other binary files. It can play sounds, thus it can be made to read the text by adding *TextToSpeech* software from *Apple*.

Notepad: *Windows* Notepad is simpler than SimpleText. It offers many of the same features: the ability to cut, copy, paste, and find (but not replace). One can set fonts and styles within the document, but these are for display only, as the output is plain text, which means no document formatting code is created from within the program.

7.1 DESIGN OF A TEXT EDITOR

In this section, we will discuss the design of a text editor. Apparently, a text editor consists of a main window along with some edit controls. However, the other part of it is the text file being edited. The actual file resides in the disk in some format that may be totally different from the way it is shown in the edit window. As shown in Fig. 7.1, the key components of a text editor are the following:

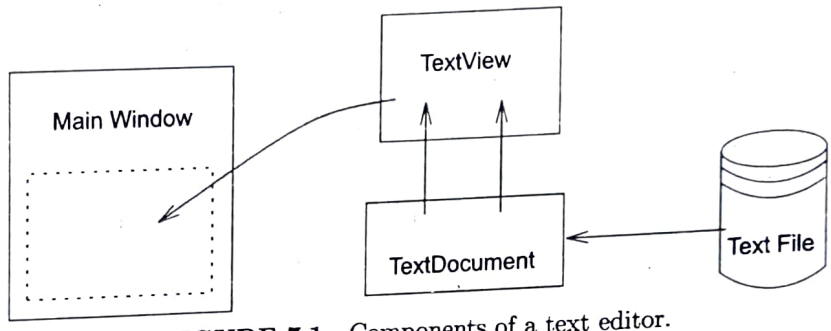

FIGURE 7.1 Components of a text editor.

1. *Main Window:* The top-level main window generally contains the title of the file being edited, menu, and status bars. It does not know how to edit text files, its only purpose is to provide an interface to the user.

2. *TextView:* TextView is the most important component of a text editor. It is generally designed as a separate window that is a child of the *Main Window*. The *TextView* is the visual component of a text editor. Its primary purpose is to display and edit text. However, it needs to do a number of other things, such as displaying scrollbars, process mouse and keyboard inputs, and support drag and drop.

3. *TextDocument:* It is not a visible component, but is used to store and manipulate a text file, once it has been loaded into the memory. It does not have any concept of windows, mice, drawing or painting. All that it knows is how to manipulate text, and supply text to the edit control when it wants to draw something.

4. *Text File:* Though it is not really a part of the text editor, it plays an important role nonetheless. It is the disk file that stores the text we want to edit. The TextDocument will interface directly with this text file, reading from and writing to it when it is time to load or save a new document.

It is important to note that separation of *TextView* from the data it stores (that is, *TextDocument*) provides the flexibility to modify the way we store and represent the actual data in the text file without impacting upon the design of visual component. Similarly, the visual component can be modified without any necessity to modify the storage organization of the text file. The interface between the components may also vary. For example, the *TextView* is a line-oriented, graphical entity. Most of the work performed by the edit-window will be updating the display. This will entail retrieving data from the *TextDocument* on a line-by-line basis. On the other hand, the *TextDocument* will either load the entire file into memory in one go, or access the file in manageable chunks.

7.1.1 Common Edit-Control Features

The edit-control features are the controls available to the user of the editor. Thus, before concentrating on the design of the editor, these features are to be fixed. The features present in most of the modern text editors include the following:

1. Unlimited file size and line length editing with little (or no) degradation in performance for larger files.
2. Fast, smooth graphic display. This includes text selection and smooth scrolling with no artifacts or glitches.
3. Syntax colouring. This shows the various components of a programming language (such as keywords, variables, constants) in different colours to enable their quick identification on seeing a code segment.
4. Full Clipboard and Drag-and-Drop support.
5. Single- or Multi-font display.
6. Complete Undo and Redo support.

7.2 DATA STRUCTURES FOR TEXT SEQUENCES

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. In a *sequence*, reading an item (by position number) is extremely localized. There are several data structures that can be used to store sequences, starting from the most common ones like arrays, link list, etc. to complex ones that keep the *sequence* as a recursive sequence of spans of text. To judge the relative merits and demerits of these schemes, we have to first understand the type and frequency of operations carried out on such sequences.

- *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, as to look through the large file, the entire file needs to be converted to the internal form first. Thus, editors not requiring this operation can interleave the file reading with text editing—the user need not worry how big a file is getting loaded into the memory.

- *Close* operation is also infrequent and not normally an important factor in comparing sequence data structures.
- *ItemAt* operation is of course very frequent, but it is also very localized due to the fact that text editors almost always access characters sequentially. When the screen is drawn (showing the file content on the window), the characters on the screen are accessed sequentially beginning with the first character on the screen. If the user pages forward or backward, the characters are accessed sequentially. Searches normally begin at current position and proceed forward or backward sequentially. Overall, there is almost no random access in a text editor.
- *Edits* (*inserts* and *deletes*) are the most important operations. The sequence data structures should be optimized for edits. In a typical text editing, the display is changed after each *insert* and *delete* and this requires a number of *ItemAts*, often just a few characters around the edit but possibly the whole rest of the window.

Next, we introduce a few terminologies that will be used hereafter.

1. An *item* is the basic element. Usually it will be a character.
2. A *sequence* is an ordered set of items.
3. Sequential items in a sequence are said to be *logically contiguous*.
4. Items of the sequence are kept 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.
5. Sequentially addressed items in a buffer are *physically contiguous*.
6. When a string of items is physically contiguous in a buffer and is also logically contiguous in the sequence, we call it a *span*.
7. A *descriptor* is a pointer to a span.

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.

7.2.1 Basic Sequence Data Structures

In this section, we will discuss about simple sequence data structures that use a fixed number of external descriptors.

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 the other one for the buffer. The buffer contains the items of the sequence in a physically contiguous order. *Deletes* are handled by moving all 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. The array method has been shown in Fig. 7.2. Naturally, this would not be an efficient data structure if a lot of editing has to be done. However, its simplicity makes it a reasonable choice in situations where a few inserts and deletes are made (for example, read-only sequence), or the sequences are relatively small (for example, one-line text editor).

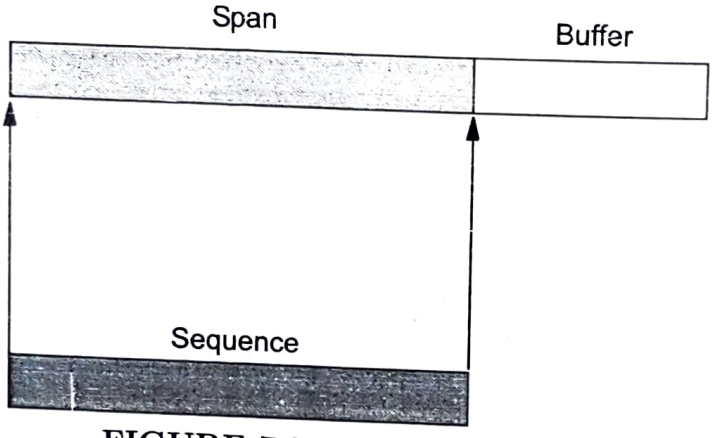

FIGURE 7.2 The array method.

The gap method: two spans in one buffer. This method is slightly more complex than the array method, however much more efficient. In this method, we 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. 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 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. The structure has been shown in Fig. 7.3. When the cursor moves, the gap is also moved. Thus, 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. Moreover, the gap does not need to move every time the cursor is moved. Only when an editing operation is requested, the gap is moved. Thus, moving the cursor around the file while paging or searching will not cause unnecessary gap movements. If the gap fills up, the second span is moved in the buffer to create a new gap. Gap size is determined through some algorithm. In practice, it is usually increased by some fixed increment or by some fixed percentage of the current buffer size. With virtual memory, one can make the buffer so large that it is unlikely to fill up. The gap method is simple and surprisingly efficient. It is also known as the *split buffer* method.

The linked list method. This method has a span for each item in the sequence and the span is kept in the descriptor. This requires a large number of descriptors which must be sequenced. If the descriptors are linked together into a linked list, then only the

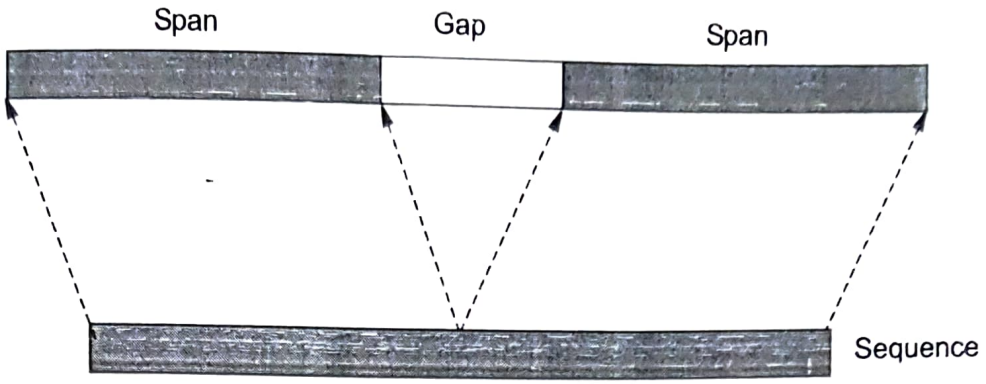

FIGURE 7.3 The gap method.

head of the list is needed to be remembered. *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. The structure has been shown in Fig. 7.4.

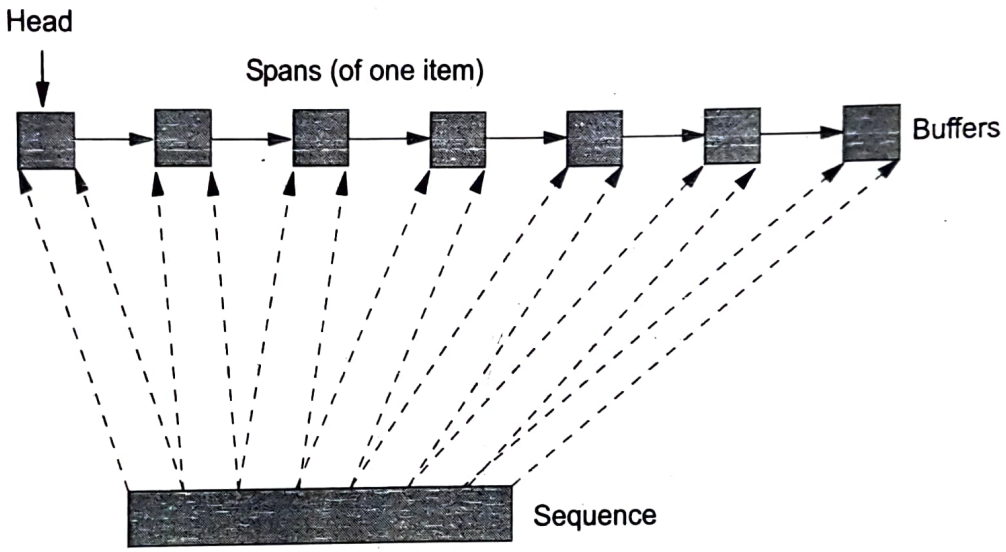

FIGURE 7.4 The linked list method.

7.2.2 Recursive Sequence Data Structures

In this section, we present a few data structures that use variable number of descriptors and so must be using a recursive structure.

The line span method: one span per line. Since most of the text editors do many operations on lines, it seems reasonable to use a data structure that is line oriented, so that 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. Figure 7.5 shows the line span 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 the deleted or changed lines are not reused since the effort of dynamic memory allocation is not worthwhile. A disk file that continues to grow at the end can be handled quite efficiently by most file systems. The line span 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 data structures described in the previous section is used to maintain the sequence of line descriptors. 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.

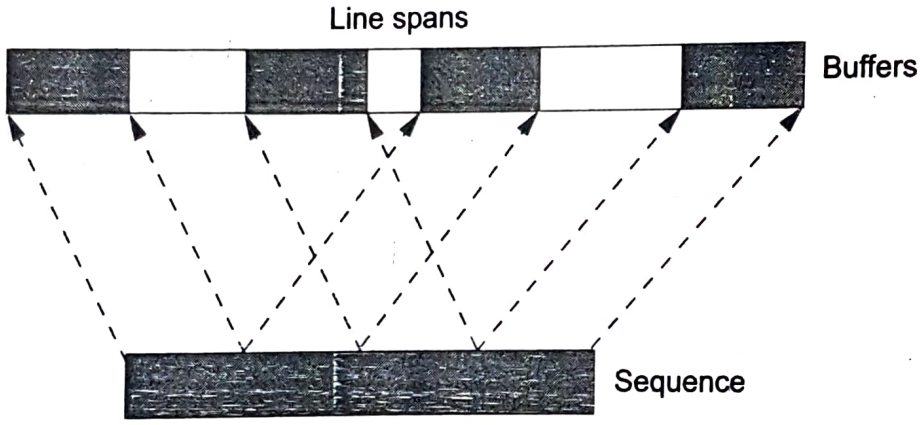

FIGURE 7.5 The line span method.

The line span method is more suitable for a line oriented text editor (for example, EDLIN) but is not common these days, since strict line orientation is seen as too restrictive. It does require preprocessing of the entire buffer before one can begin editing, since the line descriptors have to be set up.

Fixed size buffers. In the line span 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. 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. The scheme has been shown in Fig. 7.6.

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 upon the virtual memory system or the file system for efficient use of the disk. Also, 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

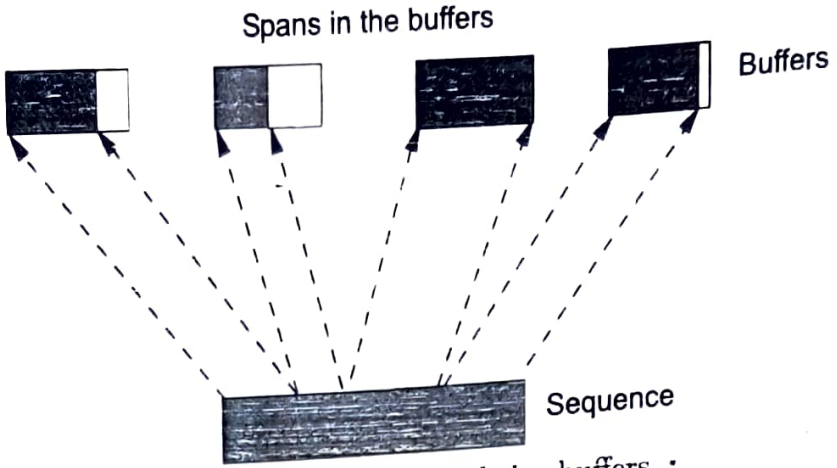

FIGURE 7.6 Fixed size buffers.

and occasionally merging two buffers to prevent the accumulation of large number of buffers. For example, assuming disk blocks to be 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 affect only 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.

The piece table method. In this 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. Each span is called a *piece* of the sequence, its descriptor is called a *piece descriptor*. The sequence of piece descriptors is called the *piece table*. The piece table uses the following two buffers.

1. *File buffer* contains the original content of the file being edited. It is of fixed size and is read-only.
2. *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). Figure 7.7 shows a piece table structure. 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. On the other hand, 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 insertion at the beginning or end of a piece. If several

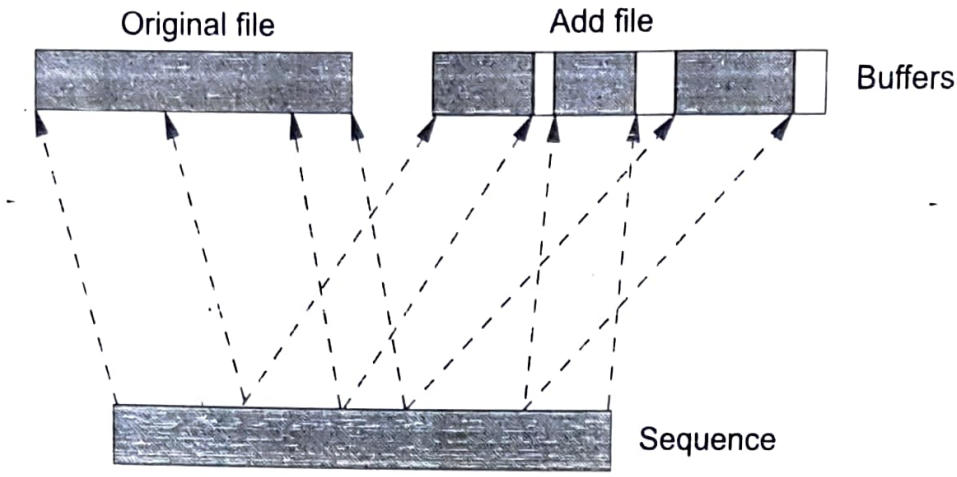

FIGURE 7.7 The piece table method.

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. Figure 7.8 shows the piece table after a file has been read initially. This is a very short file containing 20 characters. Figure 7.9 shows the piece table after the word "large" has been deleted. Figure 7.10 shows the piece table after the word "English" has been added. It may be noted 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.

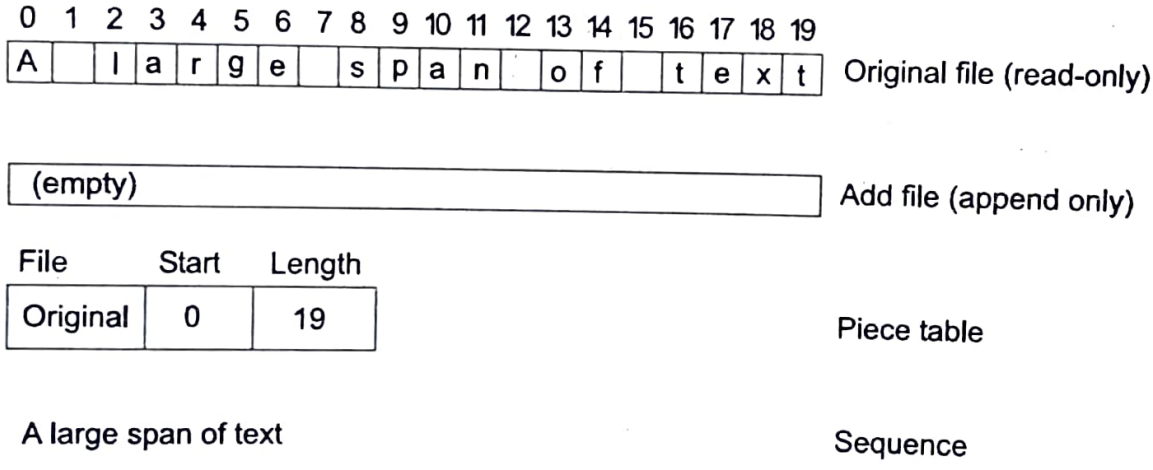

FIGURE 7.8 A piece table before any edits.

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 undos can be easily supported.

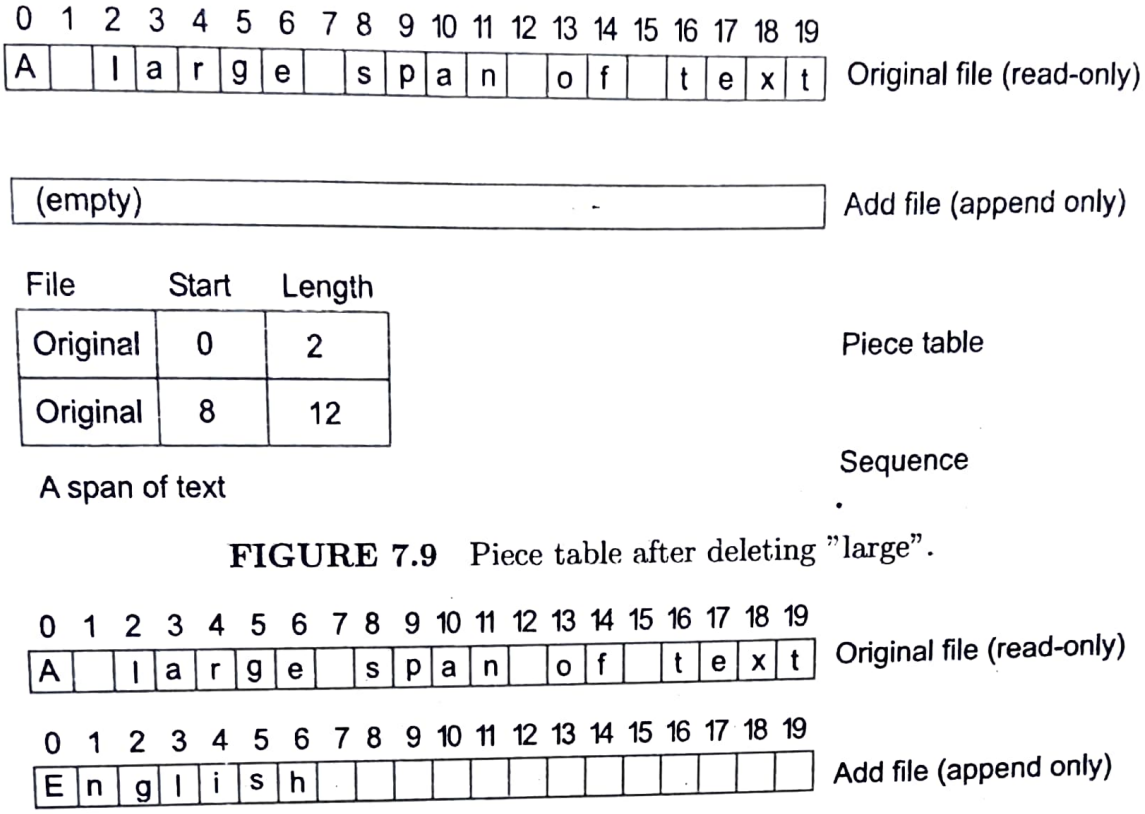

FIGURE 7.9 Piece table after deleting "large".

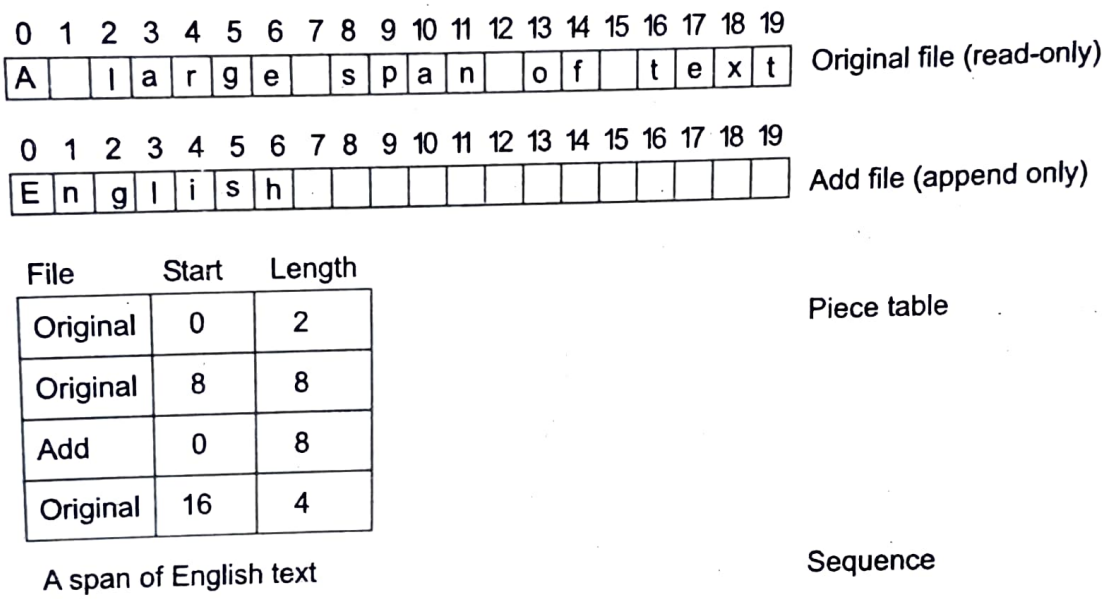

FIGURE 7.10 Piece table after deleting "large" and inserting "English".

- No file preprocessing is needed. 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.
- Text editor is free to split pieces to suit its purposes. For example, a word processor needs to keep formatting information as well as the text sequence itself. The 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.

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.

7.3 TEXTDOCUMENT DESIGN

TextDocument has the responsibility of loading the file into appropriate sequence data structure and provide text to the edit-control when asked for. Many operations during editing need to access text in a line-by-line manner. Often, the user wants to specify the number of the line (or range of lines) to be displayed and/or edited next. A line of text is a sequence of characters within a file with a well-defined, end-of-line marker. There are three main conventions for delimiting lines of text—under DOS and Windows, a *carriage-return/line-feed* pair is used, under UNIX and Linux, a single *line-feed* character is used, and under Macintosh operating system, a single *carriage-return* is used.

A *line-buffer* data structure is commonly used to locate individual lines. It is just an array of integer offsets that specify where each line of text in the document starts. The array is filled up by processing the text—searching for *carriage-return/line-feed* sequences and recording the offset for each line as shown in Fig. 7.11.

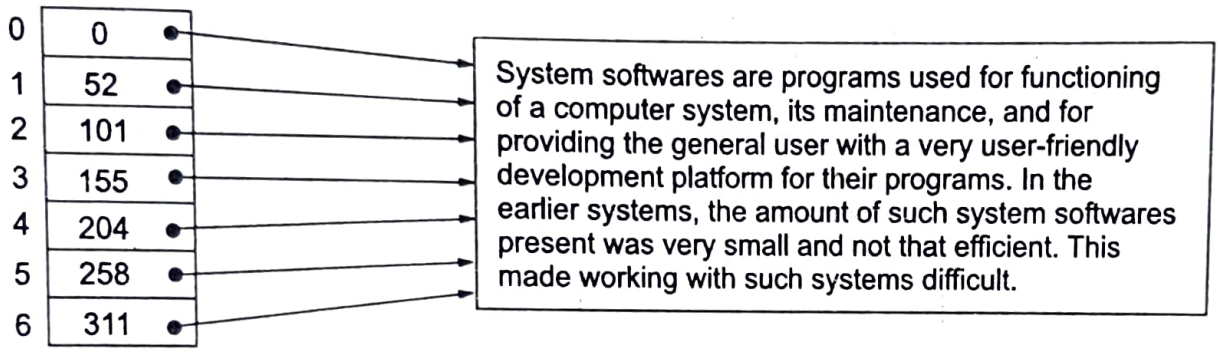

FIGURE 7.11 Line buffer and text.

7.4 TEXTVIEW DESIGN

TextView controls the way a document is displayed for editing to the user. The first major thing to do here is to draw the lines of text. Design of this drawing part is very much dependent on the underlying operating system. However, it basically consists of the following steps:

1. Create a basic drawing framework with a rectangular update area for displaying text.
2. Determine the *first* and *last* line of text that can be drawn in the update area by dividing the *top* and *bottom* co-ordinate (available as pixel) of the update window by the font height.
3. Now, the update area can be considered to be able to hold $(last - first + 1)$ lines of text, each line being drawn in a window of height equal to the font height. A basic drawing utility can thus be used to draw a line into its corresponding window.

It may be noted that the actual line of text that we want to draw is retrieved from *TextDocument* using some method to get a line of text. Thus, it does not matter if we change how to store the text file inside *TextDocument* as long as the interface to the *TextView* is not modified.

7.4.1 Scrolling

The *TextView* designed so far cannot handle *scrolling*. However, the scrollbars and the associated scrolling of text being displayed can be introduced easily into this design. It may be noted that all window systems come up with some technique to obtain the current position of the vertical scrollbar (say *m_nVScrollPos*) and of the horizontal scrollbar (say *m_nHScrollPos*). Moving the vertical scrollbar modifies the lines to be drawn. Thus, the *first* and *last* lines to be drawn can be computed as,

$$\begin{aligned} first &= m\_nVScrollPos + (top/FontHeight) \\ last &= m\_nVScrollPos + (bottom/FontHeight) \end{aligned}$$

where *top* and *bottom* are the window extents in pixels and *FontHeight* is also specified as number of vertical pixels needed to display a single character. Here, effectively we change the line index that we draw. Thus, as the scrollbar moves down, the lines of text effectively move up the display. We still draw lines at the same physical location within the window, but we draw *different* lines to give the illusion that we are scrolling through the document.

For horizontal scrolling, we consider the case of drawing a single line in its display window discussed earlier. We modify the *left, right, top,* and *bottom* extents of the window as follows:

$$\begin{aligned} left &= -m\_nHScrollPos * FontWidth \\ right &= right \\ top &= (nLineNo - m\_nVScrollPos) * FontHeight \\ bottom &= top + FontHeight \end{aligned}$$

When the horizontal scrollbar's position is increased (that is, we scroll to the right), we would expect a page to scroll to the left. So, as the scroll position increases, the text position must decrease. Hence "*m_nHScrollPos*" has been used. This logical text position is then multiplied by the current font width to produce a pixel based co-ordinate that can be used for drawing. The right-most edge of the line of text must still be fixed to the right edge of the window, so this value is left unchanged. As we scroll to the right, the length of the text that we draw gets larger and larger because the starting x-co-ordinate becomes more and more negative. The display device-context will clip the negative part of the window to give the illusion of horizontal scrolling.

Another approach for horizontal scrolling can be to find out the correct place to draw within the line of buffered characters and do all the drawing from a fixed x-co-ordiante of zero.

As the scrollbars are moved, the *TextView* receives different types of messages to be handled. The messages correspond to various events occurring due to the user asking to scroll the text. The procedure to handle such messages for horizontal and vertical scrollbars is almost identical. Hence, we concentrate our discussion only on the vertical scrollbars.

- *TOP:* The window is scrolled to position zero and redrawn.

$$\begin{aligned} m\_nVScrollPos &= 0 \\ RefreshWindow() & \end{aligned}$$

- *BOTTOM:* The window is scrolled to the maximum position and redrawn.

$$\begin{aligned} m\_nVScrollPos &= m\_nVScrollMax \\ RefreshWindow() & \end{aligned}$$

- *THUMB:* Here the user requests the window to be updated as the scrollbar is made to move up or down through some mechanism. The current scroll position is obtained via some tracking routine provided which is operating system specific.

$$\begin{aligned} m\_nVScrollPos &= TrackScrollBar() \\ RefreshWindow() & \end{aligned}$$

- *LINEUP:* Scroll position is decremented by one.

$$\begin{aligned} m\_nVScrollPos-- \\ RefreshWindow() \end{aligned}$$

- *LINEDOWN:* Scroll position is incremented by one.

$$\begin{aligned} m\_nVScrollPos++ \\ RefreshWindow() \end{aligned}$$

- *PAGEUP:* Scroll up by entire page. Assuming that the variable *m_nWindowLines* holds the number of lines in current window, scroll position is updated by this amount.

$$\begin{aligned} m\_nVScrollPos &= m\_nVScrollPos - m\_nWindowLines \\ RefreshWindow() & \end{aligned}$$

- *PAGEDOWN:* Scroll down by entire page. Assuming that the variable *m_nWindowLines* holds the number of lines in current window, scroll position is increased by this amount.

$$\begin{aligned} m\_nVScrollPos &= m\_nVScrollPos + m\_nWindowLines \\ RefreshWindow() & \end{aligned}$$

7.4.2 Multicoloured Text

The text may need to be displayed in different colours. This is needed for *syntax colouring* and more often for *selection highlighting*—to distinguish between the selected and unselected text. Since colour and style information is not stored in the text file, character colours must be computed separately and applied to the displayed text at runtime.

7.4.3 Multifont Display

There are some added complications that arise from the use of multiple fonts.

1. Fonts can vary in width and height. The width does not create much of trouble; however, fonts of different heights must be aligned so that their bases are on the same horizontal line.
2. The height of a line of text is not based on a single font any more—it must be big enough to hold the tallest font currently in use.
3. Smaller fonts do not fill the vertical extents of the line when painted. This will require us to fill this extra *dead space* ourselves.

7.5 CONCLUSION

In this chapter, we have seen different types of editors and their design techniques. The data structures needed for the manipulation of text have been discussed. The *piece table* method seems to be a good structure for this purpose. The design of an editor is very much dependent upon the facilities provided by the underlying operating system. We have seen different utilities needed (particularly for text window manipulation) to implement various features in an editor. In the next chapter, we will discuss another program development aid—the debugger.

EXERCISES

7.1 Distinguish between the line editor and screen editor. Give examples of both categories.

7.2 Study the editors available in your computer system. Compare between them in terms of relative advantages and disadvantages.

7.3 What are the components of a text editor? Explain the role of individual components.

7.4 Describe the commonly available edit control features.

7.5 What are the operations performed on text sequences in the process of editing? Which of these needs more care to design an efficient editor?

7.6 Compare and contrast between the data structures available for the text sequences.

7.7 Show the piece tables after each of the following sequence of editing over the original string: "The distinction between the two categories is made based upon the smallest unit that can be edited".

(a) Delete the word "distinction".

(b) Insert the word "wonderful" between "two" and "categories".

(c) Delete the word "smallest".

(d) Insert the word "largest" at the place where "smallest" was stored earlier.

(e) Change word "edited" to "deleted".

7.8 Explain how vertical and horizontal scrolling are implemented in the display during editing.