Sanedit Implementation Notes

This document contains implementation details and notes. It should provide answers why certain design decisions were made.

Buffer

Text buffer implementation. It is a piece tree a piece table variant which stores the pieces in a red-black tree instead of a vector. To understand what a piece tree is lets start with a piece table.

Piece table uses two text buffers, original buffer which contains the original text and an add buffer which contains added text. The original buffer is read only and the add buffer is append only. This means that you can index into these buffers as the text never changed. Piece table stores pieces which are an offset + length pair. The pieces then reference the buffers and form the buffer contents. This means the buffer content is not contiguous in memory. For more information on piece tables

Then a piece tree is a piece table that stores the pieces in a red-black tree. The nodes contain left subtree length (in bytes) so we can search for a bytes offset in $O(\log(n))$ where n is the number of pieces in the tree. Rust provides easy access to cow functionality using `Rc::make_mut` which is used in the tree. It allows us to take lightweight snapshots of the tree, which can then be restored. This is a built in solution for undoing changes. The snapshots are lightweight as the trees can share nodes, new nodes are only created if snapshots referencing the node still exist.

For more information on piece trees see Improving the AbiWord's Piece Table,
VS Code Text Buffer Reimplementation

This implementation operates on bytes and does not guarantee that the content is valid UTF-8. The bytes are decoded to UTF-8 when needed. Line counts could also be cached to nodes (like left subtree byte length) to provide a fast line search, but this implementation opts of that so we do not need to read the whole file on load. This means opening large files should be as fast as opening small ones.

Piece tree is good at

- Random insertion and deletion
- Large files
- Easy undo implementation
- Easy text position tracking by indexing into backing buffers

Piece tree is bad at

- Sequential insert and delete are not $O(1)$ amortized time.
- Performance deteriorates over time as more pieces are in the tree
- Complex implementation because of red-black trees

Marks

Marks track a text position through edits. Piece tree makes marks easy to implement, because the add buffer or original buffer never change we can point to them using offsets ie. `(BufferType::Add, 500)`.

To create a mark we find the piece responsible for that position of the buffer and get the buffer type and appropriate offset into the buffer the piece references. Now the mark can point to its new position as long as the piece that holds that offset is still in the tree even through edits.

To find the marks position after edits we can iterate over all the pieces and find out if the mark is included in them. If it is we can return the pieces position in buffer + appropriate offset the mark is pointing to.

If the piece is deleted the mark can not find its position again.

Multicursors and piece tree

Problem

Suppose we are inserting a character at m positions (cursors) when a character is inserted at each position, it is appended to the add buffer m times where m is the number of cursors. This is relatively bad in itself. But the real problem comes with the next character, now we add the character m times to the add buffer again, but notice, in single cursor case the characters would be sequential in the add buffer but in this case they are not. This means that the pieces referencing this part of the buffer cannot be appended to at all, we must create a new piece. So if we insert n characters at m points, it would produce $m \cdot n$ new pieces. This would deteriorate performance way too quickly.

Solution

Problem is solved by exposing a new function for multi insert. It takes the text and multiple positions to insert to. Now we can simply append the character to the add buffer and reference it multiple times. This however creates an unexpected problem, previously all of the pieces were unique in a sense that only one piece referenced to a slice of add buffer. This assumption was used in the mark implementation. To identify the pieces from one another we need to add a `count` field. The `count` field is the index of the multiple positions array we got at the function call. Now the pieces can again be uniquely identified and the mark implementation is happy.

Cli

Commandline interface to launch the editor.

Editor

Editor implementation itself.

Messages

Messages sent between editor and clients.

Regex

Regex implementation to search non-continuous text.

Terminal-client

Terminal client for the editor. Its job is to send keyboard and mouse events to the editor and draw the screen.

Ucd

Unicode character database, used to implement unicode properties, grapheme breaks etc. This is generated using the ucd-generate utility.