README

How to run my code?

- I used java to write the code so make sure Java8 is available on the machine.
- Use a Linux/Unix system. Otherwise, you need to modify the path to the dictionary at line_19 of the “SimpleTextEditor.java” file and line_34 of the “GapBufferTextEditor.java” file.
- go into the “src” folder in the terminal.
- input command: “javac Test.java”
- input command: “java Test”
- Then, you can see all the test result.

How did I implement the Text Editor?

I used the “gap buffer” idea to implement the text editor. Gap Buffer is basically an array and text is stored in this large buffer in two contiguous segments (one at each end), with a gap between them for inserting new text. The gap is always starting at the position where the cursor is located. In my implementation, the gap is tracked by 2 pointers, which point to the beginning and ending index of the gap in the array. The picture below is an example.

Gap Buffer: [left_half_of_text] [empty buffer for insertion] [right_half_of_text]
| |
leftPointer rightPointer

Every time we want to do some operation at a specific location, we need to move the cursor. We can achieve that by transferring characters from left to right.

The paste and cut operation is quite obvious. We first need to move the cursor to the desired position. Then we can insert the text directly into the empty gap or modify the pointers to delete text. Also, copy is just the operation of finding a substring of the text and place it in a variable. To view the whole document, we need to construct it from scratch by concatenating those two sections of text at both ends, which is time-consuming. I improve this by maintaining a cache of the document and a boolean variable indicating whether the document has been modified or not. So before constructing the document from the very beginning, we can check the boolean variable. And if it indicates the cache is valid, we can return the cache directly.

The operation for checking the number of misspelled words is made easy by the gap buffer data structure too. In my implementation, I did a thorough check when the text editor loads in the document at the beginning. And every time we modify the text, we just keep an eye on the changes around those places and update the number of misspelled words accordingly. This

implementation is quite efficient compared to the brute force one because we don't need to scan the whole document every time.

One last thing about my gap buffer implementation is resizing. When the gap is too small for the income text, we need to resize the buffer so that the new document can fit in. I set the size of the new array to twice the size of that new document. The number is just what first comes cross my mind. And to make the performance better, more evaluation and research are needed.

Why I choose gap buffer?

I did some research on how to implement the text editor efficiently. There are 3 dominant data structures available: gap buffer, rope, and piece table.

Gap buffer is efficient on the assumption that insertion and deletion operations clustered near the same location, which is often the case when humans write documents. It has amortized O(length_of_input_text) time complexity for inserting and amortized O(1) for deleting text in the desired cases because we barely need to move the cursor. The inefficiency kicks in when we need to resize the array or to move the cursor across a large part of the content. As we mentioned above, the time complexity for checking misspelled words is good because we don't need to go through the whole text. The space complexity for gap buffer is quite small because we are just using an array and only use one piece of extra space for that empty buffer in the middle. Also, by choosing the gap buffer, we can take advantage of spatial locality of the data. Relevant data are all store near each other and we need less paging from the operating system.

Rope is a tree data structure. It offers O(log N) time complexity on insertion, deletion, and traverse. Although it guarantees the worst cost to be O(log N) if the tree is balanced, we need to make a lot of effort to make sure the tree is balanced. Some extra space is needed to store the information of the tree node. Also, when we create and destroy tree nodes, some penalty will exert on the time complexity and we need to do these operations often. The tree data struct will lose some data spatial locality compared to the array data structure. Finally, I cannot think of a good way to check the number of misspelled words other than the brute force way.

Piece table seems a good candidate to implement a text editor. It uses splay tree data structure and have good time complexity. I also make use of the assumption that recently accessed data is the most likely to be required soon in the future. As I did more research on it, I find some basic operation is quite complicated that I cannot implement it in such a short time. So finally, I chose the gap buffer idea because I'm familiar with array enough and it has many good properties.

How do I evaluate the implementation?

First, I check the correctness of the gap buffer implementation by comparing it to the brute force implementation.

Then, I compare the speed of each operation in the implementations by doing each operation (or some operations combined) several times.

At last, to emulate the real use scenario, I make the text editor in both implementations to do a specific number of random operations. I separated the text cases in 3 aspects. the first one is all 5 operations have the same possibility to be chosen. The second one is intended for emulate cases where we do heavy modifications on a document so, the operations like copy, cut, paste are more likely to be chosen than the read and check operations. The third one is opposite to the second one. The read and check operation are more likely to be chosen than copy, cut, and paste operation.

An example of the result run by my program can be found at file “result.txt”.

What extension I added and what I would do if I have more time?

- I add some sanity checks for the input.
- The way I check the misspelled words is quite brute force: I split the document into a string array by the space character. Then I iterate through the string array and check whether the dictionary(a hashmap constructed when the document is loaded into text editor at the beginning) contains that word. The result of this method doesn’t quite reflect the real situation because a sentence “Hello, World!” would result in 2 misspelled words in that “Hello,” and “World!” don’t exist in the dictionary even though “hello” and “world” did. If I have more time, I will fix this.
- I would like to add undo and redo operations in the future. These two operations are quite simple and efficient for gap buffer data structure because we are operating around the current cursor.
- I also want word correction functionality to be included in my text editor. I’ll do some research on that in the future.
- The code for evaluation is quite ugly and I wrote a lot of repeated code. I would like to structure them I had more time