems with letting too many buffers accumulate:

- wasted space in the buffers,
- the recursive sequence of descriptors gets too large,
- the probability that an edit will be confined to one buffer is reduced, and

- *ItemAt* caching is less effective.

As an example of fixed size buffers, suppose disk blocks are 4K bytes long. Each buffer will be 4K bytes long and will contain a span of length from 2K to 4K bytes. Each buffer is handled using the array method, that is, inserts and deletes are done by moving the items up or down in the buffer. Typically an edit will only affect one buffer but if a buffer fills up it is split into two buffers and if the span in a buffer falls below 2K bytes then items are moved into it from an adjacent buffer or it is coalesced with an adjacent buffer.

Each descriptor points to a buffer and contains the length of the span in the buffer. The fixed size buffer method also requires a recursive sequence for the descriptors. This could be any of the basic methods but most examples from the literature use a linked list of descriptors. The loose packing allows small changes to be made within the buffers and the fact that the buffers are linked makes it easy to add and delete buffers.

This method is used in the text editors Gina [1] and `sam` [12] and is described by Kyle [9].

6.4 The piece table method

In the piece table method the sizes of the spans are as large as possible but are split as a result of the editing that is done on the sequence. The sequence starts out as one big span and that gets divided up as insertions and deletions are made. We call each span a *piece* (of the sequence) and its descriptor is called a *piece descriptor*. The sequence of piece descriptors is called the *piece table*.

The piece table method uses two buffers. The first (the *file buffer*) contains the original contents of the file being edited. It is of fixed size and is read-only. The second (the *add buffer*) is another buffer that can grow without bound and is append-only. All new items are placed in this buffer.

Each piece descriptor points to a span in the file buffer or in the add buffer. Thus a descriptor must contain three pieces of information: which buffer (a boolean), an offset into that buffer (a non-negative integer) and a length (a positive integer[5]).

Figure 7 shows a piece table structure. The file consists of five pieces.

Initially there is only one piece descriptor which points to the entire file buffer. A delete is handled by splitting a piece into two pieces. One of these pieces points to the items in the old piece before the deleted item and the other piece points to the items after the deleted item. A special case occurs when the deleted item is at the beginning or end of the piece in which case we simply adjust the pointer or the piece length.

An insert is handled by splitting the piece into three pieces. The first piece points to the items of the old piece before the inserted item. The third piece points to the items of the old piece after the inserted item. The inserted item is appended to the end of the add file and the second piece points to this item. Again there are special cases for in insertion at the beginning or end of a piece.

If several items are inserted in a row, the inserted items are combined into a single piece rather than using a separate piece for each item inserted.

Figures 8, 9 and 10 show the effect of a delete and an insert in a piece table. Figure 8 shows the

[5] Normally zero length pieces are eliminated.

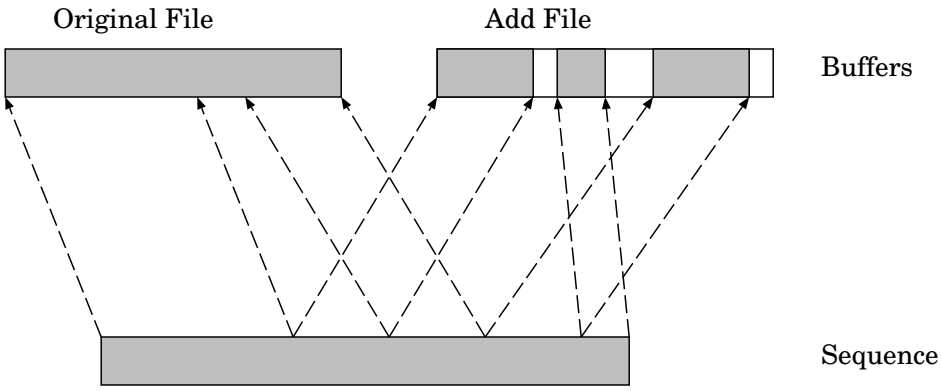

Figure 7: The piece table method

piece table after the file is read in initially. This is a very short file containing only 20 characters. Figure 9 shows the piece table after the word "large " has been deleted. Figure 10 shows the piece table after the word "English " has been added. Notice that, in general, a delete increases the number of pieces in the piece table by one and an insert increases the number of pieces in the piece table by two.

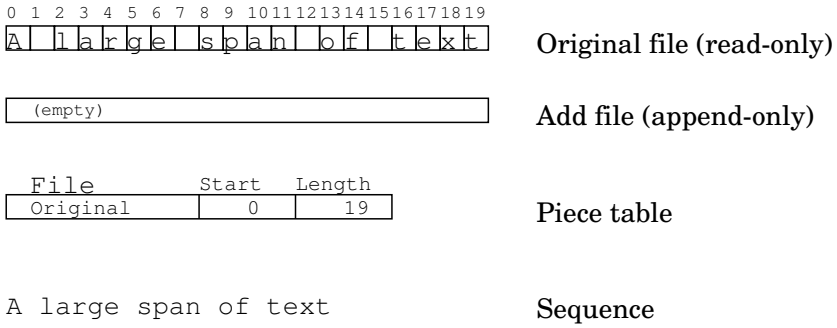

Figure 8: A piece table before any edits

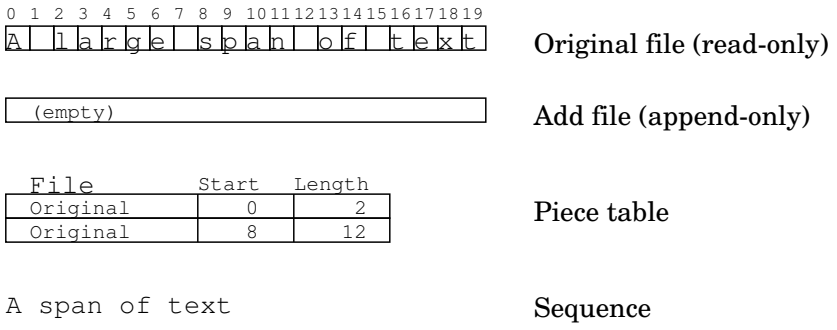

Figure 9: A piece table after a delete

Let us look at another example. Suppose we start with a new file that is 1000 bytes long and make the following edits.

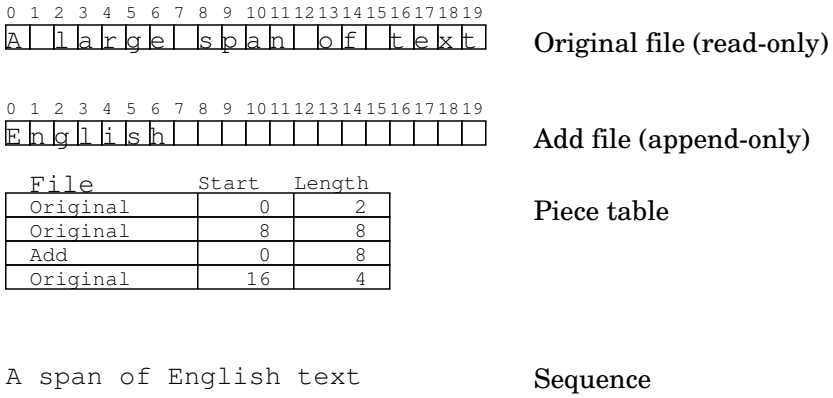

Figure 10: A piece table after a delete and an insert

1. Six characters inserted (typed in) after character 900.
2. Character 600 deleted.
3. Five characters inserted (typed in) after character 500.

The piece table after these edits will look like this:

file	*start*	*length*	*logical offset*
orig	0	500	0
add	6	5	500
orig	500	100	505
orig	601	300	605
add	0	6	905
orig	901	100	911

The "logical offset" column does not actually exist in the piece table but can be computed from it (it is the running total of the lengths). These logical offsets are not kept in the piece table because they would all have to be updated after each edit.

The piece table method has several advantages.

- The original file is never changed so it can be a read-only file. This is advantageous for caching systems since the data never changes.
- The add file is append-only and so, once written, it never changes either.
- Items never move once they have been written into a buffer so they can be pointed to by other data structures working together with the piece table.
- Undo is made much easier by the fact that items are never written over. It is never necessary to save deleted or changed items. Undo is just a matter of keeping the right piece descriptors around. Unlimited undoes can be easily supported.

- No file preprocessing is required. The initial piece can be set up only knowing the length of the original file, information that can be quickly and easily obtained from the file system. Thus the size of the file does not affect the startup time.
- The amount of memory used is a function of the number of edits not the size of the file. Thus edits on very large files will be quite efficient.

The above description implies that a sequence must start out as one big piece and only inserts and deleted can add pieces. Following this rule keeps the number of pieces at a minimum and the fewer pieces there are the more efficient the *ItemAt* operations are. But the text editor is free to split pieces at other times to suit its purposes.

For example, a word processor needs to keep formatting information as well as the text sequence itself. This formatting information can be kept as a tree where the leaves of the tree are pieces. A word in bold face would be kept as a separate piece so it could be pointed to by a "bold" format node in the format tree. The text editor Lara [8] uses piece tables this way.

As another example, suppose the text editor implements hypertext links between any two spans of text in the sequence. The span at each end of the link can be isolated in a separate piece and the link data structure would point to these two pieces.[6] This technique is used in the Pastiche text editor [5] for fine-grained hypertext.

These techniques work because piece table descriptors and items do not move when edits occur and so these tree structures will be maintained with little extra work even if the sequence is edited heavily.

Overall, the piece table is an excellent data structure for sequences and is normally the data structure of choice. Caching can be used to speed up this data structure so it is competitive with other data structures for sequences.

Piece tables are used in the text editors: Bravo [10], Lara [8], Point [4] and Pastiche [5]. Fraser and Krishnamurthy [7] suggest the use of piece tables as a way to implement their idea of "live text".

7 A General Model of Sequence Data Structures

The discussions in the previous two sections suggest that it is possible to characterize all sequence data structures given the following assumptions:

1. The computer has main memory with sequentially addressed cells and has disk memory with sequentially addressed blocks of sequentially addressed cells.
2. Items have a fixed size in memory and on disk.
3. Items are stored directly in the memory or on disk, that is, they are not encoded. Hence every item must exist somewhere in memory or on disk.

[6] There are some details to deal with to make this all work but they are easy to handle.

4. The main memory is of limited size and hence cannot hold all the items in large sequences.[7]
5. The environment provides dynamic memory allocation (although some sequence data structures will do their own and not use the environment's dynamic memory allocation).
6. The environment provides a reasonably efficient file system for item storage that provides files of (for all practical purposes) unlimited size.

The following concepts are used in the model.

- An *item* is the basic component of our sequences. In most cases it will be a character or byte (but it might be a descriptor in a recursive sequence data structure).
- A *sequence* is an ordered set of items. During editing, items will be inserted into and deleted from the sequence. The items in the sequence are *logically contiguous.*
- A *buffer* is some contiguous space in main memory or on disk that can contain items (Assumption 4). All items in the sequence are kept in buffers (Assumption 3). Consecutive items in a buffer are *physically contiguous.*
- A *span* is one or more items that are logically contiguous in the sequence and are also physically contiguous in the buffer. (Assumption 1)
- A *descriptor* is a data structure that represents a span. Usually the descriptor contains a pointer to the span but it is also possible for the descriptor to contain the buffer that contains the span.

A *sequence data structure* is either

- A basic sequence data structure which is one of:
  - An array.
  - An array with a gap.
  - A linked list of items.
  - A more complex linked structure of items.
- A recursive sequence data structure which comprises:
  - Zero or more buffers each of which contains zero or more spans.
  - A (recursive) sequence data structure of zero or more descriptors to spans in the buffers.

This model is recursive in that to implement a sequence of items it is necessary to implement a sequence of descriptors. This recursion is usually only one step, that is, the sequence of descriptors in implemented with a basic sequence data structure. The deficiencies of the basic sequence data structures for implementing character sequences are less critical for sequences of descriptors since there are usually far fewer descriptors and so sophisticated methods are not required.

[7] Even if virtual memory is provided there will be an upper bound on it in any actual system configuration. In addition, most sophisticated sequence data structures do not rely on virtual memory to efficiently shuttle sequence data between main memory and the disk. Usually the program can do better since it understands exactly how the data is accessed.

7.1 The design space for sequence data structures

This model can be used to examine the design space for sequence data structures.

The four basic methods seem to cover most of the useful cases. A basic method is one where the number of descriptors is fixed. The array method uses two descriptors and the gap method uses three. We could generalize this to a two gap method using four descriptors and so on but it is not clear that there is any advantage in doing that. The linked list method is basic even though it uses a descriptor for every item in the sequence. The items are linked together so one descriptor serves to define the sequence. As soon as we try to put two or more items into a descriptor it becomes an instance of the recursive fixed size buffer method.

Using a more complex linked structure than a linked list will make reading items (*ItemAt*) more efficient but makes *Insert* and *Delete* less efficient. Since *ItemAt* access is nearly always sequential this tradeoff is not advantageous.

The recursive methods divide into two types. The first uses fixed size buffers and the second uses variable sized buffers.

There are two issues in the fixed size buffer method The first is the method used in maintaining the items in each fixed size buffer and the second is the method of maintaining the sequence of buffers.

The items in a single (fixed size) buffer are a sequence. The typical implementation keeps them as an array at the beginning of the buffer. In general the gap method is superior to the array method, thus it might be useful to keep characters in a single buffer using the gap method where the gap is kept at the last edit point. This should halve the expected number of bytes moved for one edit at little additional program cost. If the edits exhibit locality (as they typically do) the advantage will be greater.

A linked list is usually used to implement the sequence of descriptors (buffer pointers) and this is generally a good method because of the ease of inserting and deleting descriptors (which are frequent operations). One problem is the extra space used by the links but this is only a problem if there are lots of descriptors. With 1K blocks and editing 5M of files, this is still only 5K descriptors with 10K links. Thus this problem does not seem to be significant in practical cases.

Another issue is virtual memory performance. Linked lists do not exhibit good locality. If the descriptors were kept using the gap method, locality would be improved considerably. Assuming a descriptor takes four words (one for the pointer to the span, one for the length of the span and two for the links) the 5K descriptors would consume 20K words or 80K bytes (assuming four bytes per word). Again this is small enough that virtual memory performance would probably not be a significant problem, but if it were, the gap method could improve things.

The piece table method uses fewer descriptors than the fixed buffer method initially (before any editing) but heavy editing can create numerous pieces. There are advantages to maintaining the pieces since it allows easy implementation of unlimited undo. In addition, pieces can be used to record structures in the text (as described in section 7.1). As a consequence there might be many pieces. This means the problems presented above in the discussion of fixed size buffers (space consumed by links and virtual memory performance) might be significant here. That is, the gap method of keeping the piece table might be preferred.

Another generalization would be to use two levels of recursion, that is, to use one of the recursive

sequence data structures to implement the sequence of descriptors. The recursive methods are beneficial when the sequences are quite large so we might use a two-level recursive method if the number of descriptors was quite large. As we mentioned above, this might be the case with a piece table.

So there are four new variations that we have uncovered in this analysis.

- The fixed size buffers method using the gap method inside each buffer.
- The fixed size buffers method using the gap method for descriptors. This might be better if virtual memory performance was a consideration.
- The piece table method using the gap method for descriptors. This might be better if virtual memory performance was a consideration.
- A two level recursive method that uses a recursive method to maintain the sequence of descriptors. This would suitable if there is a very large number of descriptors.

8 Experimental comparison of sequence data structures

In order to compare the performance of these data structures I implemented each of them and a simulator program that would simulate typical editing behavior. The simulator has a number of parameters that I will discuss in presenting the results. I ran these tests on a several machines (Micro VAX III/VAX, SUN3/M68020, SparcStation 2/SPARC 2, DEC 5000/MIPS 3000), under two compilers (cc and gcc) and with maximum optimization. The results for the different architectures and compilers were all similar. Most of the results in this section were obtained using gcc and gprof on a SUN 3/60.

The measurements were made with the following parameters (except where one of these parameters is being experimentally varied):

- Sequence length of 8000 characters
- Block size of 1024 characters
- Fixed buffer methods keep buffers at least half full (from 512 to 1024 characters)
- The location of 98% of the edits is normally distributed around the location of the previous edit with a standard deviation of 25
- The location of 2% of the edits is uniformly distributed over the entire sequence
- After each edit, 25 characters on each side of the edit location are accessed
- Every 250 edits the entire file is scanned sequentially with *ItemAts*

The sequence data structures measured (and their abbreviated names) are:

- Null — the null method that does nothing. This is for comparison since it measures the overhead of the procedure calls.
- Arr — The array method.
- List — The list method.
- Gap — The gap method.
- FsbA – The fixed size buffer method with the array method used to maintain the sequence inside each buffer.
- FsbAOpt – The fixed size buffer method with the array method used to maintain the sequence inside each buffer and with *ItemAt* optimized.
- FsbG – The fixed size buffer method with the gap method used to maintain the sequence inside each buffer.
- Piece – The piece table method.
- PieceOpt – The piece table method with *ItemAt* optimized

I will present graphs for *Insert* and *ItemAt* operations. The *Delete* operation takes about the same time as the *Insert* operation for all these sequence data structures.

Figure 11 shows how the speed of the *ItemAt* operation is affected by the size of the sequence. It basically has no effect except for the interesting result that *ItemAt* operation for the PieceOpt

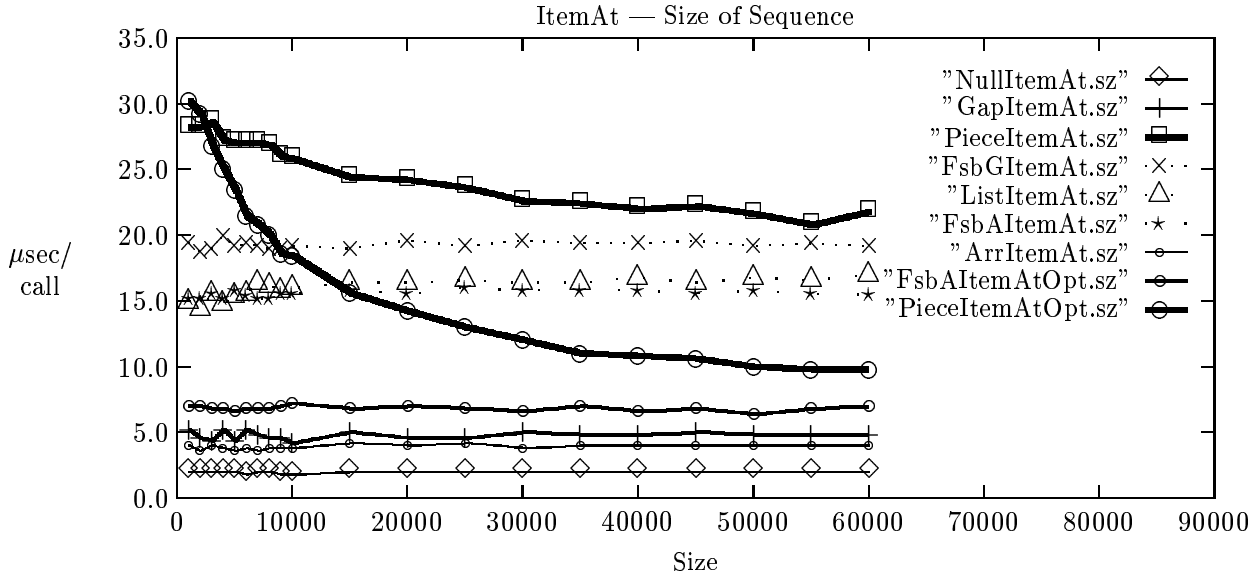

Figure 11: *ItemAt* times as the length of the sequence varies

method gets faster for larger arrays. The reason for this is that for longer sequences the caching used

in the optimization becomes more effective. Each *ItemAt* is faster although (since the sequences are longer) *ItemAts* is called many more times.

The Arr method is the fastest and is nearly as fast as the Null method. The Gap method is only a little slower. The FsbA method is much slower and is about the same as the List method, the FsbG method and the Piece method. The optimized FsbA method is nearly as fast as the Gap method and the PieceOpt method gets close. Even so, the PieceOpt *ItemAt* is half the speed of the Arr *ItemAt*. Since *ItemAt* is such a frequent operation it is necessary to optimize (with caching) all the methods except for the Arr and the Gap method.

Figure 12 shows how the speed of the *Insert* operation is affected by the size of the sequence. It

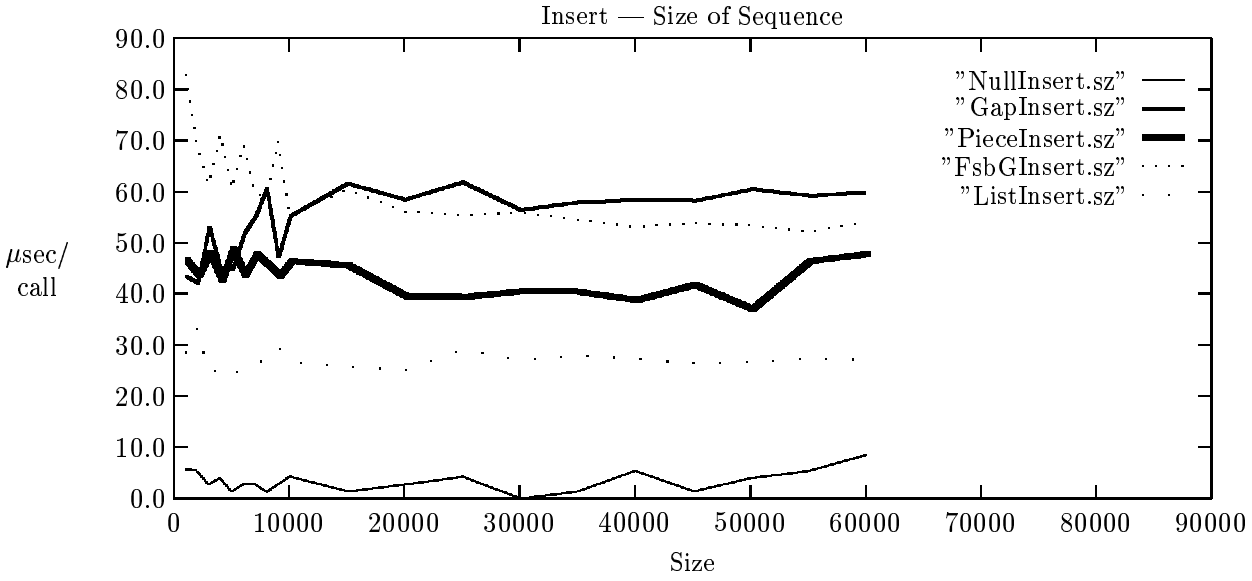

Figure 12: *Insert* times as the size of the sequence varies

has no effect except for shorter sequences. The List method is the fastest and the FsbG, Gap and Piece methods are all about half its speed.

The Arr and FsbA methods are not shown on this graph because they are so much slower that they would distort the graph (as the next two graphs show). Figure 13 includes the FsbA method which is an order of magnitude slower than the other methods (for the *Insert* operation). Figure 14 includes the Arr method which is two orders of magnitude slower than the other methods. Note that it gets slower linearly with the size of the sequence, as one would expect.

8.1 Sensitivity to parameters

Figure 15 shows how changes in the standard deviation of the normal distribution affect the *Insert* operation for the various methods. Only the Gap method and the FsbG are affected and only at much higher standard deviations that one would expect in normal text editing. The Arr and FsbA methods are not shown (because they are too large) but they are unaffected by increases in the

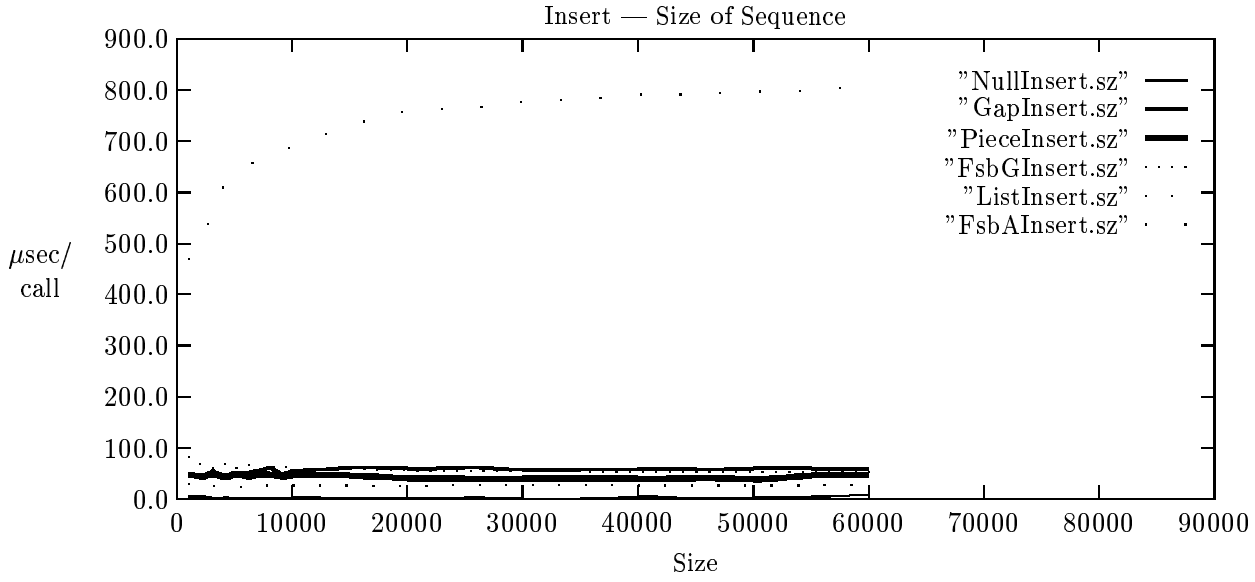

Figure 13: *Insert* times as the size of the sequence varies

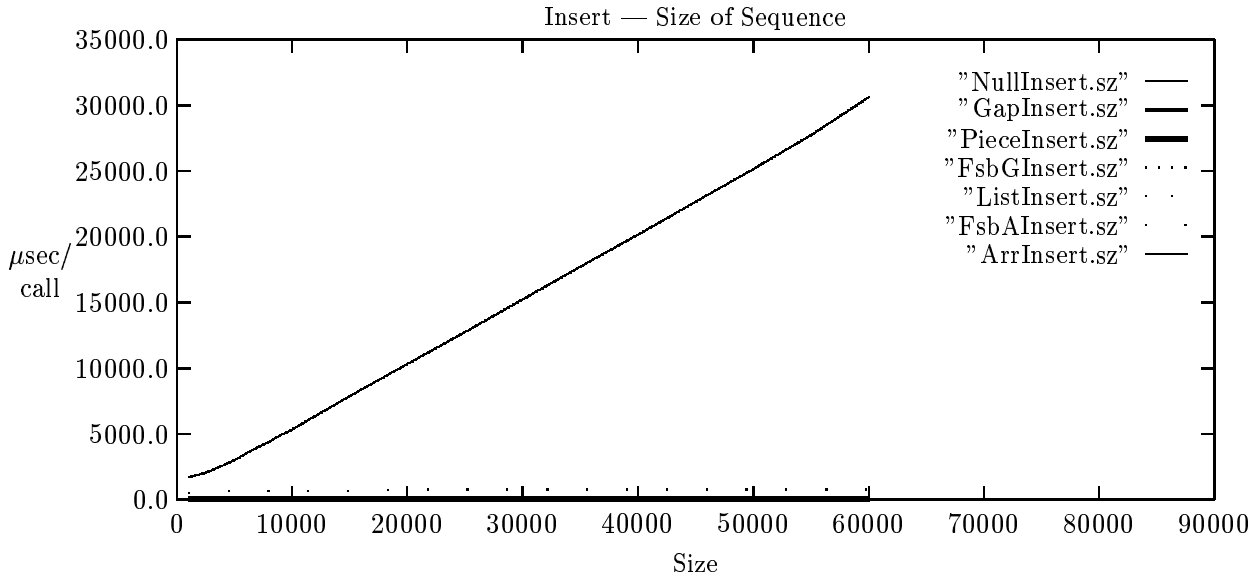

Figure 14: *Insert* times as the size of the sequence varies

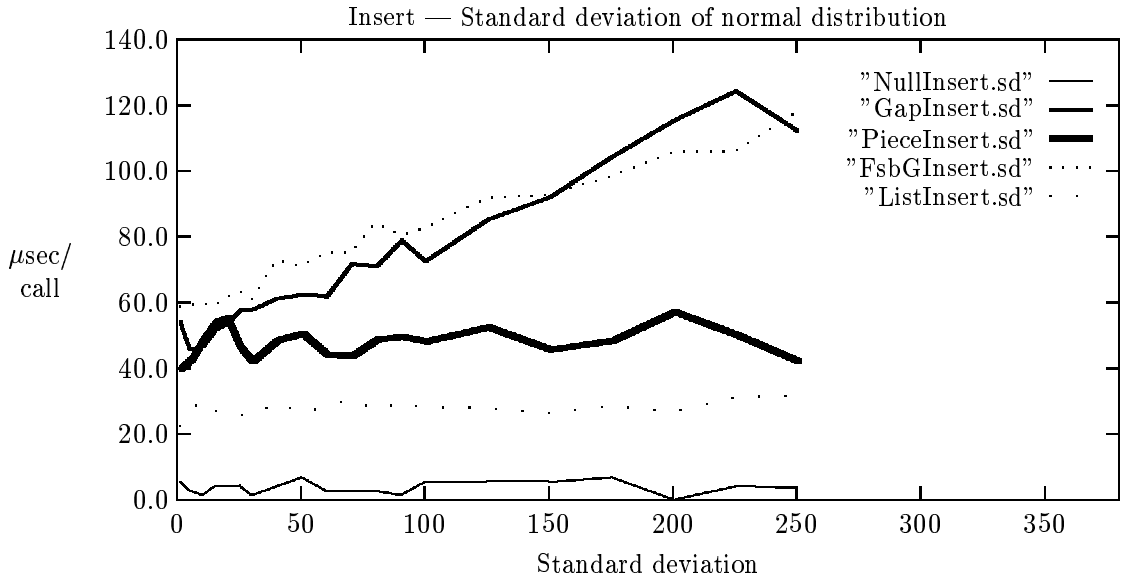

Figure 15: *Insert* times as the standard ceviation varies

standard deviation of the normal distribution.

The *ItemAt* operation is unaffected by increases in the standard deviation of the normal distribution.

Figure 16 shows how the *Insert* operation is affected by changes in the percent of edit locations that are taken from a uniform distribution over the entire sequence (that is, where the next edit is randomly located in the sequence instead of instead of being normally distributed around the location of the previous edit). Only the Gap method is affected.

Figure 17 shows how the *ItemAt* operation is affected by changes in the percent of edit locations that are taken from a uniform distribution over the entire sequence (that is, where the next edit is randomly located in the sequence). Only the Piece and List methods are affected but only in ranges that one would not expect to find in normal text editing.

Figure 18 shows how the buffer size affects the time taken by the *Insert* operation in the FsbA and FsbG methods. The FsbG method is unaffected by the buffer size while the FsbA method goes up linearly (and sharply) as the buffer size increases. The increase levels off at 8000 where the buffer size is equal to the sequence size and so the entire sequence is in one buffer and the method has degenerated into the Arr method.

The following table gives the general trends of the results. The units vary from machine to machine but the ratios were reasonably steady. Some of the results have wide ranges. This means that the figure depends on one or more of: the size of the file being editing, the distribution of the position of edits in the sequence, and the size of the buffers (for the Fsb method).

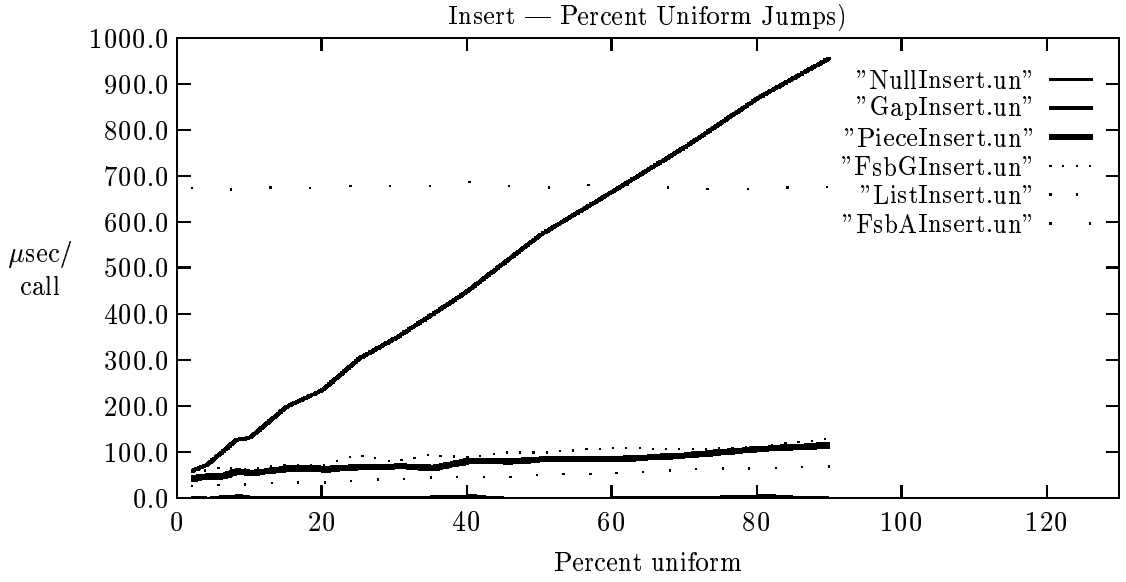

Figure 16: *Insert* times as the percent of uniform jumps varies

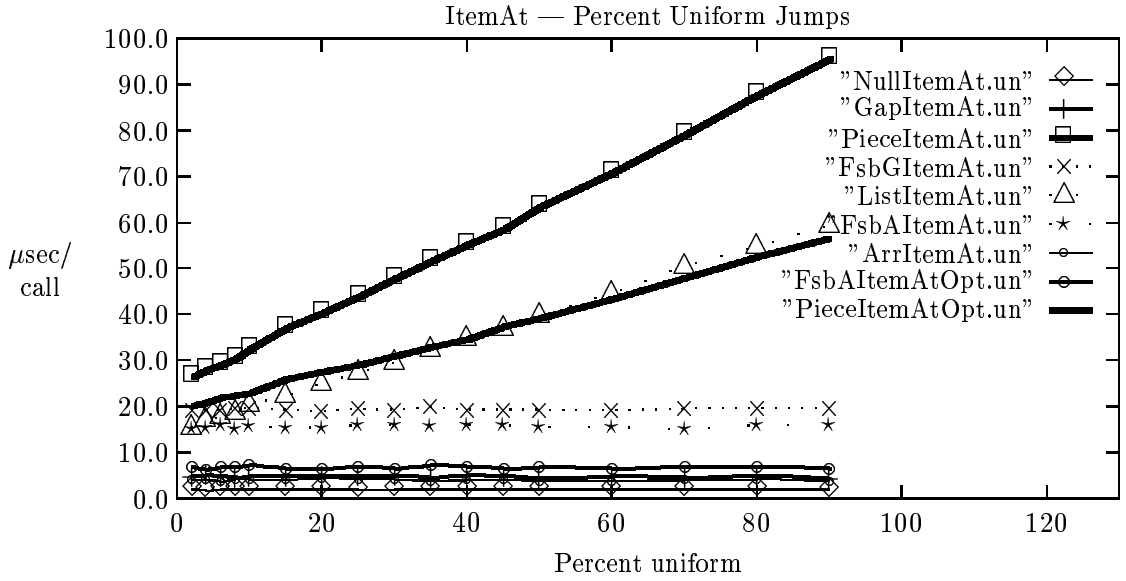

Figure 17: *ItemAt* times as the percent of uniform jumps varies

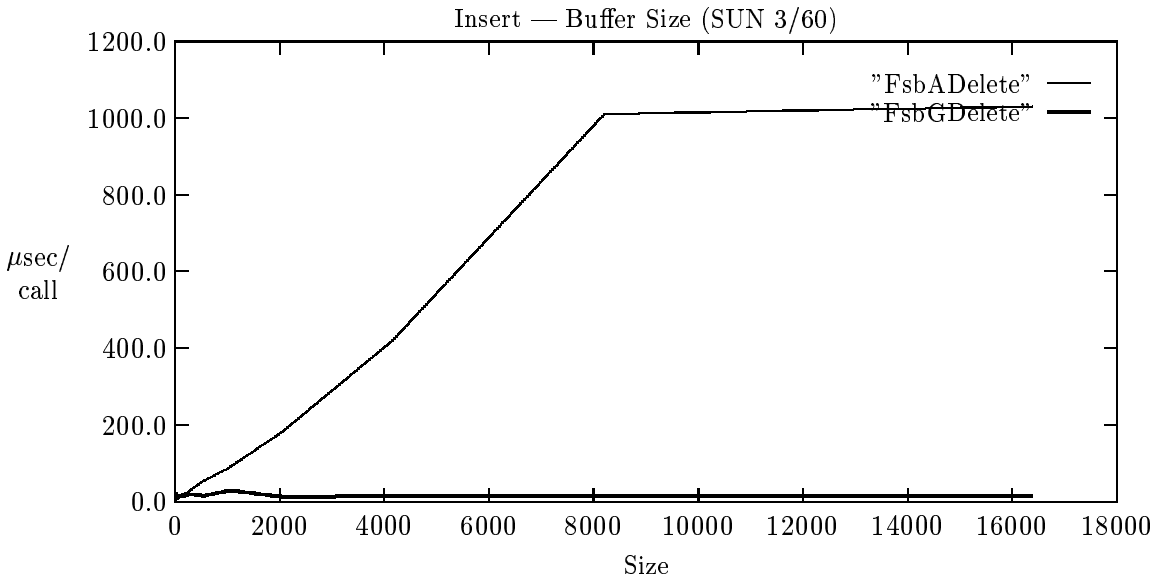

Figure 18: *Insert* times as the size of the buffer increases

Method	*ItemAt*	*Delete*	*Insert*
Null	0.5	0.5	0.5
Array	1.0	400–2000	400–2000
Gap	1.5	20–60	20–70
Linked List	4.5	6–8	9–16
FSB-Array	4.0	30–180	20–120
FSB-Array-Opt	1.8	30–180	20–120
FSB-Gap	4.5	18–20	35–50
Piece	6.7	12	15–20
Piece-Opt	5.2	12	15–20

8.2 Discussion of the timing results

Only time will be considered in the following discussion. In the next section these sequence data structures will be compared on a range of criteria. Remember that the *ItemAt* times assume a very high locality of reference. If the references were not local the *ItemAt* times would be much higher and the ratios would be different.

The Arr method has the fastest *ItemAt* but is terrible for *Insert* and *Delete*. It is not a practical method.

The Gap method is nearly as fast for *ItemAt* but is also quite fast for *Insert* and *Delete*. The Gap method is a generally fast method but some other problems as will be seen in the next section.

The List method has the fastest *Inserts* and *Deletes* by far but a fairly slow *ItemAt*. The List method uses a lot of extra space (two pointers per item). It is useful for in-memory sequences with large items.

The FsbA method is slow for *Inserts* and *Deletes* but its *ItemAt* can be made quite fast with some simple caching. It is possible to reduce the *ItemAt* time even further by making it an inline operation. The FsbG method reduces the *Insert* and *Delete* times radically but the *ItemAt* time is a bit higher. The equivalent *ItemAt* caching would be a little more complicated and a little slower.

The Piece method has excellent *Insert* and *Delete* times (only slightly slower than the linked list method) but its *ItemAt* time is quite slow even with simple caching. More complex *ItemAt* caching that avoids procedure calls is necessary when using the Piece method. The idea of the caching is simple. Instead of requesting an item you request the address and length of the longest span starting at a particular position. Then all the items in the span can be accessed with a pointer dereference (and increment). This will bring the *ItemAt* time down to the level of the array and gap methods.

8.3 Experimental comparison of memory use

TDB

9 Comparison of Sequence Data Structures

9.1 Basic sequence data structures

The array method is just too slow for *Inserts* and *Deletes*. In addition its paging behavior is very bad (it touches lots of pages). It might be useful for a one line text editor or a text editor where few edits are expected.

The linked list method takes far too much space for long sequences. It is useful for short in-memory sequences where the edits and *ItemAts* are not localized. The linked list method has one great advantage and that is that the items never move in memory. This makes it easy to embed a list sequence in another data structures (such as a tree to provide fast searching).

Of the basic sequence data structures, only the gap method can be seriously considered for a general purpose text editor. The gap method has one major problem and that is when the gap fills up. This will require lots of item movement to reestablish the gap.

REDO: be more positive about the gap method.

	Array	*Gap*	*Linked List*
Time	Slow	Fast	Fast
Space	Low	Low	Very high
Ease of programming	Easy	Easy	Easy
Size of code	Low (39 lines)	Low (59 lines)	Medium (79 lines)

The lines of code measure was taken from the sample implementations.

9.2 Recursive sequence data structures

The line span method is an older method that has little to recommend it in modern text editors.

The Fsb methods and the Piece method are both good choices for professional-quality text editors. Both methods:

- are acceptably fast if caching is used
- handle large files without slowing down
- handle many of files without slowing down
- are efficient in their use of space
- allow efficient buffer management
- provide excellent locality of buffer use

Overall however the piece table method seems to be better. It has a number of advantages:

- All buffers except one are read-only and the other buffer is append-only. (Thus the buffers are easy to cache and work well over a network.)
- The code is quite simple. (The code for Fsb is complicated by the need to balance buffers.)
- Huge files load as quickly as tiny files. (No preprocessing is required for large files so they load quickly.)
- Disk buffers are always full of data (rather than 3/4 full—on the average—as they are in the fixed buffer method). Thus disk caching is more efficient.
- Items never move once they are placed.

The last point is important. If the piece sequence is kept as a list then the pieces never move either. This allows the sequence to be pointed to by other data structures that record additional information about the sequence. For example it is fairly easy to implement "sticky" pointers **[Reference to Fisher and Ladner]** (that is, pointers that point to the content of the sequence rather than relative sequence positions). For example we might want to attach a sticky pointer to the beginning of a procedure definition. Such a facility would be useful in implementing a "tag" facility such as the one found in Unix[2].

As another example, the text editor Lara [8] also formats its text. It keeps the formatting state in a tree structure where pieces are the leaves of the tree. Inserts and deletes require very little bookkeeping because the items and pieces never move around when you use a piece table.

	FSB-Array	*FSB-Gap*	*Piece*
Time	Fairly fast	Fast (with caching)	Fairly fast (with caching)
Space	Low	Low	Low
Ease of programming	Hard	Hard	Medium
Size of code	Medium to large (218 lines)	Large (301 lines)	Medium (162 lines)

10 Conclusions and Recommendations

This paper has two purposes:

- to examine systematically data structures for sequences and
- to present the piece table method and describe its advantages.

A review of the literature shows that there are only a few different data structures for text sequences that have been used in text editors. A careful examination of the design space showed that there really are not that many fundamental types of sequence data structures.

Sequence data structures are divided into two categories. Basic sequence data structures (array, gap and linked list) and recursive sequence data structures (line spans, fixed size buffers and piece tables).

The array method is the obvious one: keep the text in an array of characters. The gap method is similar but it keeps a gap in the middle of the array at the text editor insertion point. The linked list method keeps the characters on a linked list.

The recursive methods keeps the text in a number of separate "spans" and keeps track of a sequence of pointers to these spans. The line span method uses a span for each line. The fixed size buffer method keeps a sequence of fixed size buffers each of which contains one span. The piece table method uses spans of any size either in the original file or in a file for added characters.

In examining these data structures a general model of sequence data structures was formulated and used this model and the examples were used to discover several new variations for sequence data structures that might improve performance in some situations.

A series of experiments was performed on these data structures to determine their relative performance and they were compared on a variety of criteria including time. The main conclusion is that the piece table structure is the best data structure for text sequences although some of the other methods might be useful in certain cases.

The piece table method has a number of advantages and is an especially good method for text with additional structure. Thus it would be the best choice for a word processor or a editor with hypertext facilities.

References

[1] C. C. Charlton and P. H. Leng. Editors: two for the price of one. *Software—Practice and Experience*, 11:195–202, 1981.

[2] Computer System Research Group, EECS, University of California, Berkeley, CA 94720. *UNIX User's Reference Manual (4.3 Berkeley Software Distribution)*, April 1986.

[3] Computer System Research Group, EECS, University of California, Berkeley, CA 94720. *UNIX User's Supplementary Documents (4.3 Berkeley Software Distribution)*, April 1986.

[4] C. Crowley. The Point text editor for X. Technical Report CS91-3, University of New Mexico, 1991.

[5] C. Crowley. Using fine-grained hypertext for recording and viewing program structures. Technical Report CS91-2, University of New Mexico, 1991.

[6] B. Elliot. Design of a simple screen editor. *Software—Practice and Experience*, 12:375–384, 1982.

[7] C. W. Fraser and B. Krishnamurthy. Live text. *Software—Practice and Experience*, 20(8):851–858, August 1990.

[8] J. Gutknecht. Concepts of the text editor Lara. *Communications of the ACM*, 28(9):942–960, September 1985.

[9] J. Kyle. Split buffers, patched links, and half-transpositions. *Computer Language*, pages 67–70, December 1989.

[10] B. W. Lampson. *Bravo Manual in the Alto User's Handbook*. Xerox Palo Alto Research Center, Palo Alto, CA, 1976.

[11] I. A. MacLeod. Design and implementation of a display oriented text editor. *Software—Practice and Experience*, 7:771–778, 1977.

[12] R. Pike. The text editor `sam`. *Software—Practice and Experience*, 17(11):813–845, November 1987.