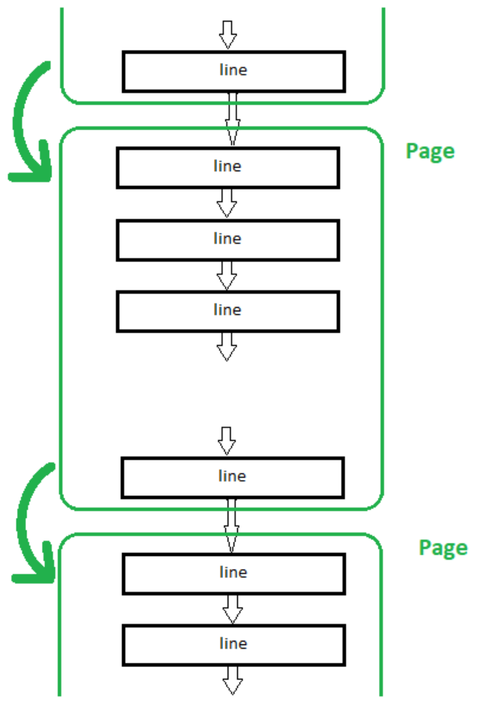
Text Editor

Data Structure for saving text:

The text is split into smaller parts (called lines) and saved in linked list for efficient insert and deletion.

The lines are combined into pages. These pages are numbered and allow binary search for efficient text lookup.



Class Page:

- Total lines in page
- Pointer to First line
- Pointer to next page
- Position of last character in the page

Class Line:

- Text
- Pointer to next line

Dictionary is stored in a sorted list and binary search is used for lookup.

A cache is used along with dictionary for quick lookup of frequent characters.

Resulting Performance:

On input data of **10 MB**, following timings were obtained:

	My approach	*Naïve approach*
100 cut paste operations	0.00429 sec	1.07717 sec
100 copy paste operations	0.01105 sec	0.54624 sec
100 misspellings operations	5.30213 sec	57.40903 sec
100 random operations	0.01237 sec	1.08636 sec

On input data of **1 MB**, following timings were obtained:

	My approach	*Naïve approach*
100 cut paste operations	0.00124 sec	0.14206 sec
100 copy paste operations	0.00259 sec	0.08857 sec
100 misspellings operations	2.40801 sec	5.74618 sec
100 random operations	0.00203 sec	0.11037 sec

Using 100 random operations is a better measure of performance as when a text editor is used, the order of operations used is random.

Running the program

No external files are used.

RUN AS: python text_editor.py

Inside class TextEditor, LINE_SIZE can be changed to see how lines are split or combined depending length

Complexities

Each page has the position of last character stored inside it. This is used in binary search to look for a specific position i in the entire document.

Each line stores the number of characters stored inside it.

To go to position i, first find the correct page using binary search and then find the correct line by linearly moving down in the found page.

If there are **P** pages and each page has **L** lines and each line has **W** words:

Then time taken to go to position **i** = O (lsog(P) + L)

Then time taken to copy from **i to j** = O (log(P) + L) + O (j – i)

Then time taken to cut from **i to j** = O (log(P) + L) + O (j – i) + O (P)

Then time taken to copy from **i to j** = O (log(P) + L) + O (j – i) + O (P)

Initialization:

At the start, the given text is broken down into line and stored as linked list. Lines are combined into pages for quick lookup.

Copy Implementation:

After finding the correct line corresponding to the given start position, text is copied from all lines till the given end position is reached.

Paste Implementation:

After finding the correct line corresponding to the given insert position, one of the following is followed:

1. If text can be added to the current line **LC** without crossing the max line length bound, then text is added to the line
2. Else
    a. the text following the insert position is appended to paste_text and a part from start of paste_text is added to the current line to satisfy the line length constraints.
    b. The remaining text in paste_text is broken into lines
    c. These lines are added after the insert line **LC.**
    d. The last line points to line after insert line (that is **LC.next_line**)
    e. New Page(s) may be added to follow page size constraints.

3. All the pages starting the current page are updated so that they store the correct position of last character stored in them.

Cut Implementation:

After finding the correct line corresponding to the given start cut position, one of the following is followed:

1. If the given end cut line is the same is start cut line, text is removed
    a. if the remaining text follows the minimum line length bound, move ahead
    b. else The line is combined with either the next line of previous line (if next line is not present).
    c. If the new line is longer then the permitted length, split it into two lines of equal length. (explained in next section)

2. Else:
    a. Find the end cut line
    b. Remove all lines from start cut line to end cut line
    c. If text in both start cut line and end cut line follows minimum length constraint, move ahead
    d. Else If end cut line and start cut line can be combined following the max length constraint, combine them else handle them by combining with next line or splitting into two.
    e. Similarly handle page, if needed, split or combine with other pages.

3. All the pages starting the current page are updated so that they store the correct position of last character stored in them.

Following the line and page length constraints.

These constraints are added so that the initial structure of data is maintainer. So that no line or page becomes too long or short.

For example, having a very long line will be same as having the entire text stored in a single string and will lower performance.

For both Page and Line,

Max_Length = 2 * Min_Length

This is taken so that if we need to combine a small line with another line, either the resulting line will be of proper length or if it is too long, it can be split into two lines of proper length.

This ensures that the affect of change in one line is not carried forward to other lines for long. This helps in good performance.

Dictionary Implementation:

A queue is used to keep track of recent words. Words are stored in a dictionary for efficient loop up (constant time operation in python).

When the cache is filled, first word from the queue is removed from both queue and dictionary.

Therefore lookup time in dictionary is O(1) if word is in cache else O(log x), where x = length of dictionary.

Trade off: extra space used to lower access time.

Trade off:

In order to make cut, copy and paste operations fast, text retrieval operation was slowed. As the text is broken down into many smaller parts, it takes a long time to recombine the parts again.

But it can be argued that, on a text editor, the main operations are cut, copy and paste. Text retrieval is not used that frequently.