**Piece Table**

**Introduction:**

The piece table is the unsung hero data-structure that is responsible for much of the functionality and performance characteristics we’ve come to expect from a text editor. Visual Studio Code has one. Microsoft Word 2.0 had one back in 1984.

Typically the text of the original document is held in one immutable block, and the text of each subsequent insert is stored in new immutable blocks. Because even deleted text is still included in the piece table, this makes multi-level or unlimited undo easier to implement with a piece table than with alternative data structures such as a gap buffer.

This data structure was invented by J Strother Moore.

**Construction:**

For this description, we use a buffer as the immutable block to hold the contents.

A piece table consists of three columns:

* Which buffer
* Start index in the buffer
* Length in the buffer

In addition to the table, two buffers are used to handle edits:

* "*Original buffer*": A buffer to the original text document. This buffer is read-only.
* "*Add buffer*": A buffer to a temporary file. This buffer is append-only.

## **Piece descriptors**

In order to know where the user inserted the text in the file, the piece table needs to track which sections of the file come from the original buffer, and which sections come from the add buffer. It does this by iterating through a list of *piece descriptors*. A piece descriptor contains three bits of information:

* source: tells us which buffer to read from.
* start: tells us which index in that buffer to start reading from.
* length: tells us how many characters to read from that buffer.

When we first open a file in our editor, only the original buffer has content, and there’s a single piece descriptor which tells our editor to read entirely from the original buffer. The add buffer is empty because we haven’t yet added any text to our file.

{

"original": "the quick brown fox\njumped over the lazy dog",

"add": "",

"pieces": [Piece(start=0, length=44, source="original")],

}

## **Adding text to a file**

Now let’s add the same text to the middle of our file just like before. Here’s how the piece table is updated to reflect this:

{

"original": "the quick brown fox\njumped over the lazy dog",

"add": "went to the park and\n",

"pieces": [

Piece(start=0, length=20, source="original"),

Piece(start=0, length=21, source="add"),

Piece(start=20, length=24, source="original"),

],

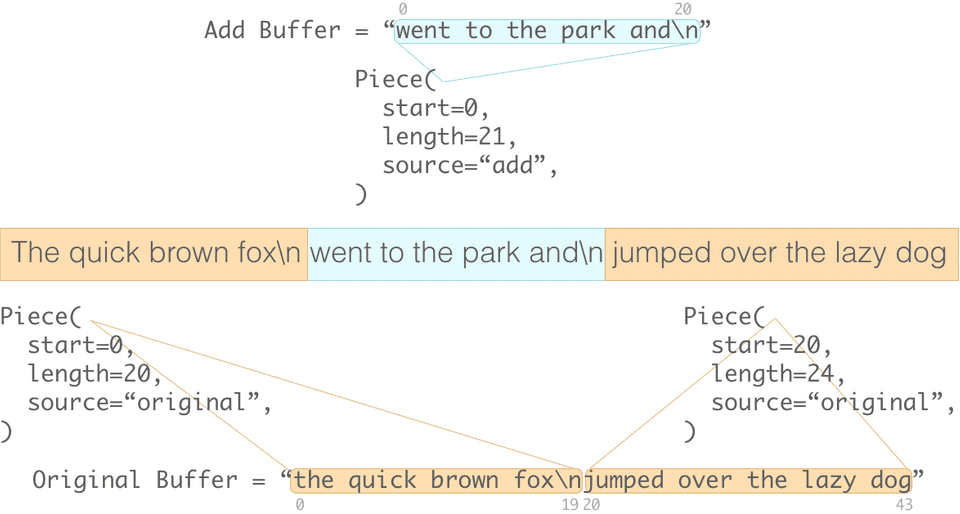
}

The text we inserted in our file using our text editor has been appended to the add buffer, and what was previously a single piece is now three.

* The first piece in the list now tells us that only the first 20 characters of the original buffer (the quick brown fox\n) make up the first span of text in our file (note that \n, representing a line break, is a single character).
* The second piece tells us that the next 21 characters of the file can be found between indices 0 and 21 in the add buffer (went to the park and\n).
* The third and final piece tells us that the final span of text in the file can be found between indices 20 and 44 (start + length) of the original buffer (jumped over the lazy dog).

The act of inserting text into a file typically results in splitting a piece into three separate pieces:

1. One piece to point to the text that falls to the left of the newly inserted text.
2. Another piece to refer to the inserted text itself (in the add buffer).
3. The third piece refers to the text that got pushed to the right of the newly inserted text.



Things are a little different when you insert text at the beginning or end of an existing piece. In this case, adding text doesn’t “split” an existing piece in half, so we only need a single extra piece to represent the newly inserted text.

### **Insert**

### Inserting characters to the text consists of:

* Appending characters to the "add file" buffer, and
* Updating the entry in piece table (breaking an entry into two or three)

### **Delete**

### Deletion involves only modifying the appropriate entry in the piece table.

## 

## **Time complexity**

## Although a piece table itself is a data structure, it does not state how it should be implemented. The time complexity of operations depends on how the table is being implemented. One possible implementation of a piece table is in a splay tree, because it provides quick access to recently-accessed elements.

## **Usage**

## Several text editors use an in-RAM piece table internally, including Bravo, Abiword, Atom and Visual Studio Code.The "fast save" feature in some versions of Microsoft Word uses a piece table for the on-disk file format.

## The on-disk representation of text files in the Oberon System uses a **piece chain** technique that allows pieces of one document to point to text stored in some other document, similar to transclusion.

## **Conclusion**

## There are several ways we could improve the piece table described above. They’re often combined with other data structures such as trees in order to improve aspects of their performance.

Ref:

<https://en.wikipedia.org/wiki/Piece_table>

<https://darrenburns.net/posts/piece-table/>