Text-Terminal: A UTF-8 Text Editor for the Linux Shell

University of Basel
Faculty of Science
Department of Mathematics and Computer Science

Heinrich Ahrend
Yorik Metzger
Jorn Riedel
Shura V. Ruben

June 11, 2025

University of Basel

Contents

1   Introduction

Text editors are an essential tool that is quite often taken for granted since pretty much every single operating system has one included by default. Despite this, quite a lot of software engineering is required to develop a software product that is able to fulfill the requirements posed by the tasks we need and wish to perform in our text editors. This is especially the case when features like word and line count should work well with very large files.

Since these aspects intrigued us and we wanted to try and develop our own solution from the ground up, we choose to develop a full text editor as our OS course project. More specifically, we have opted to try and develop a plain text (file) editor with good handling of large (even multiple gigabytes large) files, since this is one of the major weak points we identified with other text editors like Visual Studio Code. Additionally:

- It should support UTF-8 encoding since it is essential when writing a text in German and French.

- It should be compatible with the three most common line break standards: `\n` (Linux), `\r\n` (Windows), `\r` (Classic Mac OS).

- The user interface should be a bit easier to use than pure keyboard text editors (e.g. vi/vim). It should allow to use the mouse for the most important operations.

- Finally, the user interface should be responsive even when operating on large files.

2   Background

After researching possible approaches for storing the text, we settled on implementing our own *piece table* data structure inspired by C. Crowley's discussion of text editor data structures [1]. The general idea of *piece tables* is to store a sequence of *piece descriptors*, which point to contiguous text spans in a buffer. By using a separate *file buffer* for the file content and an append-only *add buffer* for new content, the complexity of editing text is essentially reduced to updating this sequence of piece descriptors. Thus, by using memory mapping for the *file buffer*, the size of the data structure only grows with the number of edits rather than with the file size. This makes *piece tables* an excellent choice for out text editor intended to handle large files.

For building the text-based user interface we use ncursesw version 6 [2], because it is a library that supports various terminals. We specifically chose the wide character version of ncurses to support Unicode and international character sets (standard ncurses only has ASCII support).

We also use xclip [3] for our copy and paste functionality, which allows accessing the clipboard of the X11 windowing system. Although X11 is increasingly being replaced by Wayland on Linux systems, we have found that xclip remains compatible with most Wayland environments thanks to XWayland compatibility.

3   Methodology

After deciding on our project topic we fixed our software requirements in more concrete form. We identified that it would be most beneficial to divide our future code into frontend and backend with the main interface being between graphical user interface (GUI) and text data structure. The `textStructure.h` header is the main interface connecting the GUI in `main.c` and the actual text data structure implementation in `textStructure.c` [4]. The other files are mostly built around these two central pieces of code. A more visual layout can be seen in figure 1.
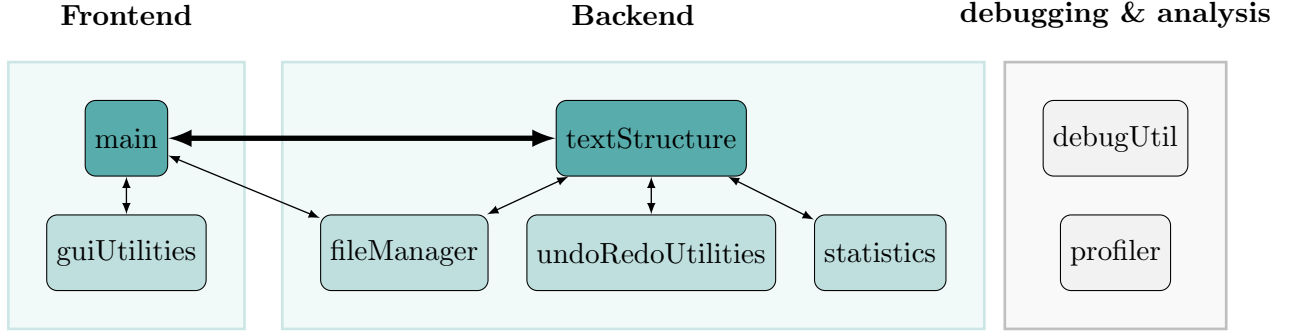
Figure 1: Project Code Structure

The piece table stores the piece descriptors in form of a doubly linked list (see figure 2). Each piece is identified by the buffer it is stored in, the offset into that buffer and its size. Editing the text sequence at a specific position, therefore requires the piece descriptor associated with that position by using the `getNodeForPosition` function. Afterwards, all changes can be done locally around the respective piece descriptor. By using empty sentinel nodes at the beginning and end of the piece table, we can simplify edge cases of changing the sequence at these positions.

The add buffer grows dynamically with the amount of content it stores. Since it is append-only, undo and redo can be easily implemented by storing the first and last piece descriptors associated with a changed span of text (see `undoRedoUtilities.c`) and the required pointers if and operation should be undone. Multiple operations can also be bundled together by pointing to the previous operation so that they are all undone at the same time (e.g. replace consisting of a deletion followed by an insertion).

When searching for a specific text, the main challenge is to keep track of the line numbers because the GUI should directly jump to the result if one is found. Since the sequence is searched linearly in most use cases, we use a cache that stores the position and line number of the last result. While searching, the amount of line breaks after the position where the search should begin is also tracked. By accessing the cached last result, this allows the line number to be determined efficiently.

In section 4, it can be seen that the described approach performs fairly well - even for large files and large piece tables.
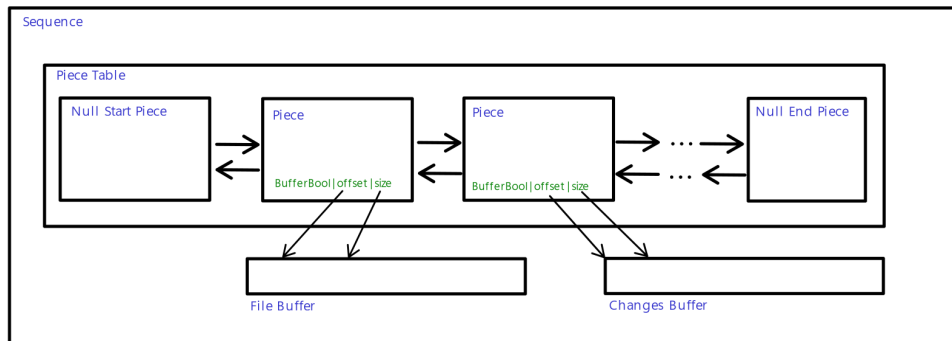

Figure 2: High level text data structure (sequence) illustration

File management(see `fileManager.c` [4]) The way we load files into the editor is optimized even for very large files (we tested up to 5.5 GB text files). This is partly because we directly map the file buffer of the sequence onto a mmap of the opened file. This directly leverages

the speed and advantage of the Linux mmap capabilities. Upon save we also use mmaps to write, but we decided to still take the slight overhead of copying the original file to a temporary location in order to keep the sequence's piece table in it's current state and there for also the whole undo history alive. Otherwise it would need to be reset each time a save is done, which we found not very user friendly.

On the front end side, `print_items_after` is one of our most important parts and our method to that displays text in our terminal. Prints a certain number of lines starting from a chosen atomic position in our text sequence. Checks if sequence exists and line break standards. If good proceeds. It walks through blocks of text data and handles things liek UTF-8 character boundaries, skips control characters and detects line breaks or end-of-block to finish a text line. Changes current line segment to wide-character string for terminal compatibility. Output is the processed string that goes to the terminal at correct screen position.
For efficiency it only prints lines that are actually visible on the terminal, so out of view lines don't get printed. For that we have a variable that has the absolute position of the line at the top of the screen (e.g. top line is actually the 5th line in the entire text). Also this is where the lineStats get update and if a line goes across multiple blocks the counters(atomicsInLine, nbrOfUtf8CharsNoControlCharsInLine) carry over to next block

In `guiUtilities` we have our line statistics like: what is the current line number at the top of the screen or how many chars are in the specified line. The methods in here are used for things like scrolling, jumping to a specific line when using the search function or managing the line stats. Characters are the converter from UTF-8 to wide characters and a way to translate cursor position to a position in our data structure.

Our Cursor refreshes independent of our text. That means when our cursor moves position it doesn't cause the text to also be refreshed, that would be a big performance hit.

4   Results

Overall we are very happy with the performance of our text structure. We ran different performance metrics with a 15 KB, a 20 MB, and a 1.9 GB test file. As is visible below, we managed to implement all real time features so that all important metrics stay under 1 or 2 milliseconds, which we deem very good performance, especially since it stays in this range even with text files that are gigabytes large.
In figure 3 we show different insertion patterns for the most significant cases in the 20 MB file. In this context, optimized insert signifies that contiguous inserts in the text sequence get merged into the same piece descriptor, allowing to reduce the size of the linked list in this common usage pattern (most often, text is written one char after another in our experience). We only included the metric of the 20 MB file since we didn't see a different behavior when comparing them across the three file sizes.

Figure 3: gui display delay in (20 MB file)

In figure 4a and 4b the average of these three insertion patterns (at each insert position) is used to display a slowdown analysis between the three file sizes mentioned above. What is to note here, is that the performance seems to be more limited by other system factors while our text structure maintains similar insertion performance across all file sizes (see average variations from one run to the other).
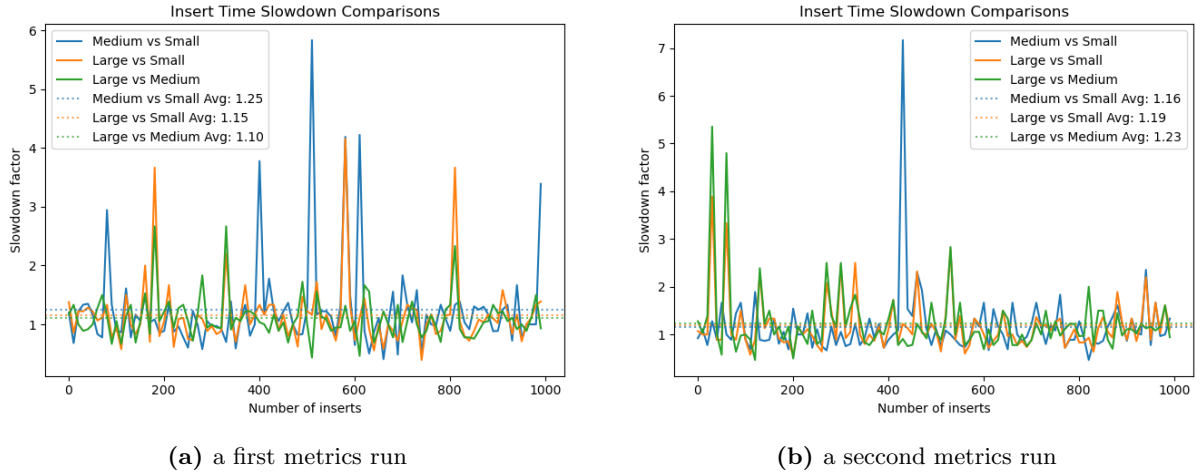

(a) a first metrics run

(b) a seccond metrics run

Figure 4

Figure 5 depicts the time taken to determine the relevant nodes for the deletion of text that spans a growing number of nodes (piece descriptors). Even for a large amount of nodes this still performs in the range of milliseconds. At the same time, we can see that undo and redo are not affected by the amount of nodes that the deletion effects, because they work locally around the first and last node of the deleted span (these nodes are directly accessible on the undo stack).

In figure 6 we can see the effect of caching the last line result when iteratively searching a text for the next occurrence of a periodically appearing pattern. Without caching, in each iteration the line number has to be determined starting from the beginning, which takes longer the bigger the position of the search result is. By using the last result in the cache, this behavior is prevented.
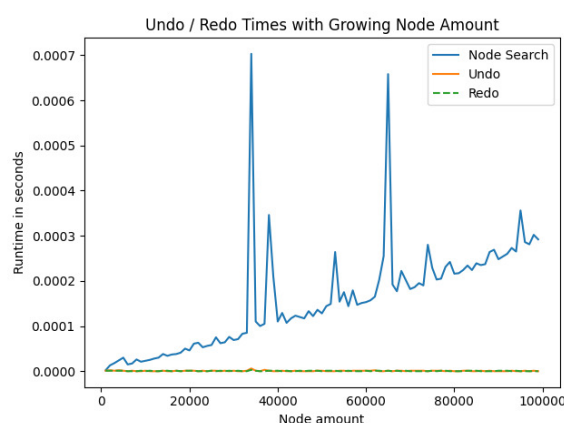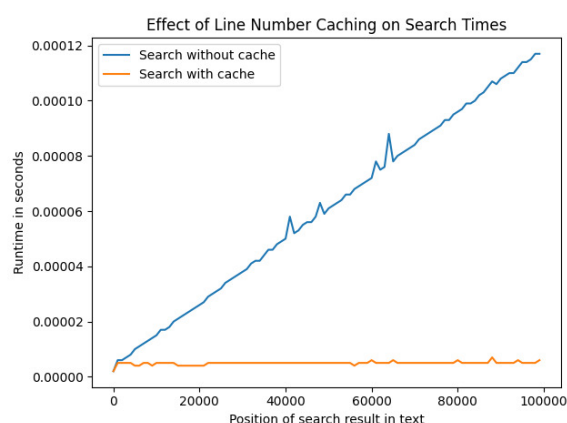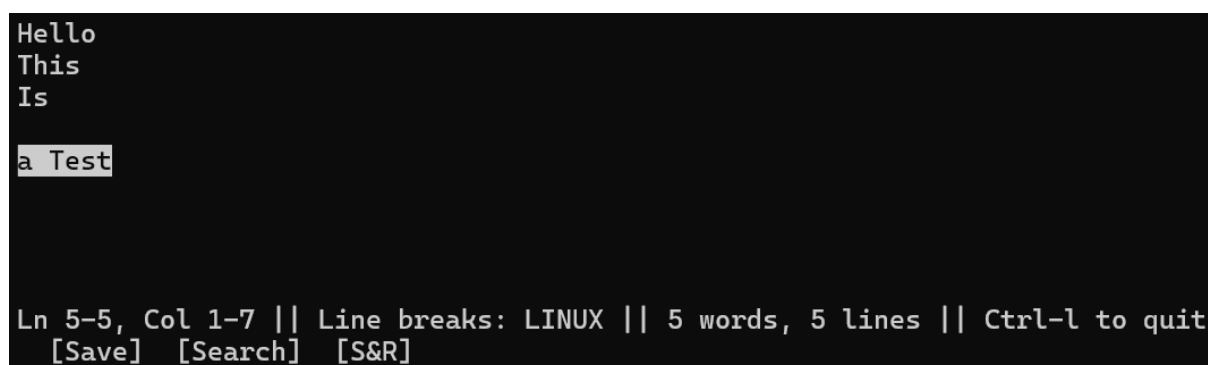
Figure 5

Figure 6

Figure 7: GUI of our text editor running on WSL

Above you can see our GUI. From the top left you can see 5 lines written, one of the blank.
'a Test' is marked and can be use to copy/paste or delete this section. Ln 5-5 means marked are rows from 5 to 5, same for Col but column 1 to 7. 'Line breaks: LINUX' means that the current line break style used is LINUX (\n).
Next to it you also have total line and total word count and to the right is the short-cut to exit the editor. At the very bottom are buttons that you can press to use them. S&R means search and replace.
Many aspetcs of our GUI will be shown in this video [5], as some of these things are impractical to show on a picture.

5 Discussion

We have most normal features a text editor should have like a cursor, mouse integration, deleting, line breaks etc. We even have some advanced features like search/replace, copy/past and can even open multi GB files, so in terms of actual features we are pretty satisfied with the outcome. But there are also some problems that could use some work, for example emojis are displayed as more than one character even though they are only one UTF-8 character, so they are cumbersom to use in the editor and can cause bugs.
Horizontal scrolling is also not working in the final build, it used to work at some point but stopped working somwhere down the line. Now when you reach the right end of the screen it just stops.

6 Conclusion

Over all we have managed to build a really good editor backend that leverages our customized and well adapted data structure concepts. We are achieved all the goals we have set ourselves a the beginning of the project and full filled more software requirements then initially planed A.1. Whilst the front end is functionally good, the text structure back end enables to full fill all the points we listed in the introduction even with very good performance.

7 Lessons Learned

For the GUI aspect we learned that cursor and internal position can very easily de-synchronized, it happened way too often and not even because of wide character support.
The other big part for the GUI group was also time management, we were way too slow at the beginning and middle of the project and definitely caused the other group some headaches. Next time we should definitely try to put a lot more work into the beginning.

Text structure wise we learn very much in this project, especially what considerations go into the design of a text editor and what should be given special attention when handling UTF-8 encoded text. Last but not least we took great pleasure in discovering the low level optimization capabilities provided by C and the satisfaction of having code run accelerated and fast (compared to java).

References

[1] C. Crowley, "Data structures for text sequences," *Computer Science Department, University of New Mexico*, pp. 1–29, 1998.

[2] Thomas E. Dickey, "ncurses." [Online]. Available: https://invisible-island.net/ncurses/. Accessed 10 Jun. 2025.

[3] astrand & Contributors, "xclip: Command line interface to the X11 clipboard." [Online]. Available: https://github.com/astrand/xclip. Accessed 10 Jun. 2025.

[4] Heinrich Ahrend,Yorik Metzger, Jorn Riedel, Shura V. Ruben, "GithUb repository of the project." [Online]. Available: https://github.com/sruben1/Text-Terminal. Accessed 10 Jun. 2025.

[5] Jorn Riedel, "Demo of Text Editor Project." [Online]. Available: https://drive.google.com/file/d/1obnfqQgP9Gp3SvdBcAY_Rjes19-MmlR8/view?usp=drive_link. Accessed 10 Jun. 2025.

Appendix A: Materials

A.1 Software Requirements

Requrements we set ourselves at the initial project submission:

- custom data structure

- possibility to open files

- utf8 support

- line break styles

- move cursor

- clickable buttons

- selectable text

- edit text

- special functions: find, replace

- copy ,paste

- handle different line break standards

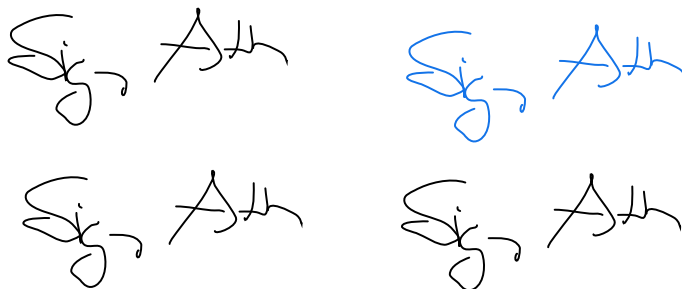- word count statistic

- line break type statistic

Additional requirements we managed to do:

- cursor position statistic

- line statistic

- replace all occurences

Appendix B: Declaration of Independent Authorship

I attest with my signature that I have completed this paper independently and without any assistance from third parties and that the information concerning the sources used in this paper is true and complete in every respect. All sources that have been quoted or paraphrased have been referenced accordingly. Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or to expulsion from the course of studies. I have taken note of the fact that in the event of a justified suspicion of the unauthorized or undisclosed use of AI in written performance assessments, I am upon request obligated to cooperate in confirming or ruling out the suspicion, for example by attending an interview.

Signature of all authors as PDF