The Piece Table

January 23, 2019
datastructures | beginners | tutorial

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.

Despite being an almost ubiquitous feature of modern text editors, it remains relatively poorly documented. In this article, I'll explain how a piece table is structured, why they're used, and how your text editor interacts with it as you edit files.

My aim is to make this post as beginner-friendly as possible, so we'll explore each concept slowly. You should, however, have a basic understanding of arrays, strings, and objects (or dicts/structs) before continuing.

When you open a file in your text editor, the contents are read from disk and into a data structure in memory. If you were to write a text editor, how would you store an open file in memory?

First instincts - the array of strings

Our first instinct might be to use an array of strings, where each string is a single line in the file. In other words, we'd represent this file...

```
the quick brown fox

jumped over the lazy dog
```

...like this:

```
lines = [
    "the quick brown fox",
    "jumped over the lazy dog",
]
```

This is an intuitive way to represent a text file in memory. It lets us think about the file in a way that is similar to how it appears on-screen in our text editor.

It's also a perfectly acceptable approach, and you could argue that the intuitiveness of it outweighs any potential flaws it may have. In fact, Visual Studio Code used a similar model until early 2018.

Unfortunately, this method becomes costly when we're dealing with large files. To see why, think about what would happen if someone were to insert a new line (`"went to the park and"`) into the middle of the file:

```
the quick brown fox
went to the park and
jumped over the lazy dog
```

In order to make room for this new line, all of the lines below it in the array need to be shifted along in memory. For large files, this quickly becomes expensive. As the file grows larger, there's more data to be moved.

This is just one of the downfalls of this method. In this fantastic blog post by the Visual Studio Code team, they highlight other flaws with this approach such as excessive memory usage and performance issues relating to splitting up the file into multiple strings on newline characters.

An append-only representation

If we append something to the end of an array, we don't have to shift any data along to make space for it, so we don't incur the same performance penalty as in the case where we insert into the middle of an array (more technically, appending to an array has O(1) time complexity, but insertion is O(n)).

The *piece table* is the powerful data-structure behind many text editors, old and new. A key characteristic of the piece table is that it records all of the insertions we make to a file in an append-only manner.

Let's explore how the piece table works.

When we read a file from disk into a piece table, the text inside the file is stored in a single string which we never modify. We call this string the *original buffer*.

```
piece_table = {
    "original": "the quick brown fox\njumped over the lazy dog",
    ...
}
```

When we add text to the file in our editor, that text is appended to the *add buffer* of the piece table, which is initially just an empty string.

Adding a copyright notice at the top of the file? Append it to the add buffer. A new function in the middle of the file? Add buffer. An extra newline at the end of the file? Add buffer!

The add buffer is the second of the two buffers in which make up a piece table, and it is append-only.

```
{
    "original": "the quick brown fox\njumped over the lazy dog",
    "add": "",
  ...
}
```

By appending everything that gets inserted into the file onto the add buffer, we record the text that the user has typed into the editor, whilst avoiding those pesky mid-array insertions we mentioned earlier.

Let's open our hypothetical text editor again, and add a line into the middle of our file as we did before. That makes our piece table look like this:

```
{
    "original": "the quick brown fox\njumped over the lazy dog",
    "add": "went to the park and\n",
  ...
}
```

The text we added to the middle of the file is in the add buffer. The original buffer has stayed the same (and always will).

These two strings, `original` and `add`, hold within them the entire contents of the file that is open in your editor, as well as everything that has ever been contained in the file since it was opened.

When the editor displays an open file, it combines different sections of these two strings together to form the what you see on your screen. Some

sections of these strings may be ignored if they're no longer present in the file (for example if the user of the text editor deletes some text).

In the text below, the middle section comes from the add buffer since it was inserted, and the rest of the text comes from the original buffer since it was part of the original file.

Right now, our hypothetical editor knows the user has inserted the string `"went to the park and\n"` into the file, but not *where* they inserted it. We haven't yet got enough information in the piece table for the editor to be able to correctly display the contents of the file. The missing piece of the puzzle (pun intended) is to track where in the document the user inserted the text.

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.

Saving and displaying the open file

As mentioned at the very beginning of this post, when we open a file in a text editor, it gets read from disk and stored inside a data structure in memory (very likely a piece table or some variant). Likewise, when we do the inverse operation and save the file, our editor needs to be able to read the piece table and write the contents back to the file on disk.

By reading the piece descriptors in consecutive order, our text editor can transform its internal representation of the file in the piece table into what you see on your screen, and what gets written to the file when you save it.

```
for piece in piece_table["pieces"]:
```

```
source: str = piece.source

buffer: str = piece_table[source]

span_of_text: str = buffer[piece.start:piece.start+piece.len

print(span_of_text)
```

Note: In Python, the `string[start:end]` syntax can be used to return a substring from a string. For example, `"hello"[1:3]` returns `"el"`.

Deleting text

When we delete some text from a file, we split an existing piece into two pieces:

1. One piece points to the text to the left of the deleted text.
2. A second piece points to the text to the right of the deleted text.

The deleted text will still be present in one of our buffers, but with no pieces referring to it, it is not considered part of our file.

You can think of piece descriptors like spotlights pointed at a wall with all of the text ever written to a file painted on it. At any point in time, these lights illuminate sections of the wall, revealing the text there. The illuminated text is considered as part of the file. Although we know there's more written on the wall, it's hidden in the darkness and considered irrelevant.

Undoing and redoing

One benefit of keeping all of that text around even when it's not currently part of the file is that it makes the implementation of undo/redo considerably easier.

Text that is currently "in darkness" may be required again in the future. Wouldn't it great if we could just adjust our lighting to illuminate that text

once again, instead of having to repaint the wall?

In the array of strings approach discussed at the start of this post, an undo operation could result in millions of characters of text being inserted or deleted across millions of different strings. With the piece table method, this cost of this action becomes trivial. The text to be added or removed as part of an undo or redo is already there in one of the buffers, and we just need to update our piece descriptors to point to it once again.

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.

That said, my goal for this post was to leave you with intuition and some appreciation for how your text editor works internally, and those tweaks fall out with that goal.

If you have any constructive feedback on this post, or if you've noticed any errors in it, let me know!

If you're interested in more content like this, follow me on Twitter, DEV, and check out my blog!

References & further reading

- Data Structures For Text Sequences (PDF) - Charles Crowley
- What's been wrought using the Piece Table? - David Lu (I think)
- Text Buffer Reimplementation, a Visual Studio Code Story - Peng Lyu