IB Extended Essay

January 10, 2022

Word Count: 3979

Research Question: What are the respective benefits and disadvantages of the rope and gap buffer data structures in terms of time complexity? Do their performance match expected behaviours?

Contents

1 Research Question / Introduction

This paper will focus on buffers, data structures relevant to the implementation of text editors, programs that take and modify user input in order to produce an output. This connects to Topic 5 within the IB Higher-Level Computer Science curriculum. Buffers represent the basic unit of text being edited, and handle input and output to / for both the operating system and user [1]. However, there are various tradeoffs between the existing methods of buffer implementation in complexity, the amount of resources required for the program to execute, especially in terms of *time* and *memory* [2]. Two specific buffers of note are the *gap buffer* and *rope*, which will be detailed in Section 2. Hence, my research question is: **What are the respective benefits and disadvantages of the rope and gap buffer data structures in terms of time complexity? Do their performances match expected behaviours?**. I believe the data structures will perform according to expected behaviours described by time complexity, with the rope being generally more efficient.

2 Background Information

2.1 Big O

Big O notation shares characteristics with the eponymous notation in mathematics, which describes the limiting behavior of a function when an argument tends towards a value (or infinity). In computer science, it is applied as a "theoretical measure of the execution of an algorithm", usually in terms of time complexity, and describes the limiting behavior of an algorithm given a problem size of n [3]. In more detail, " $f(n) = O(g(n))$ means there are positive constants c and k, such that

$$0 \leq f(n) \leq cg(n)$$

for all

$$n \geq k$$

. The values of c and k must be fixed for the function f and must not depend on n" [3].

As an example of Big O notation applied in computer science, a code segment with constant execution time will have a time complexity of $O(1)$. This represents the code's execution time graph behaving like a constant function as the size of the problem n increases, meaning that it takes the same amount of time to execute regardless of n. For the sake of simplicity, in cases where multiple operations are composed, the time complexity is that of the most complex operation.

Big O notation only expresses limiting behavior of an algorithm's time complexity. Therefore, with smaller sample sizes, an algorithm with a "worse" time complexity but lower "coefficient" may perform better than an algorithm with supposedly better time complexity [4]. For example, if a $O(1)$ function has a significantly higher initial value than a slow-scaling $O(\log n)$ function, the latter will perform better at lower values of n.

2.2 Gap Buffers

Gap buffers are a type of buffer implemented as an array or another form of block memory with a moveable internal "gap", which does not store data. How the gap is represented depends on the underlying data structure, but it is generally represented with three components:

- Start of gap
- End of gap
- Length of gap

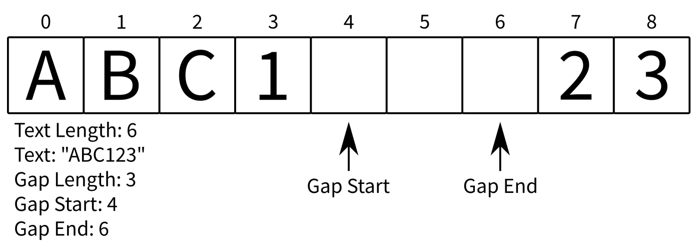

Figure 1: A visual representation of a gap buffer. Text is stored on both sides of the buffer (own work).

The start and end of the gap can be stored as pointers in an array, links in a linked list, indices, or any other metric of position based on the underlying data structure. The length of the gap should be a numerical value. As this project's gap buffer is implemented using a linked list, the methods described below will make reference to this implementation type. Therefore, "incrementing" and "decrementing" pointers will refer to changing a pointer to a link to its next or previous link, respectively.

For an insertion or deletion operation, the gap must first be "moved" to the desired index, if it is not at that position [5] [1]. The data between the current and desired gap positions must also be shifted to preserve continuity, as demonstrated in Figure 2. In this project's implementation, after the gap is delinked from its surrounding nodes, n nodes must be travelled to move it. This results in a time complexity of $O(n)$, where n represents nodes travelled. Because of the slight optimisation of the linked list described in Section 4.2.1, the worst-case for indexing is moved from the end of the linked list to the centre, after which theoretical indexing times should start to decrease.

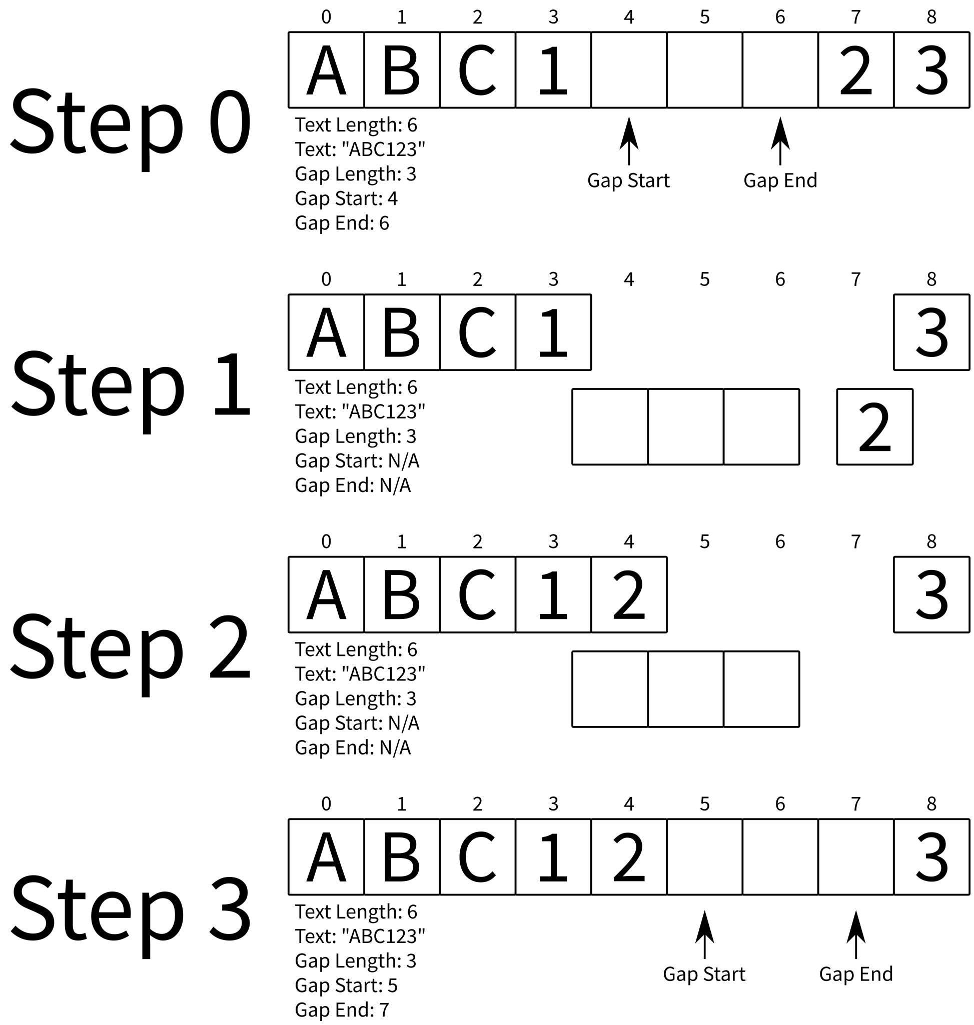

Figure 2: A visual representation of moving the gap, in order of steps described. The data after the gap is moved to the position before the gap (own work).

After the gap is positioned, insert and deletion operations can be performed at the gap. Inserting information involves replacing characters at the start of the gap while incrementing the start position pointer and decrementing the gap size, as demonstrated in Figure 3. This results in a time complexity of $O(n)$, where n represents the inserted length [6] [1].

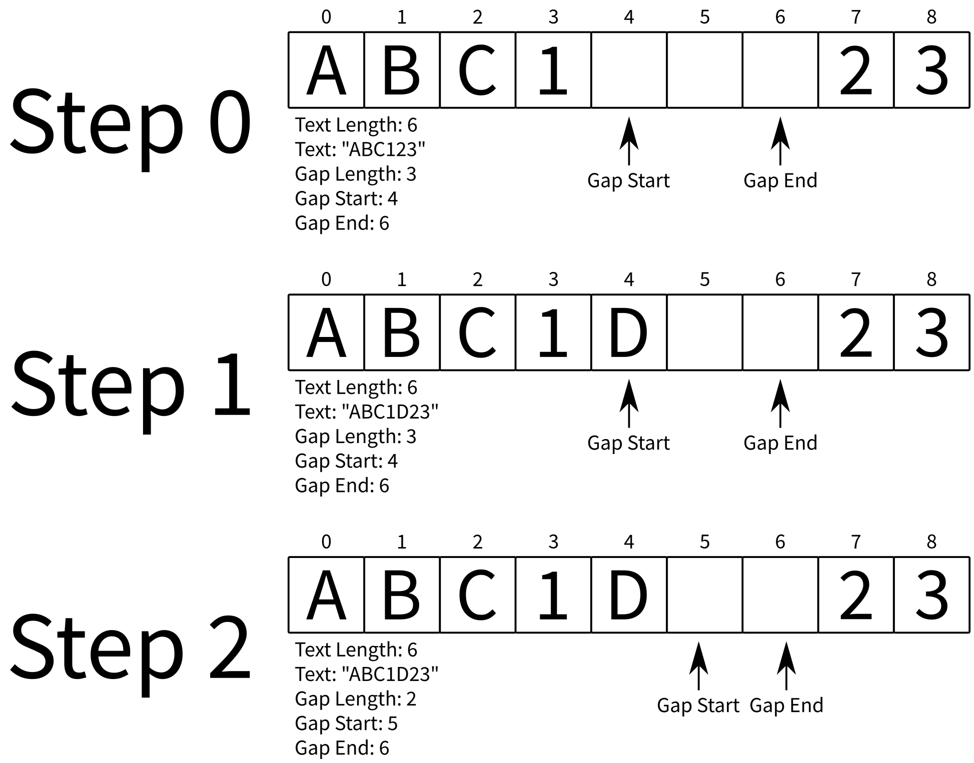

Figure 3: A Visual representation of inserting in a gap buffer, in order of steps described (own work).

However, if the information to be inserted is longer than the length of the gap, the gap must be extended, a time-intensive process that requires allocating new memory to make space for the new data. In the buffer implemented, this is an $O(n)$ process because it involves creating n new nodes [1].

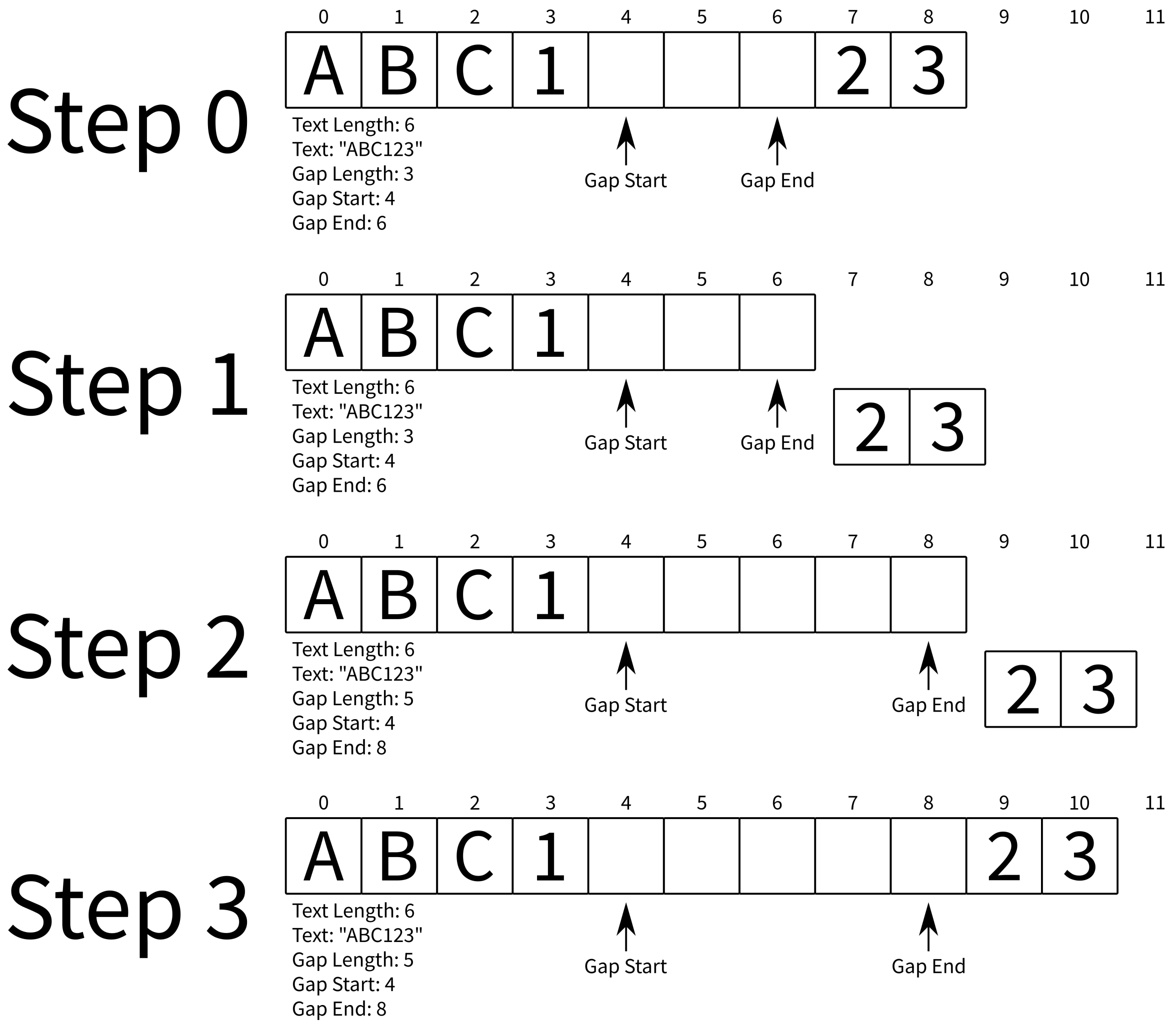

Figure 4: Visual representations of gap extension, in order of steps (own work).

Deleting information involves ”extending” the end of the gap over the deleted indices, saving time on memory allocation by increasing the gap’s size and therefore available space, as demonstrated in Figure 5. In this project’s implementation, once the gap is moved to the correct position, the gap end pointer is incremented with gap length until a sufficient amount of text has been deleted. This is an $O(n)$ process because it involves travelling across n deleted nodes [7].

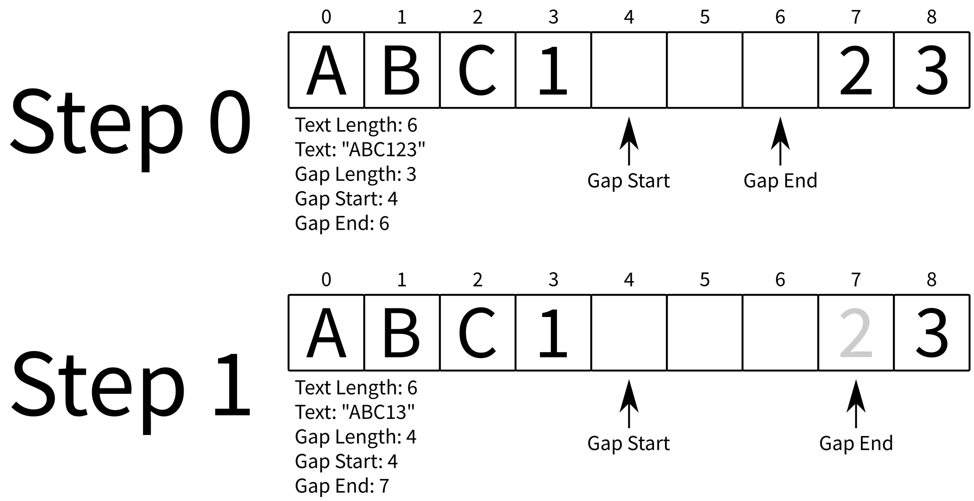

Figure 5: Visual representations of deleting in a gap buffer, in order of steps. The grayed-out character represents a value within the gap, which should be excluded during the report process (own work).

Reporting (getting) and indexing the contents of a gap buffer depend on the underlying data structure. In this project's implementation, the former is an $O(n)$ process because it involves travelling over n nodes to either get to an index or report n characters. Links in the gap are always skipped in counting for length and in being actually reported in the final text. The latter was discussed previously.

Concatenating another gap buffer to a gap buffer can be done in $O(1)$ time for this project's implementation, as the tail of one's linked list can effectively be linked to the head of the other.

Constructing (loading) a gap buffer from existing text involves the following steps:

- Constructing the underlying data structure of a sufficient size (including the gap)
- Initialising the gap. It is convenient for the gap to begin at either the start or end of the data structure, because it allows the text to be inserted without consideration of the gap
- Inserting the existing text into the gap buffer.

This is usually an $O(n)$ process where n represents size inserted, as the insertion time usually dwarfs the time to create the gap.

Gap buffers are simple to implement, and have reasonable performance for inserts and deletes at the gap (as well as relatively fast report times), but moving the gap itself is costly [1].

2.3 Ropes

Ropes (often referred to as cords) are buffers that use a binary tree to store text, with an additional "weight" attribute on each node, and are described by Boehm et al. in the referenced paper [8]. The basic functionality of binary trees will not be detailed within this paper. The "weight" parameter refers to the length of the text stored in the left subtree of a given node. Text is only stored in the leaves, and can be greater than length 1 [9].

Indexing the node that contains a certain position is effectively a binary search that is run using the weight attribute of each node, as demonstrated in Figure 6. As a result, this is an $O(\log n)$ process.

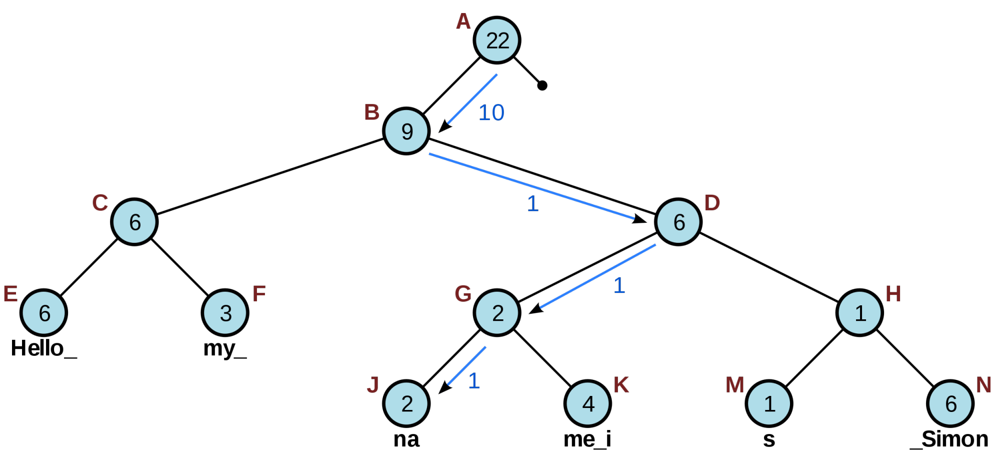

Figure 6: A demonstration of the process of indexing a rope. When moving left, the search index is maintained. When moving right, the current node's weight is subtracted from the search index.

(From `https://commons.wikimedia.org/w/index.php?curid=30952989`)

The two key manipulations of a rope are the *split* and *concatenation* methods, which can be combined into further operations [8].

The split method splits the rope at a position, leaving subtrees S_1 (which includes all data before the split point) and S_2 (which includes all data after the split point). This operation is implemented by first indexing the node containing the position, then travelling up the tree while breaking off and storing nodes to the right of the indexed node (as demonstrated in Figure 7). Weights should be updated when necessary. The broken-off nodes should be concatenated together in order to form S_2.

A special case is when the index is in the middle of a node's data, in which case a new node should be created with the data to the right of the index, and is the first item stored during the breaking off process. To allow for continued efficient operations, both of the resulting trees should be rebalanced [9].

Splits are $O(\log n)$ processes because they involve travelling through the height of the tree, which scales logarithmically with input size in balanced trees.

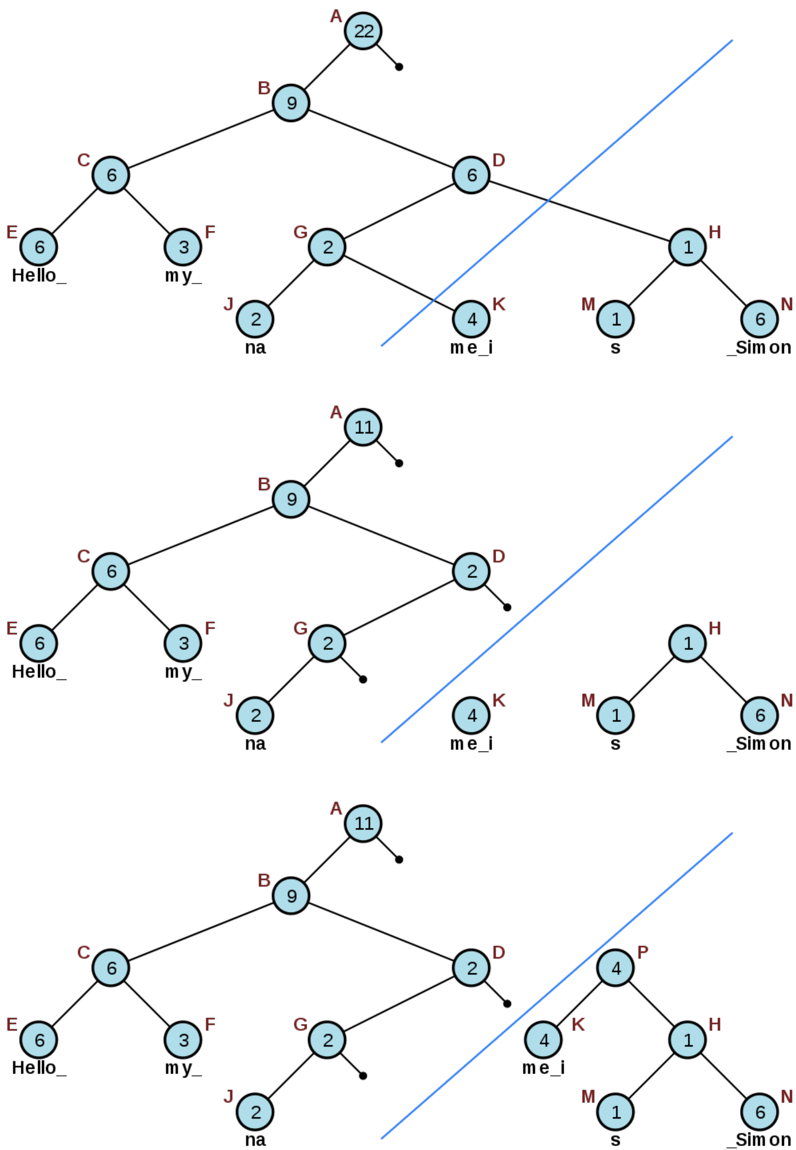

Figure 7: A demonstration of the process of splitting a rope. Elements to the right of the desired split index are broken off, and then concatenated together (with weights updated as necessary).

(From `https://commons.wikimedia.org/w/index.php?curid=30953234`)

The concatenation (concat) method combines two subtrees (left to right) S_1 and S_2 into one new tree, as demonstrated in Figure 8. This is an $O(1)$ process in ropes that use a different balancing scheme than described by Boehm et al, which may require tree reconstruction and thus has a $O(n)$ time complexity, justified later in this subsection [8] [10].

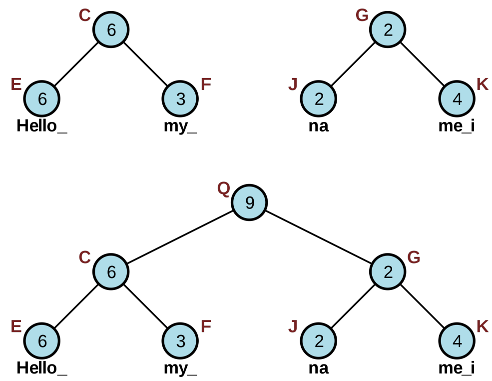

Figure 8: A demonstration of the process of concatenating two ropes. A new parent is created for the two, inheriting the weight of the left rope. Rebalancing may be required on more complicated ropes. (From `https://commons.wikimedia.org/w/index.php?curid=30953529`)

Table 1 describes the implementation of append, insert, and delete operations off of these two fundamental components. They inherit the time complexities of their constituent operations, which are generally $O(\log n)$ except the $O(1)$ concat used in append. Theoretically $O(\log n)$ operations may happen in $O(n)$ time in poorly-written implementations due to tree imbalance, which prevents efficient searches.

Append	Create rope using new text, concat to right of current rope.
Insert	Create rope using new text, producing S_3. Split rope at insert point, creating S_1 and S_2. Concat S_1 to S_3, producing S_4. Concat S_4 to S_2.
Delete	Split rope at end of deleted range into S_1, S_2. Split S_1 into S_3, S_4 at start of deleted range. Concat S_3 and S_2 together.

Table 1: Table of additional operations. Concats are expressed with left element first, and right element second.

To report the contents of a rope, first, the starting node should be indexed using the appropriate method.

Then, the tree should be traversed in-order from this point, until a satisfactory amount of data is reported. In-order traversal methods will not be discussed in this paper. This is an $O(\log n)$ operation because it usually involves visiting $\log n$ nodes in the tree, where n is the length reported [9].

To construct (load) a rope from existing text, the text should be divided into equally-sized segments, and then recursively concatenated in order, forming a perfectly balanced rope. This is an $O(n)$ operation because it involves the creation and concatenation of n new nodes.

Ropes have reasonable performance for concatenation and splitting text, and can have good memory performance and the ability to implement edit history. However, single character replacements and indexing are time-intensive, and iteration over the structure is complex. Additionally, memory overhead is significant, and implementation of a self-balancing rope is difficult [8].

3 Testing Methodology

To verify the conformity of the data structures to their time complexities and to compare their execution times, they were first implemented using the Go programming language. Further attributes are described in Section 4. They adhere to the following interface, which also expresses the features that were tested.

```go
// Rune represents a character in Go
type StorageType interface {
	Report() ([]rune, error)                  // report entire buffer
	Insert(i int, content []rune) error       // insert content at position i.
	Append(content []rune) error              // used during testing, ignore
	Concat(s StorageType) error               // self-explanatory
	Split(i int) (StorageType, error)         // splits at point i, returning a new
	    ↪ buffer [i ... n]
	DeleteRange(i, j int) ([]rune, error) // deletes from [i, j)
	Load(contents []byte) error               // loads contents into buffer
	ToString() string                         // self-explanatory
	Index(i int) (rune, error)                //  returns character at position
}
```

Tests were run wih a UTF-8 text file providing input data, in this case a book downloaded off of Project Gutenberg [11]. The standard Go benchmarking suite, part of the testing package, was used to measure times per operation, and to set up and run each round of tests. Exact system state/specifications during testing were assumed to be irrelevant. Compiler settings were not changed across tests. Since this paper's goal is to produce *representative* results in the form of determining adherence to end behaviours and general trends, system variations that may have affected the *specific* results produced were thought to be irrelevant.

Table 2 outlines the specifics of the tests run. Test sizes were large in order to produce measureable times, and to better approximate measure *end* behaviour. n represents scaling with the *size* of the buffer, while k represents scaling with the *position* within a buffer.

The *independent* variable is indicated in the sub-test column, with greater detail about its specific variation in the increments column. The *dependent* variable is always average time per operation, measured in nanoseconds by the testing benchmarking suite. On "relative" tests, the fractions in the increments column refer to a proportion of the buffer size in some way. Since, in these tests, buffer size was varied in conjunction with the tested factor, this proportional approach allowed for spaced data points regardless of buffer size.

Test	**Sub-test**	**Expected Behaviour**		**Increments**	**Constants**	**Delay?**
		Rope	**Gap Buffer**			
Index	Buffer Size	$O(\log n)$	$O(1)$	10: 40000 character length increase in load size for each, starts at 40000 and ends at 400000	Position Indexed: 20000	Yes, 1ns
	Relative Position	$O(1)$	$O(k)$	7: For each buffer size, indices at 0, 1/8, 1/4, 1/2, 3/4, 7/8, $n-1 \approxeq 1$ of the buffer size	N/A	Yes, 1ns
Load	Buffer Size	$O(n)$	$O(n)$	See Index - Buffer Size subtest	N/A	No
Report	Buffer Size	$O(n)$	$O(n)$	See Index - Buffer Size subtest	Reporting entire buffer	Yes, 1ns
Split	Buffer Size	$O(\log n)$	$O(1)$	See Index - Buffer Size subtest	Split Index: 20000	No

	Relative Position	$O(1)$	$O(k)$	5: For each buffer size, indices at 1/8, 1/4, 1/2, 3/4, 7/8 of the buffer size	N/A	No
Insert	Buffer Size	$O(\log n)$	$O(1)$	5: 80000 character length increase in load size for each, starts at 80000 and ends at 400000	Insert at index 40000, inserting 2000 characters	No
Delete	Buffer Size	$O(\log n)$	$O(1)$	See Index - Buffer Size subtest	Deletion range: between indices 10000 and 30000	No
Concat	Buffer Size	$O(1)$	$O(1)$	See Index - Buffer Size subtest	Concat Input size: 2000 character buffer of same type	Yes, 1ns
	Relative Length	$O(1)$	$O(1)$	6: For each buffer size, lengths of 1/8, 1/4, 1/2, 3/4, 7/8, 1 of the buffer size concated	N/A	Yes, 1ns

Table 2: Table of tests, and a review of expected results. Some tests included a 1ns delay to provide more consistently measureable results, represented in the delay column

As the testing library returns average time results for a test after sufficient iterations to establish a consistent average, and the delay was consistent across all trials, the 1 nanosecond (ns) delays after some test operations likely did not significantly influence the accuracy of results. At worst, they were averaged out by the benchmarking software [12].

After 5 trials of each test, data was averaged and graphs were created, with basic information about variance between tests visually given in the form of standard deviation bars, which will be visually compared. Regression curves of the appropriate time complexity type were fitted.

The R^2 (a common measure of correlation with regression curves, with 1 representing a perfect correlation) of these curves were used as a measure of whether the data structures adhered to expectations, with an R^2 exceeding 0.8 suggesting a good correlation of the data to a certain trend, and therefore a certain time complexity. Simple visual comparison was used to compare their performance.

In tests where a factor other than buffer size is changing, the slopes of the linear regression for each buffer size were compared using a single-sample T-test, which produces a p-value, representing the probability that a null hypothesis is true. In this case, the null hypothesis is that there is no statistically significant difference between the set of values and an expected mean, which was always 0 (representing a constant time complexity's slope). p-values greater than 0.05 mean that the null hypothesis cannot be refuted, meaning that there is no statistically significant difference between the slopes and the expected average constant time slope.

The use of these tests allow for the simplified comparison of behaviours across multiple buffer sizes according to an expected standard, foregoing the need for graphs. Instead, a "representative" graph taken from the largest tested buffer size was used as an example of the trends for other cases, as the large test size should amplify their visibility.

4 Implementation Details

4.1 Rope Details

The rope tested was implemented using an AVL-type tree. This type of tree strictly enforces no greater than 1 level of height difference between subtrees of a given node.[13] Boehm et al do not use this type of balancing in their paper but consider it a viable option, as the stricter balancing can facilitate better operation times [8].

Each leaf stored a maximum of 10 characters. The influence of this parameter on results was not tested.

The rope implementation does not merge underfilled leaves, which may result in unnecessarily large tree sizes after successive edits, decreasing performance. However, since the buffers were completely reset after each operation during testing, this deficiency did not significantly affect the results produced.

A consideration for evaluation of the rope that will not be covered in this paper is its implementation complexity. If efficiency is judged to be a meaningful tradeoff for development time, alternatives may be used.

4.2 Gap Buffer Details

The gap buffer tested was implemented using a doubly-linked list, which allows for faster movement of the gap at the cost of slower data access and higher memory usage. Double links were used to slightly increase data access speed, as discussed in Section 4.2.1. The "gap" is stored with pointers to its beginning and end nodes in the doubly linked list, its start index, and its length.

In more realistic usage, the gap should be initialised with a size larger than an average input, so as to reduce reallocations by accomodating most realistic inputs, but this functionality was not implemented.

When evaluating the results produced by this data structure, it is important to note that almost all tests required large gap movements, a time-intensive process. However, in practical use, small shifts near the current gap position are more likely. The current implementation requires a costly indexing operation for all non-end gap movements, whereas small shifts could be handled more efficiently using the pointers before and after the gap if the position is closer to the gap start than an end.

Finally, the gap buffer uses excessive memory and has excessive operation times because it stores every character in its own individual link. In an ideal implementation, to reduce memory usage and retrievals, more characters should be stored in each link, or an array should be used instead of a linked list. However, the former is more difficult to implement, and the latter will require significant memory allocation times upon gap expansion.

4.2.1 Linked List Details

The doubly-linked list used in the gap buffer is a typical implementation that stores the end pointer.

During indexing, a travel direction (from start or end) is selected depending on which is closer to the desired position, allowing for faster access times near both ends of the list. This results in linear time complexity, with a peak at the centre of the linked list, which will appear polynomial.

5 Results

5.1 Index

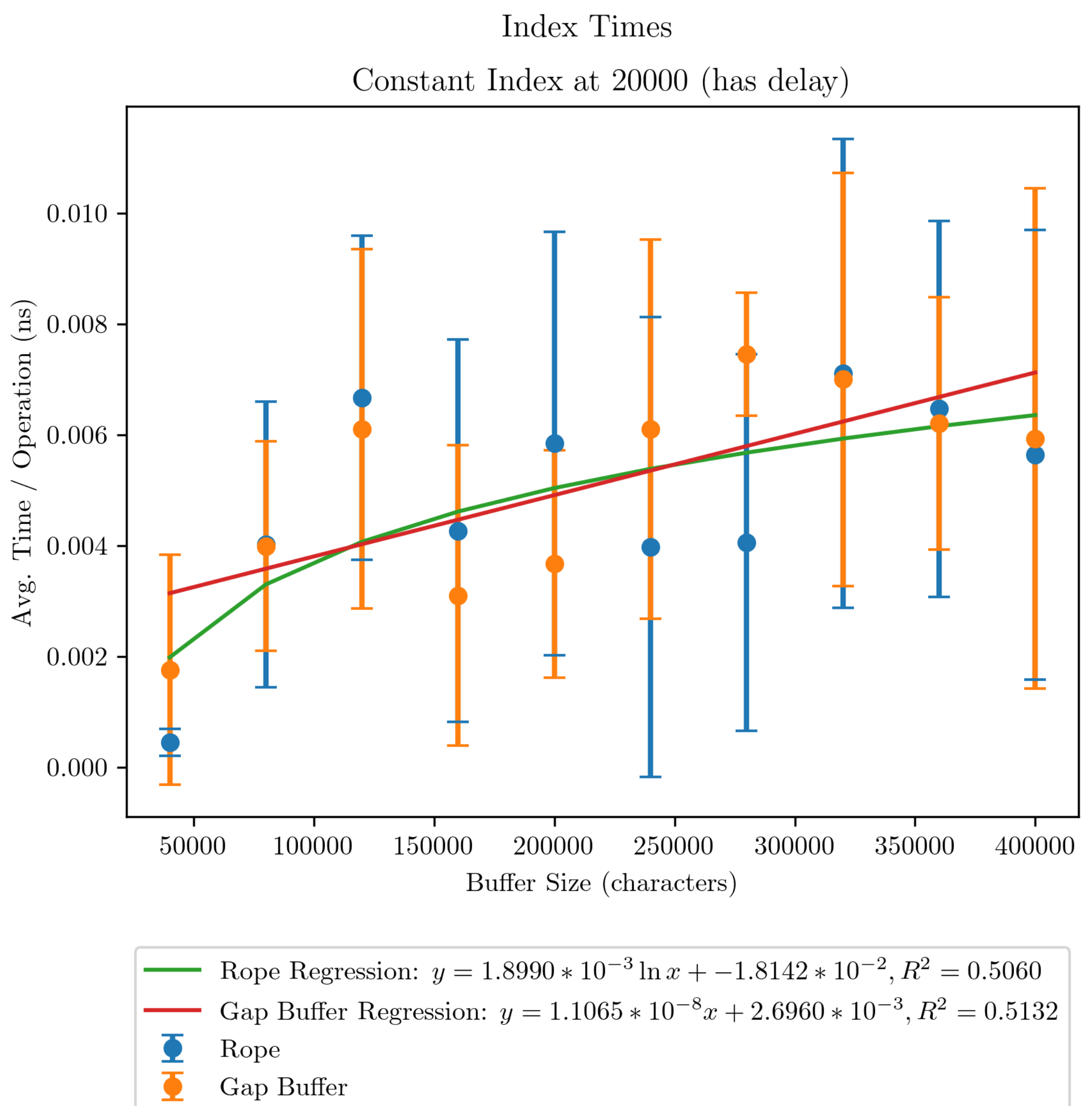

Figure 9: Index times, varying buffer size

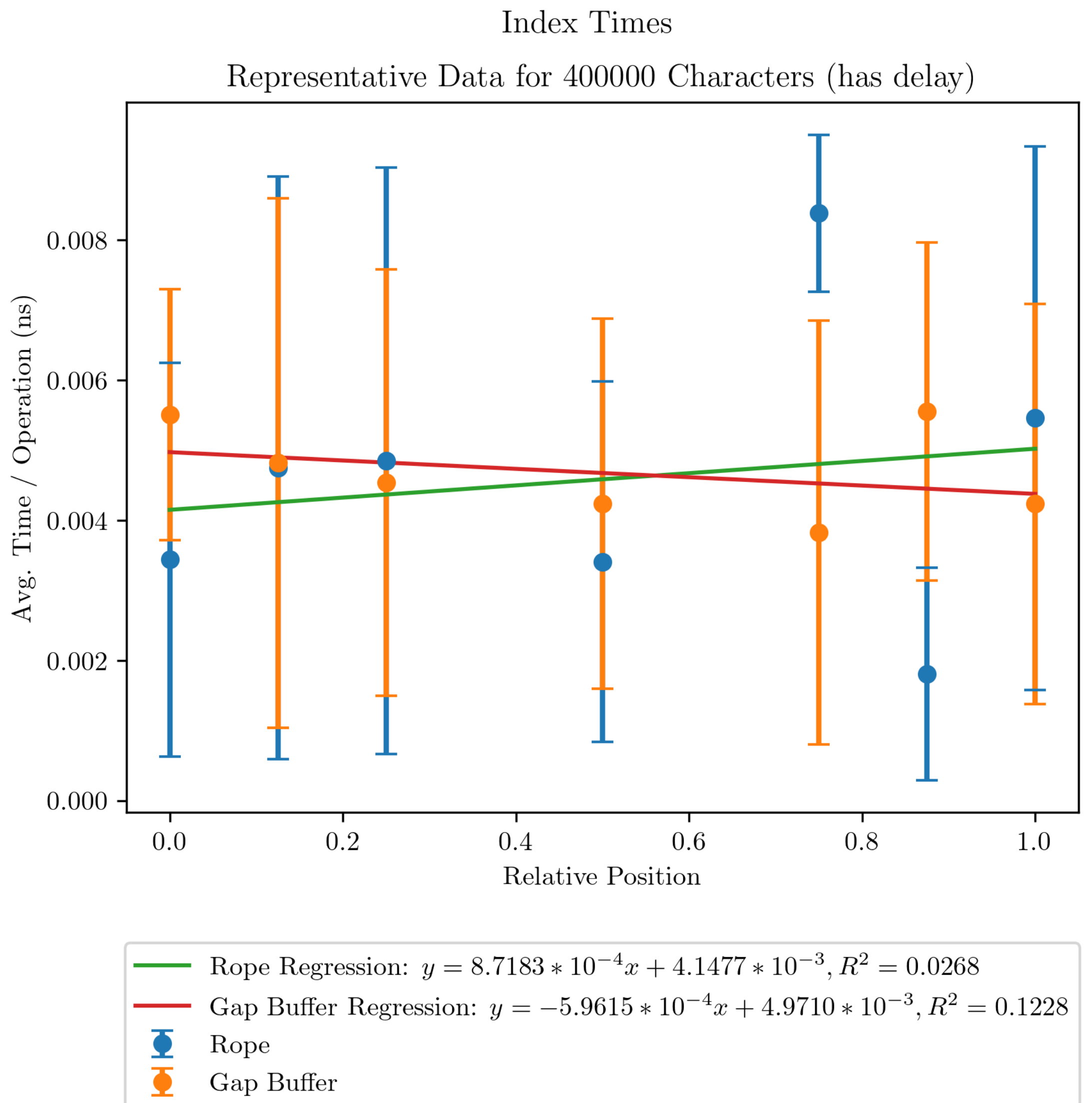

Figure 10: Representative index times, varying relative position

Indexing results across both sub-tests had significant variance (indicated by error bars on the graphs), and therefore cannot justify compliance with expectations.

Buffer Size	Rope Slopes	Gap Buffer Slopes
40000	-0.001622422	0.0030767281
80000	0.0015821302	0.0021261695
120000	-0.001186442	0.0009978332
160000	-0.001591962	-0.002000174
200000	0.00052248	-9.617627E-06
240000	-0.002438146	0.0006781722
280000	-1.533017E-05	-0.002237946
320000	0.0014709003	0.0007787471
360000	-0.001125356	-0.002032095
400000	0.0001992963	-8.535864E-05

	Rope	Gap Buffer
Sample Size	10	10
Sample Mean	-0.000420485	0.000129246
Sample STDEV	0.001374238	0.0017959129
Hypothesised Mean	0	0
Test Statistic	-0.967584173	0.2275787648
Degrees of Freedom	9	9
P-Values	0.3585325674	0.825059566

Figure 11: Table of T-test results for index

However, as demonstrated by Figure 11, in the relative position test, the slopes of the linear regressions produced by different buffer lengths that were likely to be statistically indistinguishable from 0. This suggests that the tests generally exhibited constant time behaviour, but again the variance makes this judgement uncertain.

5.2 Load

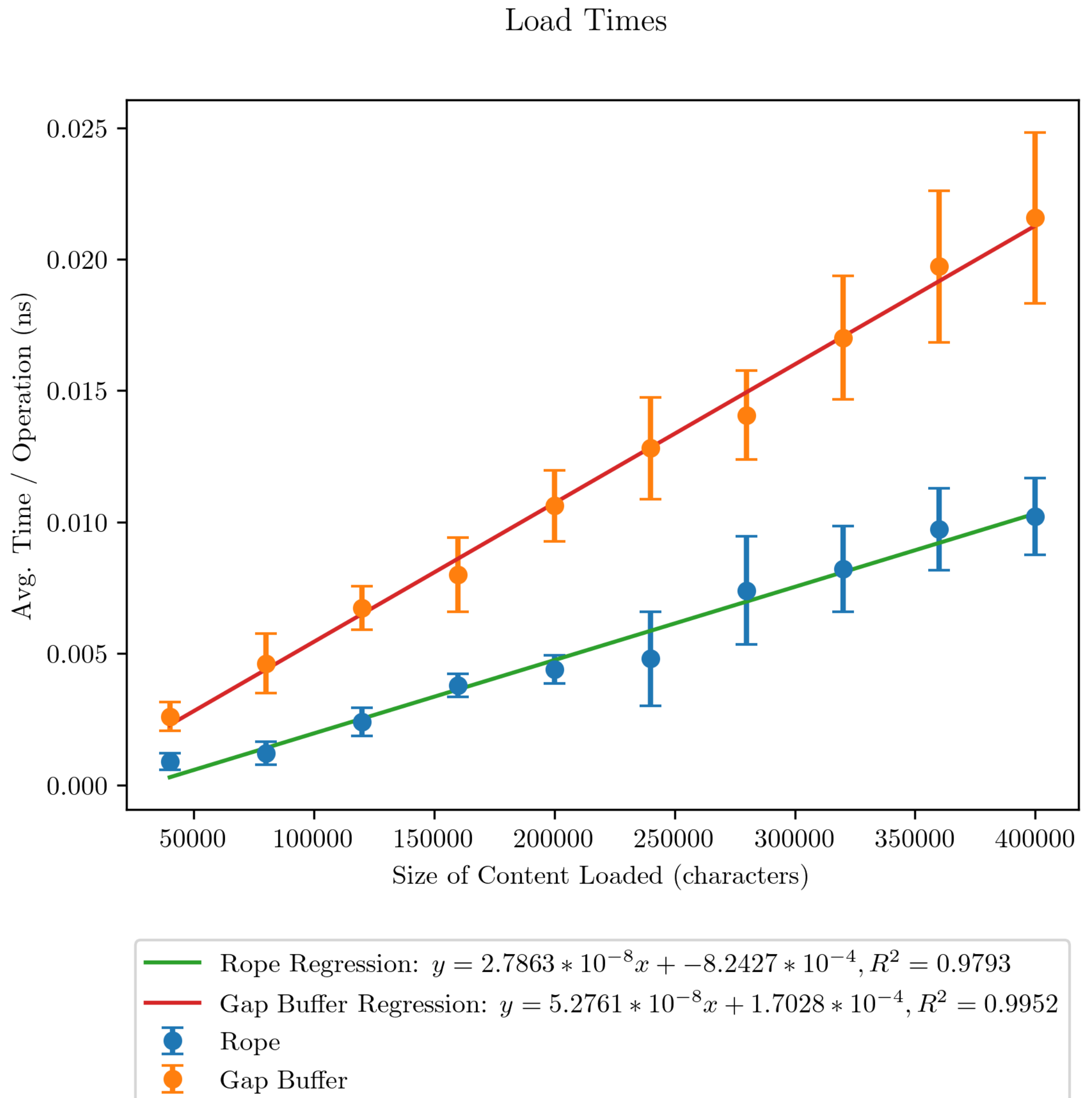

Figure 12: Load times, varying buffer size

Load time tests produced results that conformed to expectations. Both data structures adhered to their linear time complexities, and the rope performed better than the gap buffer. Variance was relatively high but acceptable.

5.3 Report

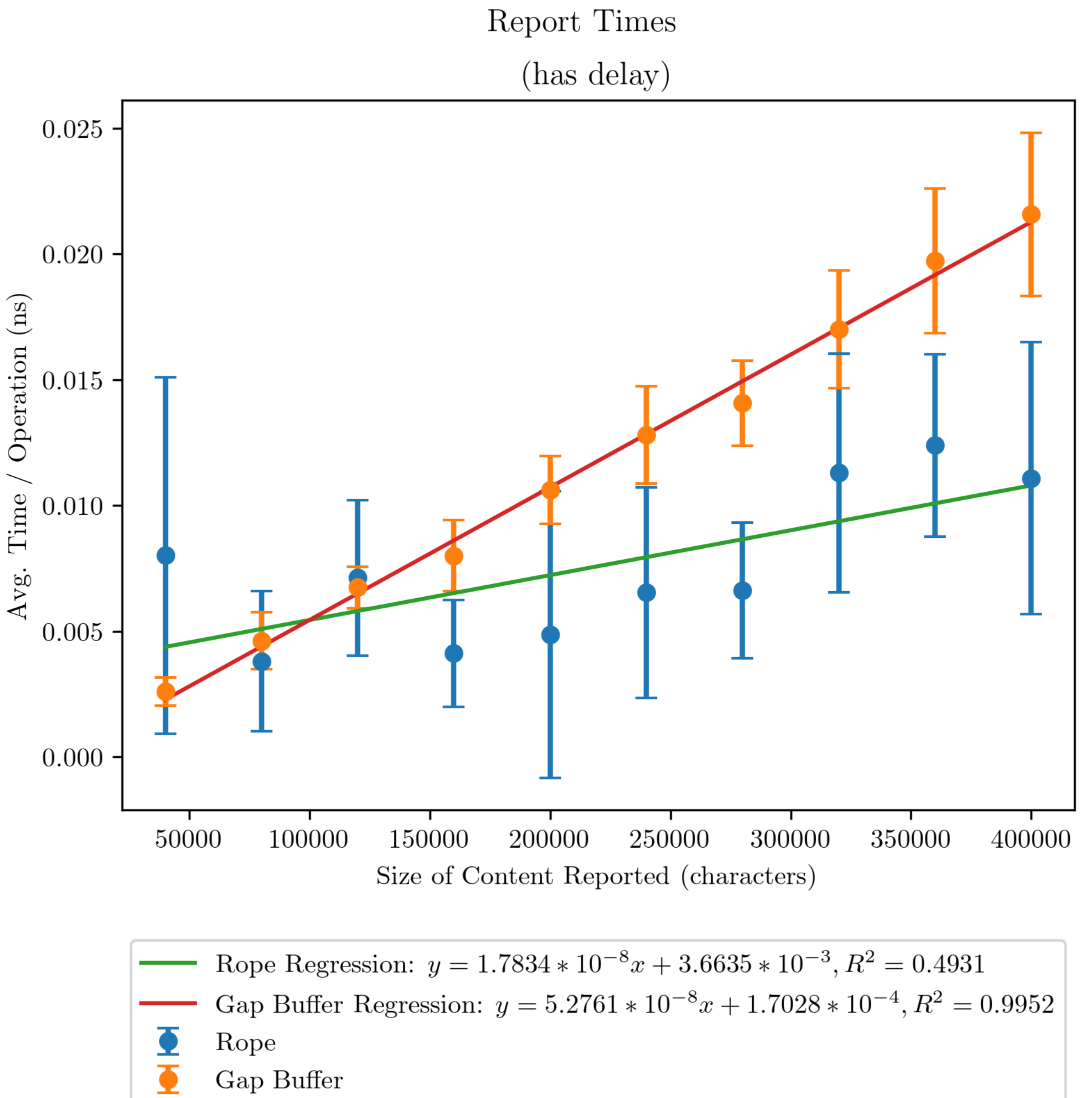

Figure 13: Report times, varying buffer size

The gap buffer's performance data has reasonable R^2 values and fair variance, so it can be said to perform according to expectations. The rope's exhibits comparatively high variance and a low R^2, so no judgement can be made about it. However, even at the extremes of its error it appears to perform better than the gap buffer.

5.4 Split

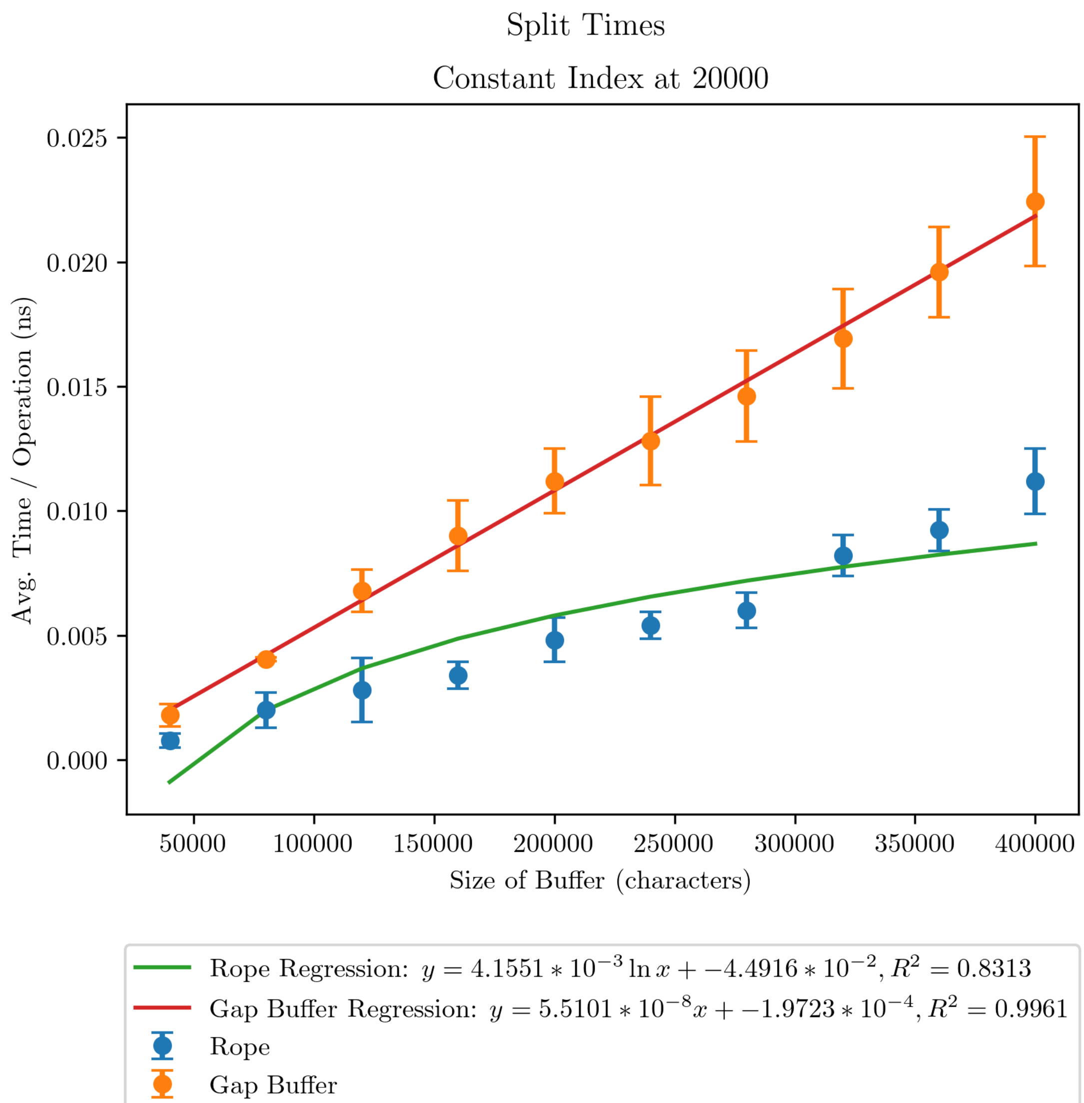

Figure 14: Split times, varying buffer size

For the constant position test, the gap buffer did not achieve its theoretical constant time performance (reasons for this will be discussed further in Section 6.1), but the rope did achieve a somewhat logarithmic performance. As expected, the rope performed better than the gap buffer. Again, variance was more pronounced on the gap buffer, but was low.

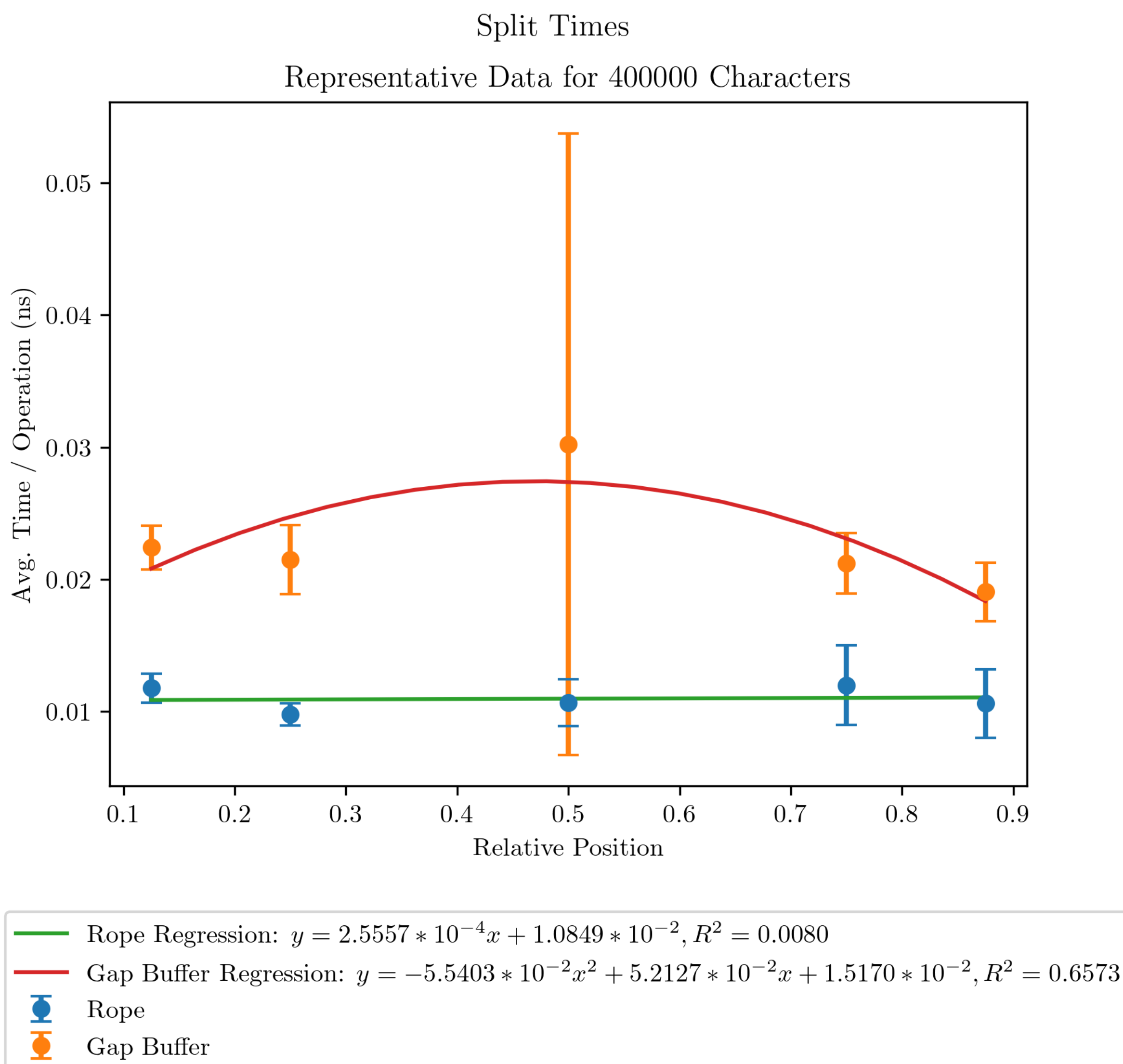

Figure 15: Representative split times, varying relative position

Relative position tests produced results with R^2 values far too low to support any conclusion. The polynomial-appearing performance of the gap buffer is not surprising - since the index operation is the most complex part of this operation, its previously described polynomial-appearing behaviour is inherited. As for the rope, the outcome is logical as the number of nodes visited and split off does not vary when the tree size is kept constant, which is the case for the representative data.

Buffer Size	Rope Slopes	Gap Buffer Slopes
40000	-2.275846E-09	3.84E-09
80000	7.836615E-09	-1.600154E-08
120000	-1.407179E-09	-7.539487E-09
160000	1.655385E-09	-7.775E-09
200000	4.903077E-09	-5.575385E-09
240000	-2.231538E-09	3.592308E-09
280000	-1.324615E-09	1.915165E-08
320000	1.509615E-10	4.317308E-09
360000	4.139658E-09	-3.634188E-09
400000	6.389231E-10	-8.189231E-09

	Rope	Gap Buffer
Sample Size	10	10
Sample Mean	1.208544E-09	-1.781357E-09
Sample STDEV	3.416447E-09	9.809532E-09
Hypothesised Mean	0	0
Test Statistic	1.118633375	-0.57425208
Degrees of Freedom	9	9
P-Values	0.2922636287	0.5798644417

Figure 16: Table of T-test results for split

Figure 16 shows the slopes for the linear line of regression of the relative position tests produced by the indicated buffer sizes, and the p-value produced y the t-tests. In the case of the rope, this suggests that the other tests also exhibited the expected constant time behaviour. In the case of the gap buffer, since the data mapped to an polynomial-type curve would return a constant-type regression if fitted to a linear curve, this result is also not surprising.

5.5 Insert

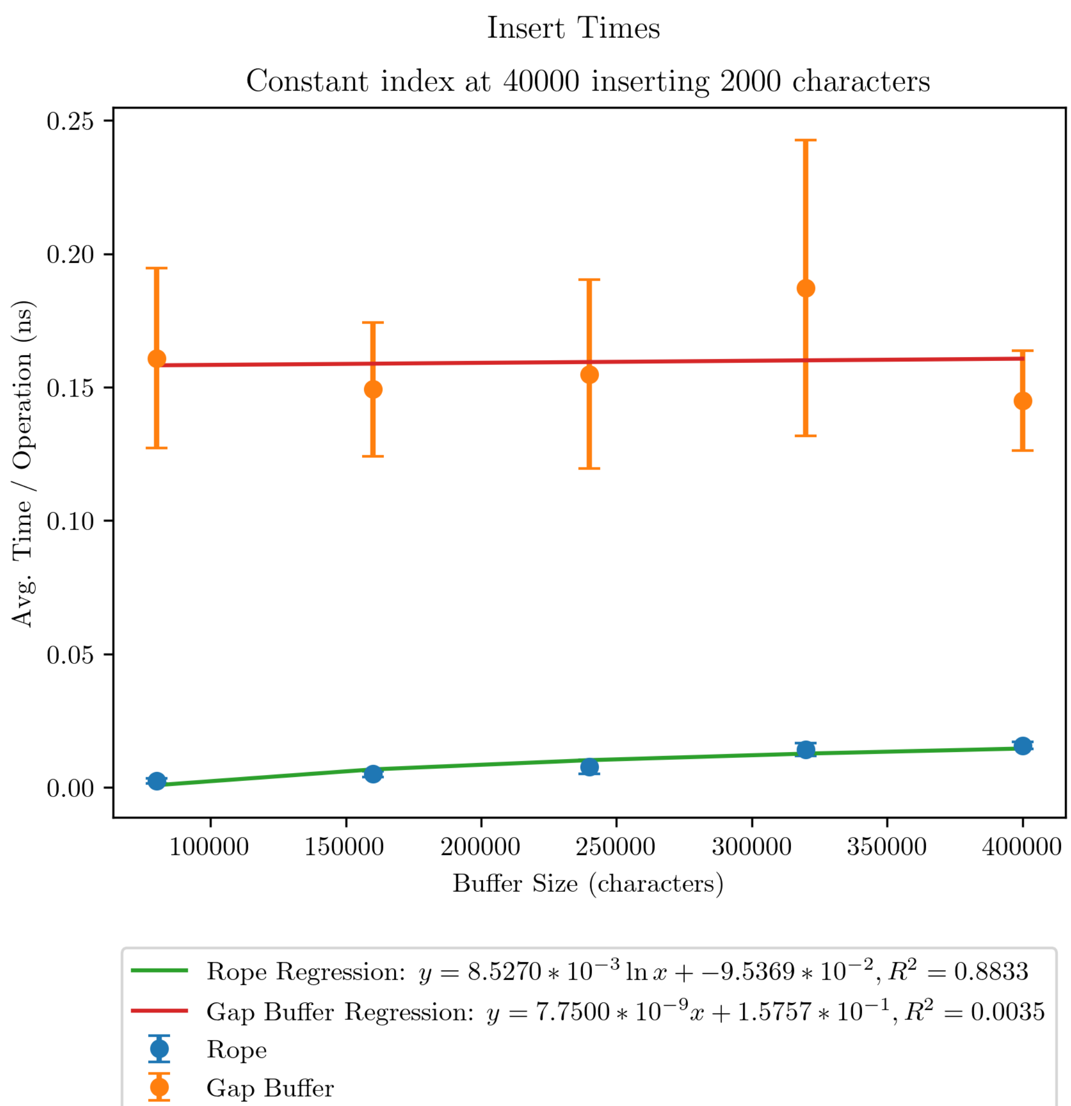

Figure 17: Insert times, varying buffer size

The rope performed according to expectations, performing better than the gap buffer. The gap buffer's performance data has a low R^2 and significant variance, precluding a conclusion, but the constant trend is visible.

5.6 Delete

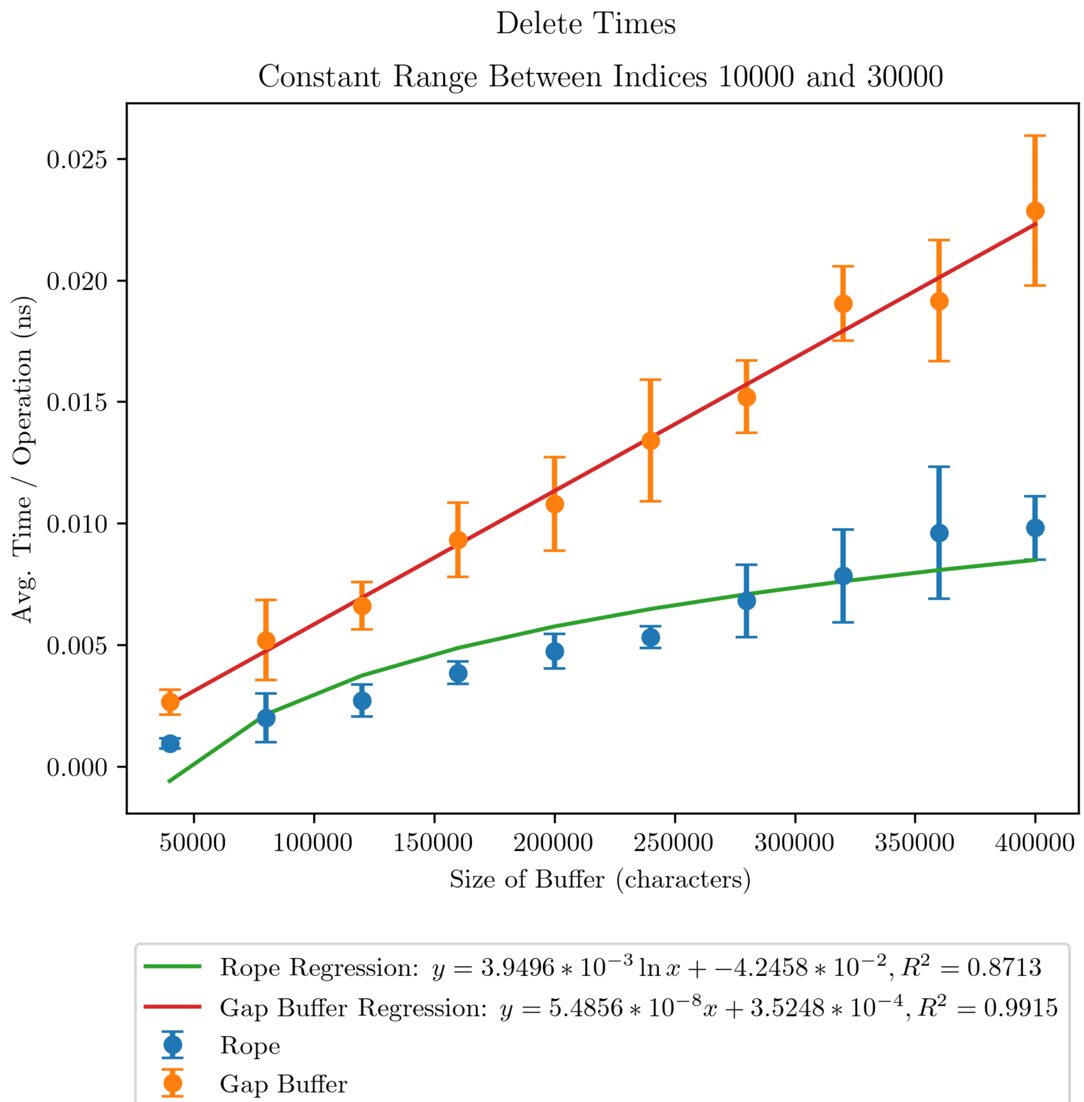

Figure 18: Delete times, varying buffer size

The gap buffer did not achieve its expected constant performance (which will be discussed further in Section 6.1), but the rope did, resulting in better performance. Variance was similar across both buffers, with fair R^2 values.

5.7 Concat

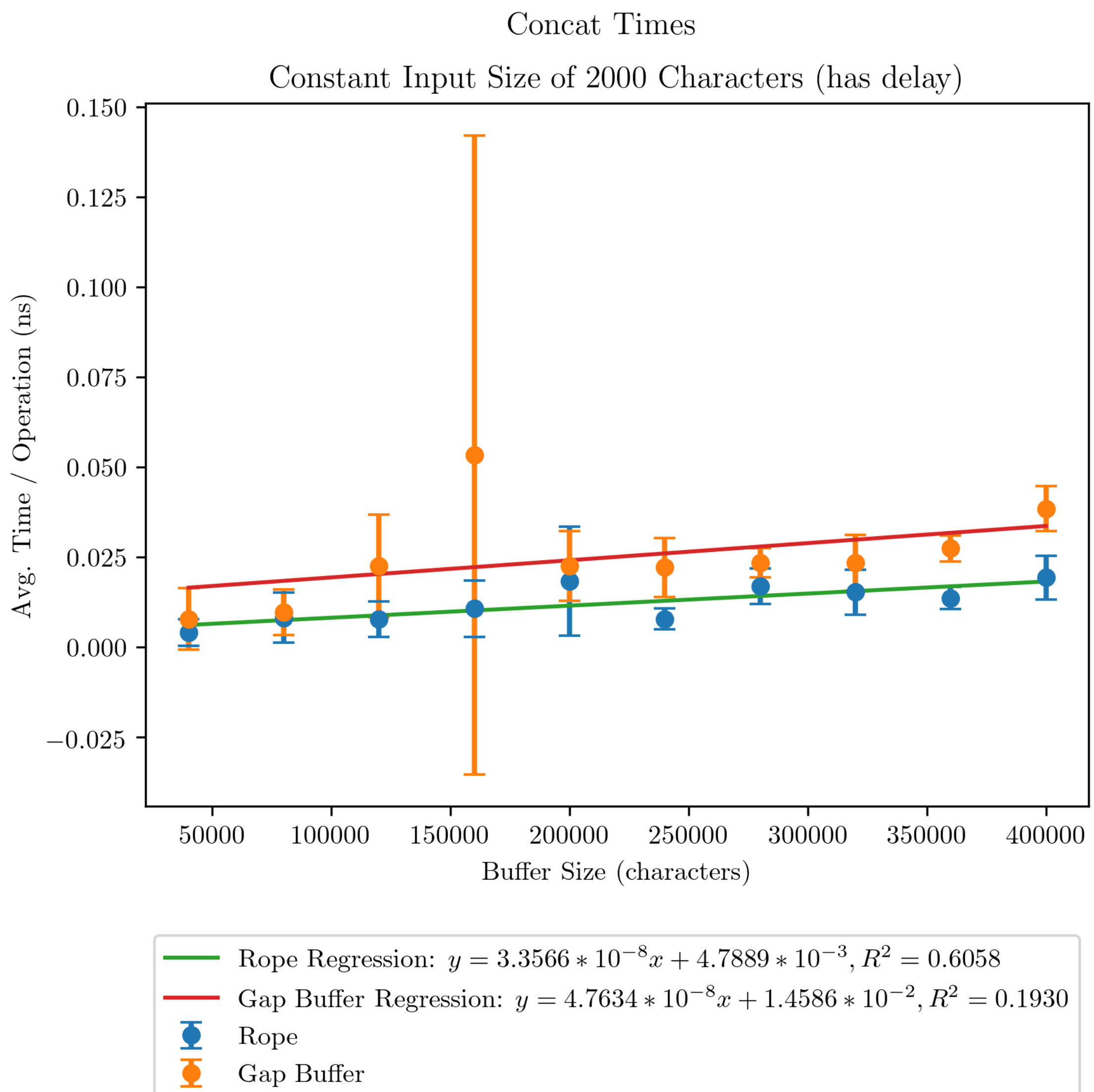

Figure 19: Insert times, varying buffer size

While the results appear linear, concat data while varying buffer size has too much variance to reach a conclusion, further demonstrated by the low R^2 values. Variance was especially pronounced with the gap buffer data at 160000 characters.

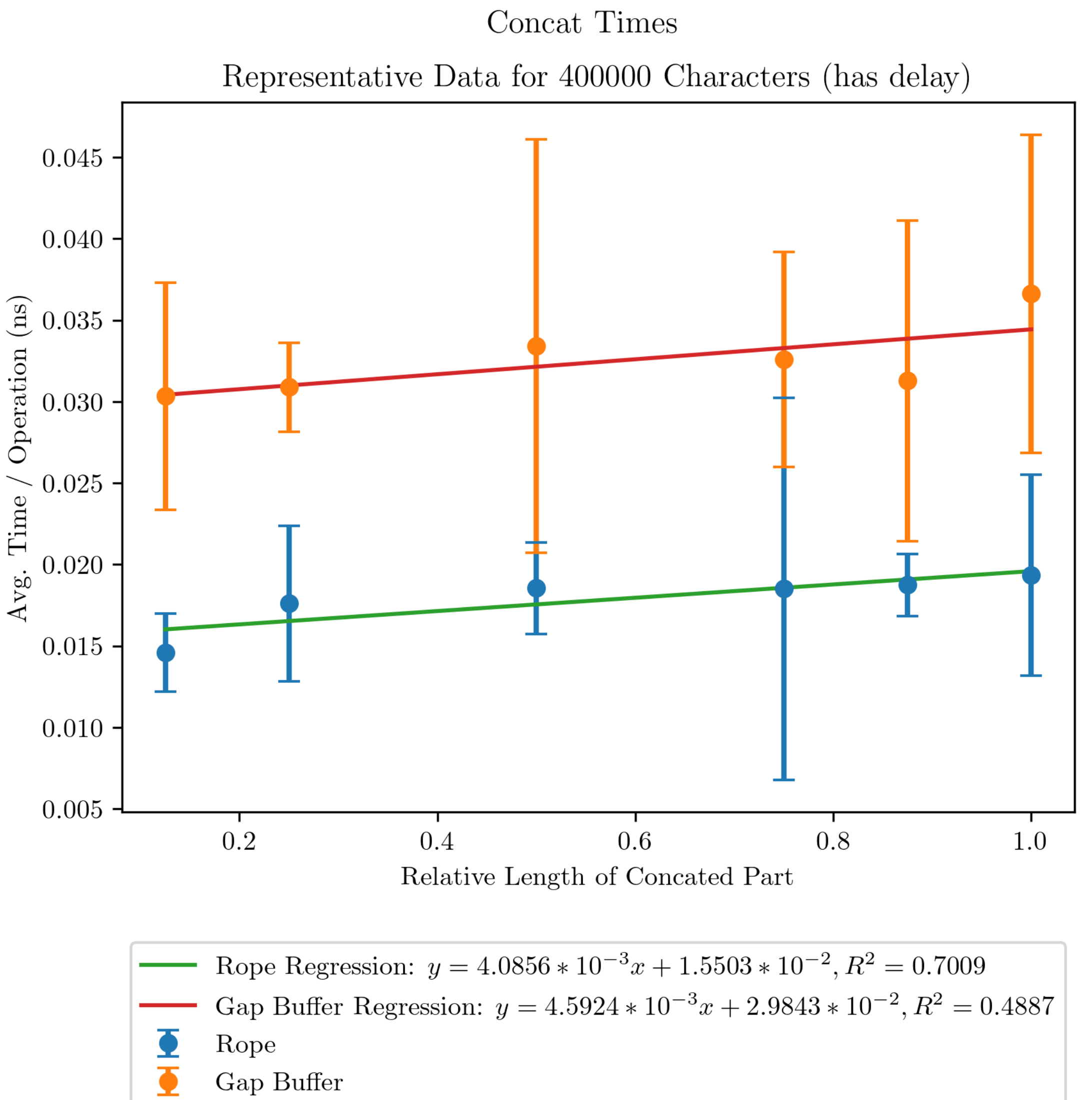

Figure 20: Representative insert times, varying relative position

In relative length tests, variance was far higher. Correlations were generally better, but are still insufficient to judge conformity to expectations.

Buffer Size	Rope Slopes	Gap Buffer Slopes
40000	-0.001036018	-0.000587539
80000	8.490847E-05	0.0004192678
120000	0.0003090441	0.0009599729
160000	-0.006917207	4.359322E-05
200000	-0.008217546	0.0006912814
240000	-0.005898549	0.0130817627
280000	-0.001377451	-0.004911729
320000	0.0021790373	0.0006795932
360000	0.0046366915	0.0012748475
400000	0.0040855593	0.0045924068

	Rope	Gap Buffer
Sample Size	10	10
Sample Mean	-0.001215153	0.0016243458
Sample STDEV	0.0044896802	0.0046424633
Hypothesised Mean	0	0
Test Statistic	-0.855885251	1.106445424
Degrees of Freedom	9	9
P-Values	0.4142664918	0.2972298255

Figure 21: Table of T-test results for concat

The p-values for relative length tests once again suggests that the slopes for other sample sizes are not significantly distinguishable from 0, so these operations most likely conformed to the expectation of constant time.

6 Conclusion and Further Investigation

6.1 Conclusion

Table 3 reviews the expected and actual outcomes of the tests, as previously outlined in Sections 5 and 3. There were some tests that showed their expected trend (by exceeding the R^2 threshold of 0.8), but many test results demonstrated too much variance to justify a conclusion.

Test	**Sub-test**	**Expected Behaviour**		**Matched Expectations?**	
		Rope	**Gap Buffer**	**Rope**	**Gap Buffer**
Index	Buffer Size	$O(\log n)$	$O(1)$	No, significant variance	Same conclusion as Rope
	Relative Position	$O(1)$	$O(k)$	Significant variance, but slopes across buffer sizes appear to be linear	Same conclusion as Rope
Load	Buffer Size	$O(n)$	$O(n)$	Yes	Yes

Report	Buffer Size	$O(n)$	$O(n)$	No, high variance	Yes
Split	Buffer Size	$O(\log n)$	$O(1)$	Yes	No, very strong $O(n)$ trend.
	Relative Position	$O(1)$	$O(k)$	Slopes across buffer sizes appear to be linear	Same as Rope, but higher variance
Insert	Buffer Size	$O(\log n)$	$O(1)$	Yes	Yes, but variance
Delete	Buffer Size	$O(\log n)$	$O(1)$	Yes	No, very strong $O(n)$ trend
Concat	Buffer Size	$O(1)$	$O(1)$	No, significant variance	Same conclusion as rope
	Relative Length	$O(1)$	$O(1)$	Significant variance, but slopes across buffer sizes appear to be linear	Same conclusion as rope

Table 3: Table of expected and actual results

While the time complexity of some rope operations was theoretically worse, in reality, the linear times produced by the gap buffer often started at a far greater initial value than the logarithmic times produced by the rope. As a result, the rope performed better for the sample sizes used. This is most obviously visible in the insert test (refer to Figure 17), where the initial value of the gap buffer's execution time is far greater than that of the rope, so the rope is more efficient even as n increases.

The gap buffer's unexpected linear performance in some tests is likely the result of an error in the implementation of the Index function, which was a part of all operations that exhibited this unexpected behaviour. The result was that constant position indexing times scaled with buffer size instead of remaining constant. A less likely cause is some sort of unexpected compiling / language feature.

The variance across all tests is likely due to a failure to consider system state during testing, which should be explored in future work. It is also possible that it is the result of excessively simple test sizes, whose triviality instead lead to benchmarking of the volatile CPU scheduler or other operating system features rather than the code itself. Generally, times are only a proportion of a normal CPU cycle, which also seems inaccurate. Additionally, trials had relatively few intermediate increments and small sample sizes. On many tests, while the logarithmic R^2 for rope operations was satisfactory, the trend in points was visibly more linear. This behaviour may also result from an unforseen implementation error, but a conclusion cannot be reached without additional data.

The tests undertaken in this paper provide a middling resolution to the initial research question. In conclusion, **the gap buffer and rope somewhat conformed to their expected time complexities, and the rope does usually perform better in a majority of tests, but the variance in the data taken makes the results of many tests uncertain.** Even though many of the uncertain tests appear to exhibit the expected relationship, suggesting that the performance characteristic is most likely similar, the poor R^2 values on these tests prevent a definitive conclusion. Across all conlcusive tests, the rope measured faster execution times than the gap buffer at end points, demonstrating its expected efficiency advantage.

6.2 Further Investigation

There are a number of further research directions and improvements that can be made. During testing, significantly more tests than included in this paper were run, some of which produced interesting results (which are included in Appendix A). These results would most likely be better explored in an investigation of only one of the data structures.

To highlight the performance of the two data structures in *practical* circumstances, a series of smaller but more involved tests that would better represent a user's typical inputs could be run.

In terms of improvements to the process for this paper, it would first be important to resolve any potential issues with the gap buffer's Index function described previously. The rope should also be checked over to prevent mistakes in implementation.

Additionally, beyond the resolution of deficiencies discussed under Section 4, the Go bundled memory profiler should be used on both data structures during re-development to optimize resource utilization, allowing for a demonstration of their more ideal behaviours.

Providing a more consistent testing environment should be explored. This can be in the form of compiler options, or the use of a simplified hardware and software platform with fewer background processes and variations.

In a related vein, while the tests appear to function properly, memory allocation data was not recorded even when benchmarking operations that should allocate, which requires further investigation. This may be the result of a platform deficiency, or misuse of the relevant language features. Additionally, while benchmarks *appeared* to have functioned properly, some sort of quality control for each trial should be implemented to guarantee successful operations, to prevent the potential compiling-out of tests.

Otherwise, more trials with more increments and greater input sizes, combined with more conclusive measures of variance and correlation, would provide a better guarantee of data reliability and stabilise results on tests with high variance, such as the report tests. Finally, while the visual indication of variance on graphs is still useful, finding a numerical expression of variance would be meaningful in measuring improvements to the experimental process and comparing variance across tests.

References

[1] C. A. Finseth, *The Craft of Text Editing*. St. Paul, 1991. [Online]. Available: %5Curl%7Bhttp://www.finseth.com/craft/%7D (visited on 10/18/2021).

[2] P. E. Black. (Dec. 2004). Complexity, [Online]. Available: %5Curl%7Bhttps://xlinux.nist.gov/dads/HTML/complexity.html%7D (visited on 10/18/2021).

[3] ——, (Sep. 2019). Big-o notation, [Online]. Available: %5Curl%7Bhttps://xlinux.nist.gov/dads/HTML/bigOnotation.html%7D (visited on 10/18/2021).

[4] A. Mohr, [Online]. Available: %5Curl%7Bhttp://www.austinmohr.com/Work_files/complexity.pdf%7D (visited on 10/18/2021).

[5] G. Emacs, *The buffer gap*, https://www.gnu.org/software/emacs/manual/html_node/elisp/Buffer-Gap.html, Accessed: 2021-02-08.

[6] M. Chu-Carroll. (Jan. 2009). Ropes: Twining together strings for editors, [Online]. Available: %5Curl%7Bhttps://scienceblogs.com/goodmath/2009/01/26/ropes-twining-together-strings%7D (visited on 10/08/2021).

[7] LazyHacker, Nov. 2017. [Online]. Available: %5Curl%7Bhttps://github.com/lazyhacker/gapbuffer/blob/master/gap_buffer.cpp%7D (visited on 10/18/2021).

[8] H. J. Boehm, R. Atkinson, and M. Plass, “Ropes: An alternative to strings,” *Software: Practice and Experience*, vol. 25, no. 12, Dec. 1995. [Online]. Available: %5Curl%7Bhttps://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.14.9450&rep=rep1&type=pdf%7D (visited on 10/18/2021).

[9] “Rope ¡t, alloc¿,” Silicon Graphics International, Tech. Rep., 1999. [Online]. Available: %5Curl%7Bhttp://martinbroadhurst.com/stl/Rope.html%7D (visited on 10/18/2021).

[10] M. Chu-Carroll. (Feb. 2009). Gap buffers, or, don’t get tied up with ropes? [Online]. Available: %5Curl%7Bhttps://scienceblogs.com/goodmath/2009/02/18/gap-buffers-or-why-bother-with-1%7D (visited on 10/18/2021).

[11] A. Bierce. Project Gutenberg, Oct. 2021. [Online]. Available: https://www.gutenberg.org/ebooks/66576 (visited on 10/18/2021).

[12] 2022. [Online]. Available: https://pkg.go.dev/testing.

[13] G. Adelson-Velskii and E. Landis, 1962. [Online]. Available: https://zhjwpku.com/assets/pdf/AED2-10-avl-paper.pdf.

A Additional Insert Tests

These are graphs for Insert tests that were not included within the paper.

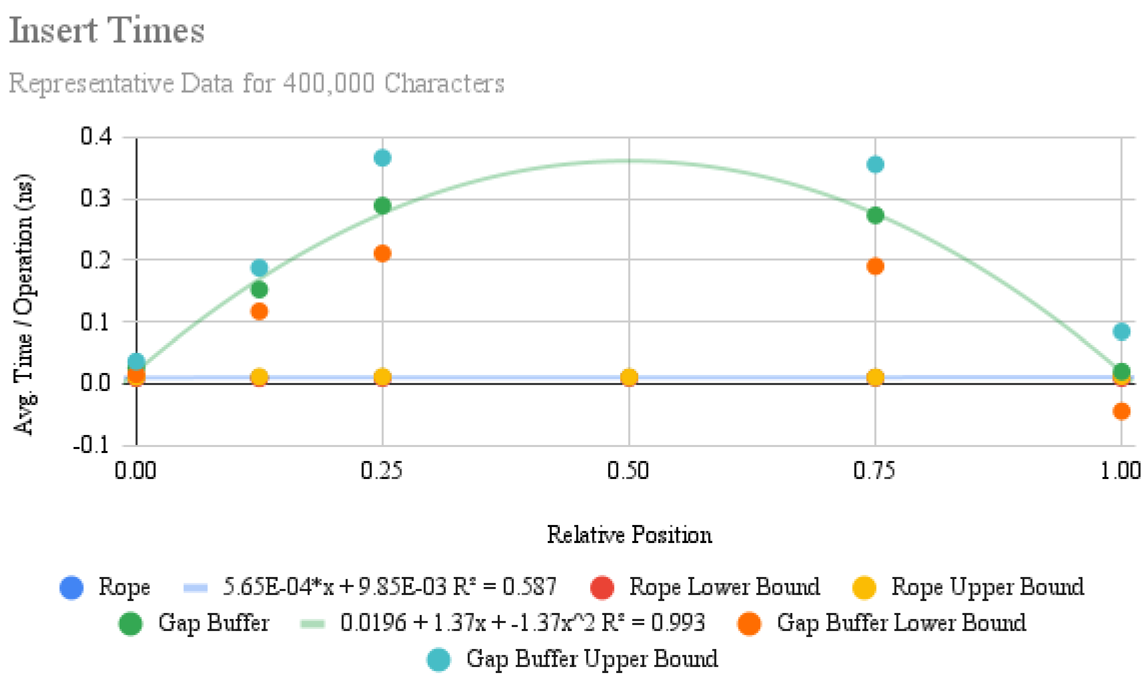

Figure 22: Additional Insert graph 1: varying relative position

This graph most visibly demonstrates the gap buffer index operation's "polynomial" behaviour. The middle value is missing because the test did not run properly.

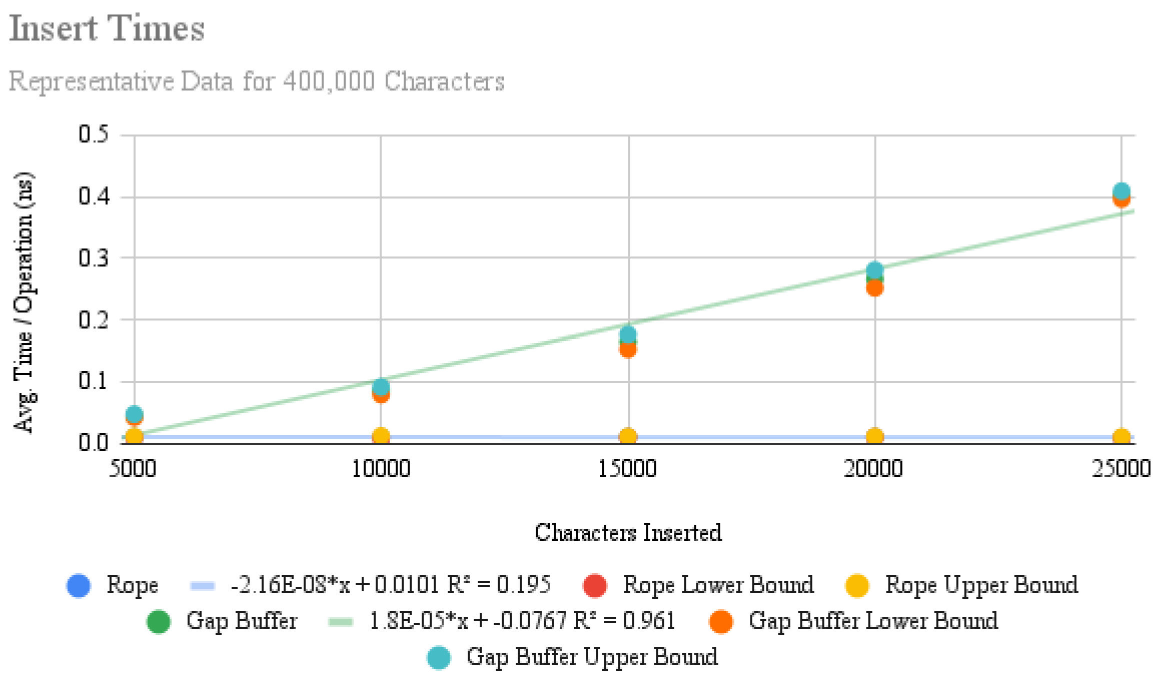

Figure 23: Additional Insert graph 2: varying number of characters inserted

This graph most visibly demonstrates the rope's superior performance, nearly approaching a constant time, and also demonstrates the gap buffer's expected linear performance.

B All Code

Intermediate data forms are not included. Test results were transferred into a text file, which was run through results−reader.go, which parsed them into .csv files. These files were then put into Google Sheets to calculate standard deviations and t-tests. Standard deviation data was copied to additional csvs which were run through “grapher.py”, with chart names in the first row. T-tests were ”prettified” and are in the PDF of the spreadsheet included after this section.

Listings

Listing 1: StorageType interface

```
package main


// Rune represents a character in Go
type StorageType interface {
        Report() ([]rune, error)                    // report entire buffer
        ReportRange(i, j int) ([]rune, error) // report segment of buffer from i
            ↪ ... j, inclusive
        Insert(i int, content []rune) error     // insert content at position i.
        Append(content []rune) error            // used during testing, ignore
        Concat(s StorageType) error             // self-explanatory
        Split(i int) (StorageType, error)       // splits at point i, returning a new
            ↪  buffer [i ... n]
```

```
        DeleteRange(i, j int) ([]rune, error) // deletes from [i, j)
        Load(contents []byte) error            // loads contents into buffer
        ToString() string                      // self-explanatory
        Index(i int) (rune, error)             //  returns character at position
}
```

Listing 2: Linked List implementation

```
package main

import (
        "errors"
)

// basic doubly and singly linked lists

type DoubleLink struct {
        Content rune
        Next    *DoubleLink
        Prior   *DoubleLink
}

func (dl *DoubleLink) TForward() *DoubleLink {
        return dl.Next
}

func (dl *DoubleLink) TReverse() *DoubleLink {
        return dl.Prior
}

func (dl *DoubleLink) DelinkL() {
        dl.Prior.Next = nil
        dl.Prior = nil
}
```

```
func (dl *DoubleLink) DelinkR() {
        dl.Next.Prior = nil
        dl.Next = nil
}

func (dl *DoubleLink) LinkL(d *DoubleLink) {
        // Links the new node to the left
        d.Next = dl
        dl.Prior = d
}

func (dl *DoubleLink) LinkR(d *DoubleLink) {
        // Links the new node to the right
        d.Prior = dl
        dl.Next = d
}

type DoublyLinkedList struct {
        Head   *DoubleLink
        End    *DoubleLink
        Length int
}

func (dll *DoublyLinkedList) getLen() int {
        return dll.Length
}

func (dll *DoublyLinkedList) TraverseTo(i int) (*DoubleLink, error) {
        if i > dll.Length {
                return nil, errors.New("Index exceeds list length")
        }
```

```
        if i == 0 {
                return dll.Head, nil
        } else if i == dll.Length-1 {
                return dll.End, nil
        }
        var ret *DoubleLink
        if i < dll.Length/2 {
                ret = dll.Head
                for ci := 0; ci < i; ci++ {
                        ret = ret.TForward()
                }
        } else if i >= dll.Length/2 {
                ret = dll.End
                for ci := dll.Length - 1; ci > i; ci-- {
                        ret = ret.TReverse()
                }
        }

        return ret, nil
}

func (dll *DoublyLinkedList) TraverseRangeF(i, j int) ([]*DoubleLink, error) {
        if j < i {
                return nil, errors.New("Bad indices")
        }
        initial, err := dll.TraverseTo(i)
        ret := []*DoubleLink{initial}
        if err != nil {
                return nil, err
        }
        for i := i; i < j; i++ {
                initial = initial.TForward()
```

```
            ret = append(ret, initial)
        }
        return ret, nil
}

func (dll *DoublyLinkedList) TraverseRangeR(i, j int) ([]*DoubleLink, error) {
        if i < j {
                return nil, errors.New("Bad indices")
        }
        initial, err := dll.TraverseTo(j)
        ret := []*DoubleLink{initial}
        if err != nil {
                return nil, err
        }
        for j := j; j > i; j-- {
                initial = initial.TReverse()
                ret = append(ret, initial)
        }
        return ret, nil
}

func (dll *DoublyLinkedList) Insert(i int, r rune) error {
        // inserts after index i (replaces current index i with new node with value
            ↪ r)
        if i > dll.Length || i < 0 {
                return errors.New("bad index")
        }
        nl := DoubleLink{
                Content: r,
                Prior:   nil,
                Next:    nil,
        }
```

```go
	if dll.Length == 0 {
		dll.Head = &nl
		dll.End = &nl
		dll.Length++
		return nil
	}

	if i == 0 {
		// case: start
		nl.Next = dll.Head
		dll.Head.Prior = &nl
		dll.Head = &nl
	} else if i == dll.Length {
		// case: end
		dll.End.LinkR(&nl)
		dll.End = dll.End.Next
	} else {
		p, _ := dll.TraverseTo(i - 1)
		a := p.Next
		p.Next = &nl
		a.Prior = &nl
		nl.Next = a
		nl.Prior = p
	}

	dll.Length++
	return nil
}

func (dll *DoublyLinkedList) Remove(i int) (rune, error) {
	removed, err := dll.TraverseTo(i)
	if err != nil {
```

```
            return -1, errors.New("Bad traverse during removal")
        }
        removed.Prior.Next = removed.Next
        removed.Next.Prior = removed.Prior
        dll.Length--
        return removed.Content, nil

}

func (dll *DoublyLinkedList) Split(i int) (*DoublyLinkedList, error) {
        // splits the linked list into segments 0...i, i+1 ... end. returns the
            ↪ portion i+1 ... end.
        splitEndNode, err := dll.TraverseTo(i)
        if err != nil {
                return nil, errors.New("error in traversing to index in split")
        }
        splitStartNode := splitEndNode.Next

        splitEndNode.DelinkR()
        right := &DoublyLinkedList{
                Head:   splitStartNode,
                End:    dll.End,
                Length: dll.Length - i - 1,
        }
        dll.End = splitEndNode
        dll.Length = i + 1

        return right, nil

}

/*
```

```
func (dll *DoublyLinkedList) Swap(i int, j int) error {
        first, err := dll.TraverseTo(i)
        if err != nil {
                return errors.New("bad traverse in swap")
        }
        second, err := dll.TraverseTo(j)
        if err != nil {
                return errors.New("bad traverse in swap")
        }

        // TODO: handle special cases of start/end

        firstNext, firstPrior := first.Next, first.Prior



        first.Prior, first.Next = second.Prior, second.Next

        second.Prior, second.Next = firstPrior, firstNext
        if (i == 0){
                dll.Head = second
        } else if (j == 0){
                dll.Head = first
        } else if (i == dll.Length - 1){
                dll.End = second
        } else if (j == dll.Length - 1){
                dll.End = first
        }

        return nil
}
*/
func (dll *DoublyLinkedList) ToString() string {
```

```
        ret := ""
        fn := dll.Head
        for fn != nil {
                ret += string(fn.Content)
                fn = fn.Next
        }
        return ret
}
```

Listing 3: Linked List tests

```
package main

import (
        "fmt"
        "testing"
)

func TestLInsert(t *testing.T) {
        t.Log("Traverse To test")

        ll := DoublyLinkedList{Head: nil, End: nil, Length: 0}
        ll.Insert(0, 'h')
        ll.Insert(1, 'a')

        ll.Insert(1, 'b')
        res, _ := ll.TraverseTo(1)
        fmt.Println(string(res.Content))
        fmt.Println(ll.ToString())
        ll.Insert(0, 'c')
        fmt.Println(ll.ToString())
        //ll.Swap(1, 2)
        //fmt.Println(ll.ToString())
```

```
}
```

Listing 4: Gap Buffer implementation

```go
package main

import (
	"errors"
)

type GapBuffer struct {
	Content     *DoublyLinkedList
	GapStart    *DoubleLink
	GapEnd      *DoubleLink
	GapStartIdx int
	GapLen      int
}

func (g *GapBuffer) isEmpty() error {
	if g.Length() == 0 {
		return errors.New("list is empty")
	}
	return nil
}

func (g *GapBuffer) Length() int {
	return g.Content.getLen() - g.GapLen
}

func (g *GapBuffer) checkIdxStrict(i ...int) error {
	err := g.isEmpty()
	if err != nil {
		return err
	}
```

```go
	prev := i[0]
	for _, el := range i {
		if el < prev {
			return errors.New("incorrect index order")
		}
		if (el < 0) || (g.Length() <= el) {
			return errors.New("Index Out of bounds")
		}
		prev = el
	}
	return nil
}

func (g *GapBuffer) checkIdxLoose(i ...int) error {
	// for anything that allows something to be put at the very end
	err := g.isEmpty()
	if err != nil {
		return err
	}
	prev := i[0]
	for _, el := range i {
		if el < prev {
			return errors.New("incorrect index order")
		}
		if (el < 0) || (g.Length() < el) {
			return errors.New("Index Out of bounds")
		}
		prev = el
	}
	return nil
}
```

```go
func (g *GapBuffer) getNode(i int) (*DoubleLink, error) {
	err := g.checkIdxStrict(i)
	if err != nil {
		return nil, err
	}
	if i < g.GapStartIdx {
		res, err := g.Content.TraverseTo(i)
		if err != nil {
			return nil, err
		}
		return res, nil
	} else {
		res, err := g.Content.TraverseTo(i + g.GapLen)
		if err != nil {
			return nil, err
		}
		return res, nil
	}
}

func (g *GapBuffer) Index(i int) (rune, error) {
	ret, err := g.getNode(i)
	return ret.Content, err
}

func (g *GapBuffer) Report() ([]rune, error) {
	// just special case of ReportRange where i = 0 and j = length - 1
	return g.ReportRange(0, g.Length()-1)
}

func (g *GapBuffer) ReportRange(i, j int) ([]rune, error) {
	err := g.checkIdxStrict(i, j)
```

```go
	if err != nil {
		return nil, err
	}
	if i == j {
		ret, err := g.ReportCharacter(i)
		return []rune{ret}, err
	}
	ret := make([]rune, 0)
	if g.GapStartIdx == 0 {
		i += g.GapLen
	}
	cn, err := g.Content.TraverseTo(i)
	if err != nil {
		return nil, err
	}
	idx := i
	if j >= g.GapStartIdx {
		j += g.GapLen
	}
	for idx <= j {
		if idx >= g.GapStartIdx+g.GapLen || idx < g.GapStartIdx {
			ret = append(ret, cn.Content)
		}
		idx++
		cn = cn.Next
	}
	return ret, nil

}

func (g *GapBuffer) ReportCharacter(i int) (rune, error) {
```

```go
	err := g.checkIdxStrict(i)
	if err != nil {
		return -1, err
	}

	d, err := g.getNode(i)
	if err != nil {
		return -1, err
	}
	return d.Content, nil
}

func (g *GapBuffer) insertAtGap(content []rune) error {

	if len(content) >= g.GapLen {
		es := len(content) - g.GapLen
		g.expandGap(es + 2)
	}
	cn := g.GapStart
	for _, r := range content {
		g.GapStart = g.GapStart.Next
		cn.Content = r
		if cn.Next == nil {
			return errors.New("bad next link while inserting in gap")
		}
		cn = cn.Next
		g.GapLen--
		g.GapStartIdx++
	}
	return nil

}
```

```go
func (g *GapBuffer) expandGap(size int) {
	for i := 0; i < size; i++ {
		g.Content.Insert(g.GapStartIdx+g.GapLen, -1) // todo: this is
		    ↪ probably the part you want to tune for indices
		g.GapEnd = g.GapEnd.Next
		g.GapLen++
	}
}

func (g *GapBuffer) Insert(i int, content []rune) error {
	err := g.moveGap(i)
	if err != nil {
		return err
	}
	g.insertAtGap(content)
	return nil
}

func (g *GapBuffer) Append(content []rune) error {

	err := g.moveGap(g.Length())
	if err != nil {
		return err
	}
	g.insertAtGap(content)
	return nil
}

func (g *GapBuffer) moveGap(i int) error {
	if i == g.GapStartIdx {
		return nil
```

```
}
err := g.checkIdxLoose(i)
if err != nil {
        return err
}
var prevIdx, nxt *DoubleLink

if i != 0 && i != g.Length() {
        prevIdx, err = g.getNode(i - 1)
        if err != nil {
                return err
        }
        nxt = prevIdx.Next
}

if g.GapStartIdx == 0 {
        g.Content.Head = g.GapEnd.Next
        g.GapEnd.DelinkR()
} else if g.GapStartIdx+g.GapLen == g.Content.Length {
        g.Content.End = g.GapStart.Prior
        g.GapStart.DelinkL()
} else {
        bf := g.GapStart.Prior
        af := g.GapEnd.Next
        g.GapStart.DelinkL()
        g.GapEnd.DelinkR()
        bf.LinkR(af)
}

if i == 0 {
        g.GapEnd.LinkR(g.Content.Head)
        g.Content.Head = g.GapStart
```

```
	} else if i == g.Length() {
		g.GapStart.LinkL(g.Content.End)
		g.Content.End = g.GapEnd
	} else {
		prevIdx.DelinkR()
		g.GapStart.LinkL(prevIdx)
		g.GapEnd.LinkR(nxt)
	}
	g.GapStartIdx = i
	return nil
}

func (g *GapBuffer) Replace(i int, content []rune) error {
	err := g.checkIdxStrict(i)
	if err != nil {
		return err
	}

	start, err := g.getNode(i)
	if err != nil {
		return err
	}
	for _, r := range content {
		if start == g.GapStart {
			for start != g.GapEnd {
				start = start.Next
			}
			start = start.Next
		}
		start.Content = r
		start = start.Next
		if start == nil {
```

```
                return errors.New("overflow list")
            }

        }
        return nil
}

func (g *GapBuffer) compressGap() error {
        if g.GapLen == 0 {
                return nil
        }
        if g.GapStartIdx == 0 {
                g.Content.Head = g.GapEnd.Next
                g.Content.Head.DelinkL()
        } else if g.GapStartIdx+g.GapLen == g.Content.Length {
                g.Content.End = g.GapStart.Prior
                g.Content.End.DelinkR()
        } else {
                bf := g.GapStart.Prior
                af := g.GapEnd.Next
                g.GapStart.DelinkL()
                g.GapEnd.DelinkR()
                bf.LinkR(af)
        }
        g.Content.Length -= g.GapLen
        g.GapStart, g.GapEnd = nil, nil
        g.GapStartIdx, g.GapLen = 0, 0
        return nil
}

func (g *GapBuffer) Split(i int) (StorageType, error) {
        if i == 0 {
```

```
		return nil, nil
	}
	err := g.checkIdxStrict(i)
	if err != nil || i == 0 {
		return nil, errors.New("bad split point")
	}

	err = g.compressGap()
	if err != nil {
		return nil, err
	}
	s, err := g.Content.Split(i - 1)
	if err != nil {
		return nil, err
	}
	nb := &GapBuffer{
		Content:     s,
		GapStart:    s.Head,
		GapEnd:      s.Head,
		GapStartIdx: 0,
		GapLen:      0,
	}
	err = nb.makeGap()
	if err != nil {
		return nil, err
	}
	err = g.makeGap()
	if err != nil {
		return nil, err
	}
	return nb, nil
}
```

```go
func (g *GapBuffer) DeleteRange(i, j int) ([]rune, error) {
	err := g.checkIdxStrict(i, j)
	if err != nil {
		return nil, err
	}

	ret := make([]rune, 0)
	inCt := j - i
	g.moveGap(i)
	for i := 0; i < inCt; i++ { // todo: tune
		g.GapEnd = g.GapEnd.Next
		ret = append(ret, g.GapEnd.Content)
		g.GapEnd.Content = -1
		g.GapLen++
	}
	return ret, nil
}

func (g *GapBuffer) Load(contents []byte) error {
	g.Content = &DoublyLinkedList{
		Length: 0,
	}
	g.GapLen = 0
	for _, b := range contents {
		err := g.Content.Insert(g.Length(), rune(b))
		if err != nil {
			return err
		}
	}
	err := g.makeGap()
	return err
```

```
}

func (g *GapBuffer) makeGap() error {
	if g.GapLen > 0 {
		return nil
	}
	// default: gap at very start
	err := g.Content.Insert(0, -1)
	if err != nil {
		return err
	}
	g.GapLen = 1
	g.GapStart = g.Content.Head
	g.GapEnd = g.Content.Head
	g.GapStartIdx = 0
	g.expandGap(9)
	return nil
}

func (g *GapBuffer) ToString() string {
	ret := ""
	head := g.Content.Head
	ct := 0
	for head != nil {
		if head.Content == -1 { //(ct >= g.GapStartIdx) && (ct <= g.
		    ↪ GapStartIdx+g.GapLen)
			ret += "_"
		} else {
			ret += string(head.Content)
		}
		ct++
		head = head.Next
```

```
	}
	return ret
}

func (gb *GapBuffer) Concat(s StorageType) error {
	contents, ok := s.(*GapBuffer)
	if ok {
		gb.moveGap(0)
		contents.compressGap()
		l := contents.Length()
		gb.Content.End.LinkR(contents.Content.Head)
		gb.Content.Length += l
	}
	return nil
}

func mkGapBuf() *GapBuffer {
	return &GapBuffer{
		Content: &DoublyLinkedList{
			Head:   nil,
			End:    nil,
			Length: 0,
		},
		GapStart:    nil,
		GapEnd:      nil,
		GapStartIdx: 0,
		GapLen:      0,
	}
}
```

Listing 5: Rope Node implementation

```
package main
```

```go
type AVLNode struct {
	Value  []rune
	Height int
	Weight int

	L *AVLNode
	R *AVLNode
	U *AVLNode
}

func (a *AVLNode) IsLeaf() bool {
	if a == nil {
		return false
	}
	return (a.L == nil) && (a.R == nil)
}

func (a *AVLNode) isLeaf() bool {
	return (a.L == nil) && (a.R == nil)
}

func (a *AVLNode) getHeight() int {
	if a != nil {
		return a.Height
	}
	return 0
}

func (a *AVLNode) updateHeight() {
	if a.isLeaf() {
		a.Height = 0
		return
```

```
        }
        a.Height = 1 + max(a.L.getHeight(), a.R.getHeight())
}

func (a *AVLNode) linkLeft(nl *AVLNode) *AVLNode {
        // normally, create new left link, return the current node
        if nl != nil {
                a.L = nl
                nl.U = a
                return a
        }
        // otherwise, compact
        return a.R
}

func (a *AVLNode) linkRight(nr *AVLNode) *AVLNode {
        // mirror of above
        if nr != nil {
                a.R = nr
                nr.U = a
                return a
        }
        return a.L
}

func (a *AVLNode) delinkLeft() *AVLNode {
        // delinks and returns old left
        tmp := a.L
        a.L.U = nil
        a.L = nil
        return tmp
}
```

```go
func (a *AVLNode) delinkRight() *AVLNode {
	// delinks and returns old right
	tmp := a.R
	a.R.U = nil
	a.R = nil
	return tmp
}

func (a *AVLNode) sumTree() int {
	// returns the sum of leaves on this subtree
	if a == nil {
		return 0
	}
	if a.isLeaf() {
		return a.Weight
	}
	return a.Weight + a.R.sumTree()
}

func (a *AVLNode) updateWeight() {
	a.Weight = a.L.sumTree()
}

func (a *AVLNode) rotateLeft() *AVLNode {
	/*
		``a`` is the root of the subtree to be rotated left

		Returns the new root of the rotated subtree
	*/
	if a == nil {
		return a
```

```
        }
        z := a.R

        // check that rotation is actually required: disable during testing

        if (z == nil) || (z.L == nil) || (z.R == nil) {
                return a
        }

        inner := z.delinkLeft() // get inner child of right subtree
        // then, let x either be the newly rotated left a, or the compacted version
            ↪ of a if inner is nil
        x := a.linkRight(inner)
        // link x to z's left
        z.linkLeft(x)
        // clear z's U
        z.U = nil

        // update heights
        x.updateHeight()
        z.updateHeight()

        // update weights: since only z's weight is changing, this is
        // the only necessary change
        z.updateWeight()

        // return necessary value
        return z
}

func (a *AVLNode) rotateRight() *AVLNode {
        /*
```

```
        ``a`` is the root of the subtree to be rotated right

        returns the new root of the rotated subtree
*/
if a == nil {
        return a
}
z := a.L

// check that rotation is actually required: disable during testing
if (z == nil) || (z.L == nil) || (z.R == nil) {
        return a
}

inner := z.delinkRight() // get inner child of left subtree
// then, let x either be the newly rotated right a, or the compacted
    ↪ version of a if inner is nil
x := a.linkLeft(inner)
// link x to z's right
z.linkRight(x)
// clear z's U
z.U = nil

// update heights
x.updateHeight()
z.updateHeight()

// update weights
x.updateWeight()
z.updateWeight()
// return necessary value
return z
```

```go
}

func (a *AVLNode) balance() *AVLNode {
	/*
		``a`` is the root of the subtree to be balanced
		returns the new root of the balanced subtree
	*/
	if a == nil {
		return nil
	}
	balanceFactor := a.L.getHeight() - a.R.getHeight()
	if balanceFactor <= -2 {
		if a.R.L.getHeight() > a.R.R.getHeight() {
			a.R = a.R.rotateRight()
			// prolly gonna have a delightful number of ptr errors here
		}
		return a.rotateLeft()
	} else if balanceFactor >= 2 {
		if a.L.R.getHeight() > a.L.L.getHeight() {
			a.L = a.L.rotateLeft()
		}
		return a.rotateRight()
	}
	return a
}

func (a *AVLNode) ToRune() []rune {
	ret := make([]rune, 0)
	a.ApplyInorder(func(n *AVLNode) {
		ret = append(ret, n.Value...)
	})
	return ret
```

```go
}

func (a *AVLNode) ApplyInorder(f func(n *AVLNode)) {
	if a == nil {
		return
	}
	a.L.ApplyInorder(f)
	if a.Value != nil {
		f(a)
	}
	a.R.ApplyInorder(f)
}

func (a *AVLNode) IndexNode(i int) (*AVLNode, int) {
	// finds and returns node that contains char at index i, returning the
	    ↪ number of characters by which to advance in
	if a == nil {
		return nil, -1
	}
	if a.isLeaf() && (0 <= i) && (i < len(a.Value)) {
		return a, i
	}
	if a.Weight <= i {
		// go right
		return a.R.IndexNode(i - a.Weight)
	}
	if a.Weight > i {
		// go left
		return a.L.IndexNode(i)
	}
	return nil, -1
}
```

```go
func max(a, b int) int {
	if a < b {
		return b
	}
	return a
}
```

Listing 6: Rope implementation

```go
package main

import (
	"errors"
)

const (
	Segsize = 10
)

func JoinRight(L, R *AVLNode) *AVLNode {
	// Joins L and R together with right bias
	ll, lr := L.L, L.R
	if lr.getHeight() <= (R.getHeight() + 1) {
		inter := mkNode(nil, lr, R)
		if inter.getHeight() <= (ll.getHeight() + 1) {
			return mkNode(nil, ll, inter)
		}
		return mkNode(nil, ll, inter.rotateRight()).rotateLeft()
	}
	inter := JoinRight(lr, R)
	if inter.getHeight() <= (ll.getHeight() + 1) {
		return mkNode(nil, ll, inter)
	}
```

```
        return mkNode(nil, ll, inter).rotateLeft()
}

func JoinLeft(L, R *AVLNode) *AVLNode {
        // Mirror of above
        rl, rr := R.L, R.R
        if rl.getHeight() <= (L.getHeight() + 1) {
                inter := mkNode(nil, L, rl)
                if inter.getHeight() <= (rr.getHeight() + 1) {
                        return mkNode(nil, inter, rr)
                }
                return mkNode(nil, inter.rotateLeft(), rr).rotateRight()
        }
        inter := JoinLeft(L, rl)
        if inter.getHeight() <= (rr.getHeight() + 1) {
                return mkNode(nil, inter, rr)
        }
        return mkNode(nil, inter, rr).rotateRight()
}

func Join(L, R *AVLNode) *AVLNode {
        if L.getHeight() > (R.getHeight() + 1) {
                return JoinRight(L, R)
        } else if R.getHeight() > (L.getHeight() + 1) {
                return JoinLeft(L, R)
        }
        return mkNode(nil, L, R)
}

func Split(ar *AVLNode, weight int) (*AVLNode, *AVLNode) {
        if ar == nil {
                return nil, nil
```

```
	}
	if ar.isLeaf() {

		return mkLeaf(ar.Value[:weight]), mkLeaf(ar.Value[weight:])
	}
	L, R := ar.L, ar.R
	if ar.Weight == weight {
		return L, R
	}
	if weight < ar.Weight {
		nL, nR := Split(L, weight)
		return nL, Join(nR, R)
	}
	// otherwise, weight > ar.Weight
	nL, nR := Split(R, weight-ar.Weight)
	return Join(L, nL), nR
}

func TraverseRange(start *AVLNode, startPos, len int) []rune {
	/*
		Returns ``len`` characters starting from startPos, inclusive
	*/
	ret := start.Value[startPos:]
	// if the starting point isn't a leaf, don't bother
	if !(start.isLeaf()) {
		return nil
	}
	return ret

}

func mkNode(val []rune, L, R *AVLNode) *AVLNode {
```

```go
	ret := &AVLNode{
		Value: val,
	}
	ret = ret.linkLeft(L)
	ret = ret.linkRight(R)

	ret.updateHeight()
	ret.updateWeight()
	return ret
}

func mkLeaf(val []rune) *AVLNode {
	return &AVLNode{
		Value:  val,
		Height: 0,
		Weight: len(val),
	}
}

type AVLRope struct {
	Head *AVLNode
}

func mkRope() *AVLRope {
	return &AVLRope{}
}

func (ar *AVLRope) Report() ([]rune, error) {
	return ar.Head.ToRune(), nil
}

func (ar *AVLRope) IndexNode(i int) (*AVLNode, int, error) {
```

```go
        node, num := ar.Head.IndexNode(i)
        return node, num, nil
}

func (ar *AVLRope) Index(i int) (rune, error) {
        ret, idx, err := ar.IndexNode(i)
        return ret.Value[idx], err
}
func (ar *AVLRope) ReportRange(i, j int) ([]rune, error) {
        ret := make([]rune, 0)
        // get start point
        from, ct, err := ar.IndexNode(i)
        if err != nil {
                return nil, errors.New("bad initial index")
        }
        // get len to be reported
        reportLen := j - i + 1 // remove this +1 to make exclusive
        if len(from.Value)-ct < reportLen {
                ret = append(ret, from.Value[ct:]...)
                reportLen -= len(ret)
        } else {
                ret = append(ret, from.Value[ct:ct+reportLen]...)
                return ret, nil
        }

        // then, while the length to be read has yet to be reached:
        curr := from.U
        for reportLen > 0 {
                // if came from right of curr, continue ascending
                if from == curr.R {
                        from = curr
                        curr = curr.U
```

```go
		} else if from == curr.L {
			// if came from left of curr, go down to right
			from = curr
			curr = curr.R
		} else if from == curr.U {
			// otherwise, if coming from higher, go down to left
			from = curr
			curr = curr.L
		}

		if curr.isLeaf() {
			// insert as much data as possible
			if len(curr.Value) < reportLen {
				ret = append(ret, curr.Value...)
				reportLen -= len(curr.Value)
			} else {
				ret = append(ret, curr.Value[:reportLen]...)
				return ret, nil
			}
			// begin backtrack
			from = curr
			curr = curr.U
		}

	}
	return ret, nil

}

func (ar *AVLRope) Insert(i int, content []rune) error {
	new := mkRope()
	new.LoadFromRune(content)
```

```go
        l, r := Split(ar.Head, i)
        l = Join(l, new.Head)
        ar.Head = Join(l, r)
        return nil
}

func (ar *AVLRope) Append(content []rune) error {
        // since this isn't actually tested, doesn't need to be efficient
        new := mkRope()
        new.LoadFromRune(content)
        ar.Head = Join(ar.Head, new.Head)
        ar.Head.updateWeight()
        return nil
}

func (ar *AVLRope) Split(i int) (StorageType, error) {
        l, r := Split(ar.Head, i)
        ar.Head = l
        return &AVLRope{Head: r}, nil
}

func (ar *AVLRope) DeleteRange(i, j int) ([]rune, error) {
        l1, r := Split(ar.Head, j)
        l2, m := Split(l1, i)
        ar.Head = Join(l2, r)
        return m.ToRune(), nil
}

func RoundUpDiv(num, dem int) int {
        if num%dem == 0 {
                return num / dem
        } else {
```

```
            return (num / dem) + 1
        }
}

func (ar *AVLRope) LoadFromRune(contents []rune) {
        // step 1: create the necessary leaves

        leaves := make([]*AVLNode, RoundUpDiv(len(contents), Segsize))
        remLen := len(contents)
        var data []rune
        for i := range leaves {
                if remLen/Segsize >= 1 {
                        data = make([]rune, Segsize)
                        remLen -= Segsize
                } else {
                        data = make([]rune, remLen)
                }

                for r := 0; r < Segsize; r++ {
                        idx := (i * Segsize) + r
                        if idx >= len(contents) {
                                break
                        }
                        data[r] = contents[idx]
                }
                leaves[i] = mkLeaf(data)
        }
        // step 2: recursively concat until something resembling balance has been
            ↪ achieved
        for len(leaves) > 1 {
                half := len(leaves) / 2
                if len(leaves)%2 == 0 {
```

```
                for r := 0; r < half; r++ {
                        leaves[r] = mkNode(nil, leaves[2*r], leaves[(2*r)
                            ↪ +1])
                }
        } else {
                for r := 0; r < half; r++ {
                        leaves[r] = mkNode(nil, leaves[2*r], leaves[(2*r)
                            ↪ +1])
                }
                leaves[half-1] = mkNode(nil, leaves[half-1], leaves[len(
                    ↪ leaves)-1])
        }
        leaves = leaves[:half]
    }
    if len(leaves) != 1 {
        panic("bad construction")
    }
    ar.Head = leaves[0]
}

func (ar *AVLRope) Load(contents []byte) error {

    // step 1: create the necessary leaves

    leaves := make([]*AVLNode, RoundUpDiv(len(contents), Segsize))
    remLen := len(contents)
    var data []rune
    for i := range leaves {

        if remLen/Segsize >= 1 {
            data = make([]rune, Segsize) // todo: this isn't very good
                ↪ for allocs, but oh well
```

```
            remLen -= Segsize
        } else {
            data = make([]rune, remLen)
        }

        for r := 0; r < Segsize; r++ {
            idx := (i * Segsize) + r
            if idx >= len(contents) {
                break
            }
            data[r] = rune(contents[idx])

        }
        leaves[i] = mkLeaf(data)
}
// step 2: recursively concat until something resembling balance has been
    ↪ achieved
// TODO: check whether current or previous ends up having more allocs: in
    ↪ case slices are being weird
//var a, b *AVLNode
for len(leaves) > 1 {
        /*
            fmt.Println("NEW CYCLE")
            for _, c := range leaves {
                PrintInline(c)
            }
            fmt.Println()

            a, b, leaves = leaves[0], leaves[1], leaves[2:]
            if b.getHeight() != a.getHeight() {
                fmt.Println("yes")
                // case where odd number of leaves in queue, so
```

```
                ↪ this must be concatted to the end
              leaves[len(leaves)-1] = mkNode(nil, leaves[len(
                ↪ leaves)-1], a)
              leaves = append([]*AVLNode{b}, leaves...)
              //a = b
              //b, leaves = leaves[0], leaves[1:]
          } else {
              leaves = append(leaves, mkNode(nil, a, b))
          }

          for _, c := range leaves {
              PrintInline(c)
          }
          fmt.Println()
      */

      half := len(leaves) / 2
      if len(leaves)%2 == 0 {
          for r := 0; r < half; r++ {
              leaves[r] = mkNode(nil, leaves[2*r], leaves[(2*r)
                ↪ +1])
          }
      } else {
          for r := 0; r < half; r++ {
              leaves[r] = mkNode(nil, leaves[2*r], leaves[(2*r)
                ↪ +1])
          }
          leaves[half-1] = mkNode(nil, leaves[half-1], leaves[len(
            ↪ leaves)-1])
      }
      leaves = leaves[:half]
}
```

```
        if len(leaves) != 1 {
                panic("bad construction")
        }
        ar.Head = leaves[0]
        return nil
}

func (ar *AVLRope) ToString() string {
        ret := ""
        ar.Head.ApplyInorder(func(n *AVLNode) { ret += (string(n.Value)) })
        return ret
}

func (ar *AVLRope) Concat(s StorageType) error {
        contents, ok := s.(*AVLRope)
        if ok {
                ar.Head = Join(ar.Head, contents.Head)
        }
        return nil
}
```

Listing 7: Unit tests of both buffers

```
package main

import (
        "fmt"
        "io"
        "os"
        "testing"
)

func GetContent(filename string, count int) []byte {
        file, err := os.Open(filename)
```

```go
	if err != nil {
		panic(err)
	}
	defer file.Close()

	ret := make([]byte, count)
	_, err = io.ReadFull(file, ret)
	if err != nil {
		panic(err)
	}
	return ret
}

type BufferTest struct {
	Start  []int // start also serves as the index in single-index operations
	End    []int // if this is -1, represents end
	Length int   // for operations that need some input, this will be used for
	    ↪ additional loads
}

func TestLoad(t *testing.T) {
	buffers := make([]StorageType, 2)
	buffers[0] = mkRope()
	buffers[1] = mkGapBuf()

	testCases := []BufferTest{
		{Length: 100},
		{Length: 200},
		{Length: 500},
		{Length: 1000},
	}
	for _, tc := range testCases {
```

```
        content := GetContent("testing.txt", tc.Length)
        for _, b := range buffers {
            t.Run(fmt.Sprintf("%T_load", b), func(t *testing.T) {
                t.Log(fmt.Sprintf("case: %d", tc.Length))
                b.Load(content)
                if b.ToString() != string(content) {
                    t.Fatal("improperly loaded string")

                }
            })
        }
    }
}

func TestInsert(t *testing.T) {
    buffers := make([]StorageType, 2)
    buffers[0] = mkRope()
    buffers[1] = mkGapBuf()
    testCases := []BufferTest{
        {Start: []int{0, 25, 50, -1}, Length: 100},
        {Start: []int{0, 50, 100, -1}, Length: 200},
        {Start: []int{0, 125, 250, -1}, Length: 500},
        {Start: []int{0, 250, 500, -1}, Length: 1000},
    }

    for _, tc := range testCases {
        content := GetContent("testing.txt", tc.Length)
        app := make([]rune, len(content))
        for i, bt := range content {
            app[i] = rune(bt)
        }
```

```go
//comparison := string(app)
//comparison = comparison + comparison

for _, b := range buffers {
        for sc_idx := range tc.Start {
                b.Load(content)
                point := tc.Start[sc_idx]
                if point == -1 {
                        point = tc.Length - 1
                }
                t.Run(fmt.Sprintf("%T insert", b), func(t *testing.
                    ↪ T) {
                        t.Log(fmt.Sprintf("case: %d", point))

                        comparison := string(append(app[:point],
                            ↪ append(app, app[point:]...)...))

                        err := b.Insert(point, app)
                        if err != nil {
                                t.Fatal(err)
                        }
                        res, err := b.Report()
                        if err != nil {
                                t.Fatal(err)
                        }
                        if string(res) != comparison {
                                t.Fatal(fmt.Sprintf("improperly
                                    ↪ appended \n expected: %s\n
                                    ↪ got: %s \n", comparison,
                                    ↪ string(res)))

                        }
```

```
                    })
                }

            }
        }
}

func TestReport(t *testing.T) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := []BufferTest{
                {Length: 100},
                {Length: 200},
                {Length: 500},
                {Length: 1000},
        }
        for _, tc := range testCases {
                content := GetContent("testing.txt", tc.Length)
                for _, b := range buffers {
                        b.Load(content)
                        t.Run(fmt.Sprintf("%T report", b), func(t *testing.T) {
                                t.Log(fmt.Sprintf("case: %d", tc.Length))
                                res, err := b.Report()
                                if err != nil {
                                        t.Fatal(err)
                                }
                                if string(res) != string(content) {
                                        t.Fatal("improperly reported contents")

                                }
```

```
                })
            }
        }
}

func TestReportRange(t *testing.T) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := []BufferTest{
                {Start: []int{0, 0, 50, 10, 30}, End: []int{-1, 10, -1, 20, 50},
                    ↪ Length: 100},
                {Start: []int{0, 0, 50, 10, 30}, End: []int{-1, 10, -1, 20, 50},
                    ↪ Length: 200},
                {Start: []int{0, 0, 50, 10, 30}, End: []int{-1, 10, -1, 20, 50},
                    ↪ Length: 500},
                {Start: []int{0, 0, 50, 10, 30}, End: []int{-1, 10, -1, 20, 50},
                    ↪ Length: 1000},
        }
        for _, tc := range testCases {
                content := GetContent("testing.txt", tc.Length)
                for _, b := range buffers {
                        b.Load(content)
                        for sc_idx := range tc.Start {
                                t.Run(fmt.Sprintf("%T_report_range", b), func(t *
                                    ↪ testing.T) {
                                        start, end := tc.Start[sc_idx], tc.End[
                                            ↪ sc_idx]
                                        if end == -1 {
                                                end = len(content) - 1
                                        }
```

```
                    t.Log(fmt.Sprintf("len: %d, range: %d:%d",
                      ↪ tc.Length, start, end))
                    res, err := b.ReportRange(start, end)
                    if err != nil {
                        t.Fatal(err)
                    }
                    if string(res) != string(content[start:end
                      ↪ +1]) {
                        t.Fatal(fmt.Sprintf("improperly
                          ↪ reported \n expected: %s\n
                          ↪ got: %s \n", string(content[
                          ↪ start:end+1]), string(res)))

                    }
                })
            }

        }
    }
}

func TestAppend(t *testing.T) {
    buffers := make([]StorageType, 2)
    buffers[0] = mkRope()
    buffers[1] = mkGapBuf()

    testCases := []BufferTest{
        {Length: 100},
        {Length: 200},
        {Length: 500},
        {Length: 1000},
    } // basically, starting with this len in buffer, and then doubling size
```

```
    ↪ with same content
for _, tc := range testCases {
        content := GetContent("testing.txt", tc.Length)
        app := make([]rune, len(content))
        for i, bt := range content {
                app[i] = rune(bt)
        }
        comparison := string(app)
        comparison = comparison + comparison
        for _, b := range buffers {
                b.Load(content)
                t.Run(fmt.Sprintf("%T append", b), func(t *testing.T) {
                        t.Log(fmt.Sprintf("case: %d", tc.Length))
                        t.Log(b.ToString())

                        err := b.Append(app)
                        if err != nil {
                                t.Fatal(err)
                        }
                        res, err := b.Report()
                        if err != nil {
                                t.Fatal(err)
                        }
                        if string(res) != comparison {
                                t.Fatal(fmt.Sprintf("improperly appended \n
                                    ↪  expected: %s\n got: %s \n",
                                    ↪ comparison, string(res)))


                        }
                })
        }
}
```

```go
}

func TestSplit(t *testing.T) {
	buffers := make([]StorageType, 2)
	buffers[0] = mkRope()
	buffers[1] = mkGapBuf()

	testCases := []BufferTest{
		{End: []int{25, 50}, Length: 100},
		{End: []int{50, 100}, Length: 200},
		{End: []int{125, 250}, Length: 500},
		{End: []int{250, 500}, Length: 1000},
	}

	for _, tc := range testCases {
		content := GetContent("testing.txt", tc.Length)
		for _, b := range buffers {
			b.Load(content)
			for sc_idx := range tc.End {
				t.Run(fmt.Sprintf("%T_split", b), func(t *testing.T
				↪ ) {
					point := tc.End[sc_idx]
					if point == -1 {
						point = len(content) - 1
					}
					t.Log(fmt.Sprintf("split: %d", point))
					res, err := b.Split(point)
					if err != nil {
						t.Fatal(err)
					}
					splitStr, err := res.Report()
```

```
                if err != nil {
                        t.Fatal(err)
                }

                if string(splitStr) != string(content[point:]) {
                        t.Log(len(splitStr))
                        t.Log(len(content[point:]))
                        t.Fatal(fmt.Sprintf("bad split off section\n expected: /%s/ \n got: /%s/ \n", content[point:], string(splitStr)))
                }
                remainStr, err := b.Report()
                if err != nil {
                        t.Fatal(err)
                }
                if string(remainStr) != string(content[:point]) {
                        t.Fatal(fmt.Sprintf("bad remaining section\n expected: /%s/ \n got: /%s/ \n", content[:point], string(remainStr)))
                }
                // cleanup
                //t.Log(string(splitStr))
                //t.Log(b.ToString())
                b.Append(splitStr)

        })
}
```

```
                }
        }
}

func TestDelete(t *testing.T) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := []BufferTest{
                {Start: []int{0, 0, 50, 10, 30}, End: []int{-1, 10, -1, 20, 50},
                    ↪ Length: 100},
                {Start: []int{0, 0, 50, 10, 30}, End: []int{-1, 10, -1, 20, 50},
                    ↪ Length: 200},
                {Start: []int{0, 0, 50, 10, 30}, End: []int{-1, 10, -1, 20, 50},
                    ↪ Length: 500},
                {Start: []int{0, 0, 50, 10, 30}, End: []int{-1, 10, -1, 20, 50},
                    ↪ Length: 1000},
        }
        for _, tc := range testCases {
                content := GetContent("testing.txt", tc.Length)
                for _, b := range buffers {
                        b.Load(content)
                        for sc_idx := range tc.Start {
                                t.Run(fmt.Sprintf("%T_Delete", b), func(t *testing.
                                    ↪ T) {
                                        start, end := tc.Start[sc_idx], tc.End[
                                            ↪ sc_idx]
                                        if end == -1 {
                                                end = len(content) - 1
                                        }
                                        t.Log(fmt.Sprintf("len: %d, range: %d:%d",
```

```
        ↪ tc.Length, start, end))
        res, err := b.DeleteRange(start, end)
        if err != nil {
                t.Fatal(err)
        }
        comp := content[start:end]
        if string(res) != string(comp) {
                t.Fatal(fmt.Sprintf("improperly␣
                    ↪ deleted␣section␣\n␣expected:
                    ↪ ␣%s\n␣got:␣%s␣\n", string(
                    ↪ comp), string(res)))

        }

        rem, err := b.Report()
        if err != nil {
                t.Fatal(err)
        }
        comp = make([]byte, len(content))
        copy(comp, content)

        comp = append(comp[:start], comp[end:]...)
        if string(rem) != string(comp) {
                t.Fatal(fmt.Sprintf("improper␣
                    ↪ remaining␣section␣\n␣
                    ↪ expected:␣%s\n␣got:␣%s␣\n",
                    ↪ string(comp), string(rem)))

        }
        b.Insert(start, res)
    })
}
```

```go
                }
        }
}

func TestIndex(t *testing.T) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := []BufferTest{
                {Start: []int{0, 25, 50, 75, 99}, Length: 100},
                {Start: []int{0, 50, 100, 150, 199}, Length: 200},
                {Start: []int{0, 125, 250, 375, 499}, Length: 500},
                {Start: []int{0, 250, 500, 750, 999}, Length: 1000},
        }
        for _, tc := range testCases {
                content := GetContent("testing.txt", tc.Length)
                for _, b := range buffers {
                        b.Load(content)
                        for _, idx := range tc.Start {
                                res, err := b.Index(idx)
                                if err != nil {
                                        t.Fatal(err)
                                }
                                if rune(content[idx]) != res {
                                        t.Fatal(fmt.Sprintf("character incorrectly 
                                           ↪ reported \n expected: %s \n got: %s"
                                           ↪ , string(content[idx]), string(res))
                                           ↪ )
                                }
                        }
```

```go
			}
		}
}

func TestConcat(t *testing.T) {
	buffers := make([]StorageType, 2)
	buffers[0] = mkRope()
	buffers[1] = mkGapBuf()

	extras := make([]StorageType, 2)
	extras[0] = mkRope()
	extras[1] = mkGapBuf()

	testCases := []BufferTest{
		{Start: []int{25, 50, 75, 99}, Length: 100},
		{Start: []int{50, 100, 150, 199}, Length: 200},
		{Start: []int{125, 250, 375, 499}, Length: 500},
		{Start: []int{250, 500, 750, 999}, Length: 1000},
	}
	for _, tc := range testCases {
		content := GetContent("testing.txt", tc.Length)
		for i, b := range buffers {
			for _, idx := range tc.Start {
				b.Load(content)
				extras[i].Load(content[:idx])

				err := b.Concat(extras[i])
				if err != nil {
					t.Fatal(err)
				}
				expected := make([]byte, len(content))
				copy(expected, content)
```

```
                                expected = append(expected, content[:idx]...)

                                res, err := b.Report()
                                if err != nil {
                                        t.Fatal(err)
                                }

                                if string(expected) != string(res) {
                                        t.Fatal(fmt.Sprintf("bad append.\n expected
                                            ↪ : %s\n got: %s\n", string(expected),
                                            ↪  string(res)))
                                }

                        }
                }
        }
}
```

Listing 8: Benchmarks

```
package main

import (
        "fmt"
        "testing"
        "time"
)

func BenchmarkLoad(b *testing.B) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := make([]BufferTest, 0)
```

```
        for x := 1; x < 11; x++ {
                testCases = append(testCases, BufferTest{Length: x * 40000})
        }

        for _, tc := range testCases {
                content := GetContent("pg66576.txt", tc.Length)
                for _, bf := range buffers {
                        b.Run(fmt.Sprintf("%T_load_%d", bf, tc.Length), func(t *
                            ↪ testing.B) {
                                bf.Load(content)
                        })
                }
        }
}

func BenchmarkReport(b *testing.B) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := make([]BufferTest, 0)
        for x := 1; x < 11; x++ {
                testCases = append(testCases, BufferTest{Length: x * 40000})
        }
        var res []rune
        var err error
        for _, tc := range testCases {
                content := GetContent("pg66576.txt", tc.Length)
                for _, bf := range buffers {
                        bf.Load(content)
                        b.Run(fmt.Sprintf("%T_report_%d", bf, tc.Length), func(t *
                            ↪ testing.B) {
```

```
                        res, err = bf.Report()
                        time.Sleep(time.Nanosecond)
                        if err != nil {
                                panic(err)
                        }

                })
        }
    }
    fmt.Println(len(res))
}

func BenchmarkReportRange(b *testing.B) {
    buffers := make([]StorageType, 2)
    buffers[0] = mkRope()
    buffers[1] = mkGapBuf()

    testCases := make([]BufferTest, 0)
    for x := 1; x < 11; x++ {
        amt := x * 40000
        testCases = append(testCases, BufferTest{Start: []int{amt / 8, amt
           ↪ / 4, amt / 2, (3 * amt) / 4}, End: []int{amt / 4, (3 * amt)
           ↪ / 8, (5 * amt) / 8, (7 * amt) / 8}, Length: amt})
    }
    var res []rune
    var err error
    for _, tc := range testCases {
        content := GetContent("pg66576.txt", tc.Length)
        for _, bf := range buffers {
            bf.Load(content)

            // first, changing size with constant report length
```

```
b.Run(fmt.Sprintf("%T_reportRange_constLen_%d", bf, tc.
    ↪ Length), func(t *testing.B) {
        time.Sleep(time.Nanosecond)
        res, err = bf.ReportRange(10000, 30000)
        if err != nil {
                panic(err)
        }

})

for tIdx := range tc.End {
        start := tc.Start[tIdx]
        end := tc.End[tIdx]
        // second, changing length of report

        b.Run(fmt.Sprintf("%T_reportRange_diffLen_%d_%d:%d"
            ↪ , bf, tc.Length, tc.Start[0], end), func(t *
            ↪ testing.B) {
                time.Sleep(time.Nanosecond)
                res, err = bf.ReportRange(tc.Start[0], end)
                if err != nil {
                        panic(err)
                }

        })

        // third, changing position of the report with a
            ↪ constant length of n/8

        b.Run(fmt.Sprintf("%T_reportRange_changePos_%d_%d:%
            ↪ d", bf, tc.Length, start, end), func(t *
            ↪ testing.B) {
```

```go
                                        time.Sleep(time.Nanosecond)
                                        res, err = bf.ReportRange(start, end)
                                        if err != nil {
                                                panic(err)
                                        }

                                })

                        }

                }
        }
        fmt.Println(len(res))
}

func BenchmarkInsert(b *testing.B) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := make([]BufferTest, 0)
        for x := 1; x < 6; x += 1 {
                testCases = append(testCases, BufferTest{
                        Start:  []int{0, x * 10000, x * 20000, x * 40000, x *
                            ↪ 60000, x * 80000},
                        Length: x * 80000})
                //BufferTest{Start: []int{0, 250, 500, -1}, Length: 1000},
        }
        var res []rune
        var err error
        for _, tc := range testCases {
                content := GetContent("pg66576.txt", tc.Length)
```

```go
insert := GetContent("pg66576.txt", tc.Length)
app := make([]rune, len(insert))
for i := 0; i < len(insert); i++ {
	app[i] = rune(insert[i])
}
for _, bf := range buffers {
	bf.Load(content)
	// constant position 2000 character insert
	b.Run(fmt.Sprintf("%T_insert_%d_const", bf, tc.Length), func(t *testing.B) {
		err = bf.Insert(40000, app[:2000])
		//time.Sleep(time.Nanosecond)
		if err != nil {
			panic(err)
		}
		b.StopTimer()
		bf.Load(content)
	})
	// varying start

	for t_idx := range tc.Start {
		start := tc.Start[t_idx]
		b.Run(fmt.Sprintf("%T_insert_%d_Start_%d", bf, tc.Length, start), func(t *testing.B) {
			err = bf.Insert(start, app[:2000])
			if err != nil {
				panic(err)
			}
			b.StopTimer()
			bf.Load(content)
		})
```

```
                                        // varying input size:
                                        if t_idx != 0 {
                                                tmp := app[:t_idx*5000]
                                                b.Run(fmt.Sprintf("%T_insert_%d_Size_%d",
                                                    ↪ bf, tc.Length, (t_idx*5000)), func(t
                                                    ↪ *testing.B) {
                                                        err = bf.Insert(100, tmp)
                                                        if err != nil {
                                                                panic(err)
                                                        }
                                                        b.StopTimer()
                                                        bf.Load(content)
                                                })
                                        }

                                }

                        }
                }
                fmt.Println(len(res))
}

func BenchmarkSplit(b *testing.B) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := make([]BufferTest, 0)
        for x := 1; x < 11; x += 1 {
                amt := x * 40000
                testCases = append(testCases, BufferTest{
                        Start: []int{amt / 8, amt / 4, amt / 2, (3 * amt) / 4, (7
```

```
        ↪ * amt) / 8},
      Length: amt})
}
var res StorageType
var err error
for _, tc := range testCases {
    content := GetContent("pg66576.txt", tc.Length)
    for _, bf := range buffers {
        bf.Load(content)
        // first, constant place

        b.Run(fmt.Sprintf("%T_split_%d_const", bf, len(content)),
            ↪ func(t *testing.B) {
                res, err = bf.Split(20000)
                //time.Sleep(time.Nanosecond)
                if err != nil {
                    panic(err)
                }
                b.StopTimer()
                bf.Load(content)
        })

        // second, varying place
        for t_idx := range tc.Start {
            start := tc.Start[t_idx]
            b.Run(fmt.Sprintf("%T_split_%d_vary_%d", bf, len(
                ↪ content), start), func(t *testing.B) {
                    res, err = bf.Split(start)
                    //time.Sleep(time.Nanosecond)
                    if err != nil {
                        panic(err)
                    }
```

```
                    b.StopTimer()
                    bf.Load(content)
                })
            }
        }
    }
    fmt.Println(len(res.ToString()))
}

func BenchmarkDelete(b *testing.B) {
    buffers := make([]StorageType, 2)
    buffers[0] = mkRope()
    buffers[1] = mkGapBuf()

    testCases := make([]BufferTest, 0)
    for x := 1; x < 11; x += 1 {
        amt := x * 40000
        testCases = append(testCases, BufferTest{
            Start:  []int{amt / 8, amt / 4, amt / 2, (3 * amt) / 4, (7
                ↪ * amt) / 8},
            End:    []int{amt / 4, (3 * amt) / 8, (5 * amt) / 8, (7 *
                ↪ amt) / 8, amt - 1},
            Length: amt})
    }
    var res []rune
    var err error
    for _, tc := range testCases {
        content := GetContent("pg66576.txt", tc.Length)
        for _, bf := range buffers {
            bf.Load(content)
            // first, constant position
```

```
b.Run(fmt.Sprintf("%T_delete_%d_const", bf, tc.Length),
    ↪ func(t *testing.B) {
        res, err = bf.DeleteRange(10000, 30000)
        //time.Sleep(time.Nanosecond)
        if err != nil {
                panic(err)
        }
        b.StopTimer()
        bf.Load(content)
})

for tIdx := range tc.End {
        start := tc.Start[tIdx]
        end := tc.End[tIdx]
        // second, changing length of delete

        b.Run(fmt.Sprintf("%T_delete_diffLen_%d_%d:%d", bf,
            ↪  tc.Length, tc.Start[0], end), func(t *
            ↪ testing.B) {
                time.Sleep(time.Nanosecond)
                res, err = bf.DeleteRange(tc.Start[0], end)

                if err != nil {
                        panic(err)
                }
                b.StopTimer()
                bf.Load(content)
        })

        // third, changing position of the delete with a
            ↪ constant length of n/8
```

```
                                b.Run(fmt.Sprintf("%T_delete_changePos_%d_%d:%d",
                                    ↪ bf, tc.Length, start, end), func(t *testing.
                                    ↪ B) {
                                        time.Sleep(time.Nanosecond)
                                        res, err = bf.DeleteRange(start, end)

                                        if err != nil {
                                                panic(err)
                                        }
                                        b.StopTimer()
                                        bf.Load(content)
                                })

                        }
                }
        }
        fmt.Println(len(res))
}

func BenchmarkIndex(b *testing.B) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        testCases := make([]BufferTest, 0)
        for x := 1; x < 11; x += 1 {
                amt := x * 40000
                testCases = append(testCases, BufferTest{
                        Start:  []int{0, amt / 8, amt / 4, amt / 2, (3 * amt) / 4,
                            ↪ (7 * amt) / 8, amt - 1},
                        Length: amt})
```

```
		//BufferTest{Start: []int{50, 10, 30}, End: []int{ -1, 20, 50},
		    ↪ Length: 100},
	}
	var res rune
	var err error
	for _, tc := range testCases {
		content := GetContent("pg66576.txt", tc.Length)
		for _, bf := range buffers {
			bf.Load(content)
			// first, constant position
			b.Run(fmt.Sprintf("%T_index_%d_const", bf, tc.Length), func
			    ↪ (t *testing.B) {
				res, err = bf.Index(20000)
				time.Sleep(time.Nanosecond)
				if err != nil {
					panic(err)
				}

			})

			// second, changing position
			for _, idx := range tc.Start {
				b.Run(fmt.Sprintf("%T_index_%d_at_%d", bf, tc.
				    ↪ Length, idx), func(t *testing.B) {
					res, err = bf.Index(idx)
					time.Sleep(time.Nanosecond)
					if err != nil {
						panic(err)
					}

				})
			}
```

```
                }
        }
        fmt.Println(res)
}

func BenchmarkAppend(b *testing.B) {
        buffers := make([]StorageType, 2)
        buffers[0] = mkRope()
        buffers[1] = mkGapBuf()

        extra := make([]StorageType, 2)
        extra[0] = mkRope()
        extra[1] = mkGapBuf()

        testCases := make([]BufferTest, 0)
        for x := 1; x < 11; x += 1 {
                amt := x * 40000
                testCases = append(testCases, BufferTest{
                        Start:  []int{2000, amt / 8, amt / 4, amt / 2, (3 * amt) /
                           ↪ 4, (7 * amt) / 8, amt},
                        Length: amt})
        }

        var res []rune
        var err error
        for _, tc := range testCases {
                content := GetContent("pg66576.txt", tc.Length)
                for i, bf := range buffers {
                        for _, length := range tc.Start {
                                bf.Load(content)
                                extra[i].Load(content[:length])
```

```
				b.Run(fmt.Sprintf("%T_Append_%d_%d", bf, len(
				    ↪ content), length), func(t *testing.B) {
					err = bf.Concat(extra[i])
					time.Sleep(time.Nanosecond)
					if err != nil {
						panic(err)
					}
					b.StopTimer()
					bf.Load(content)

				})

				res, err = bf.Report()
				if err != nil {
					panic(err)
				}
			}

		}
	}
	fmt.Println(len(res))

}
```

Listing 9: Results Converter

```
package main

import (
	"encoding/csv"
	"fmt"
	"os"
	"regexp"
)
```

```go
func ReadRes(res []byte) map[string][]string {
	// convert res into string
	interStr := string(res)
	// first, split results along new line
	splitline := regexp.MustCompile("\n").Split(interStr, -1)

	ret := make(map[string][]string)
	testRE := regexp.MustCompile("Benchmark.*-8")
	opRE := regexp.MustCompile(" 0.*ns/op")
	for _, s := range splitline {
		testname := testRE.FindString(s)
		testname = regexp.MustCompile("Benchmark.*main").ReplaceAllString(
		    ↪ testname, "")
		optime := opRE.FindString(s)
		optime = regexp.MustCompile("ns/op").ReplaceAllString(optime, "")
		if _, ok := ret[testname]; ok {
			// if the testname is already in, append to it
			ret[testname] = append(ret[testname], optime)
		} else {
			ret[testname] = make([]string, 1)
			ret[testname][0] = optime
		}
	}
	/*
		for k, r := range ret {
			fmt.Printf("/%s/: [", k)
			for _, s := range r {
				fmt.Printf("%s,", s)
			}
			fmt.Printf("]\n")
		}
```

```go
	*/
	return ret
}

func PrepareForWrite(in map[string][]string) [][]string {
	ret := make([][]string, 0)
	for k, v := range in {
		tmp := make([]string, 0)
		tmp = append(tmp, k)
		tmp = append(tmp, v...)
		ret = append(ret, tmp)
	}
	return ret
}

func main() {
	test := "insert2"
	ret, err := os.ReadFile(fmt.Sprintf("results/%s.txt", test))
	if err != nil {
		panic(err)
	}
	res := PrepareForWrite(ReadRes(ret))

	f, err := os.Create(fmt.Sprintf("results/%s.csv", test))
	if err != nil {
		panic(err)
	}
	defer f.Close()
	cwrite := csv.NewWriter(f)
	cwrite.WriteAll(res)

}
```

Listing 10: Grapher

```
import csv
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats
from scipy.interpolate import interp1d
import sympy as sym
import matplotlib.ticker as ticker

name = "split2"
from matplotlib import rcParams
rcParams['font.family'] = 'Liberation Serif'
rcParams['text.usetex'] = True

def scifmt(x):
    ret = "{:.4e}".format(x)
    ret = ret.split("e")
    if ret[1][0] == "-":
        ret[1] = ret[1].replace("0", "")
    return ret[0] + '*10^{' + ret[1] + '}'

def logfit(x, y):
    slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(np.log(x),
        ↪ y)

    label = '$y={} \ln x + {}, R^2 = {:.4f}$'.format(scifmt(slope), scifmt(
        ↪ intercept), r_value*r_value)
    return slope * np.log(x) + intercept, label

def linfit(x,y):
    slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(x, y)
    label = '$y={}x + {}, R^2 = {:.4f}$'.format(scifmt(slope), scifmt(intercept),
```

```
        ↪ r_value*r_value)
    return slope * x + intercept, label



def pfit(x,y):
    slope, r, rank, singular_values, rcond = np.polyfit(x, y, 2, full = True)
    slopes = slope.copy()
    tmp = np.polyval(slope, x)
    inter =interp1d(x,tmp, kind = "quadratic")
    newx = np.linspace(x.min(), x.max(), 20)

    yhat = tmp
    ybar = np.sum(y)/len(y)
    ssreg = np.sum((yhat-ybar) ** 2)
    sstot = np.sum((y - ybar) ** 2)

    label = '$y={}x^2 + {}x + {}, R^2 = {:.4f}$'.format(scifmt(slopes[0]), scifmt(
        ↪ slopes[1]), scifmt(slopes[2]), ssreg/sstot)

    return inter(newx), label, newx

with open(name+".csv", newline = '') as csvfile:
    res = csv.reader(csvfile)
    names = []
    scale = []
    data = [[],[],[],[]]
    #       rope, rope err, gap, gap err
    ct = 0
    for row in res:
        if ct == 0:
            names = row
            ct+=1
```

```
    else:
        idx = 0
        scale.append(float(row[0]))
        for el in row[1:]:
            data[idx].append(float(el))
            idx +=1
fig = plt.figure()
dep = fig.add_subplot(111)
dep.errorbar(scale, data[0], data[1], fmt = "o", label = "Rope", linewidth = 2,
    ↪  capsize = 4)
dep.errorbar(scale, data[2], data[3], fmt = "o", label = "Gap Buffer",linewidth
    ↪  = 2,  capsize = 4)
resr, lr = linfit(np.asarray(scale), data[0])
plt.plot(scale, resr,label = "Rope Regression: "+ lr)

resg, lg, newx = pfit(np.asarray(scale), data[2])
plt.plot(newx, resg,label = "Gap Buffer Regression: "+ lg)

lg = plt.legend(bbox_to_anchor = (0.5, -0.3), loc = "center")



plt.suptitle(names[0])
plt.xlabel(names[1])
plt.ylabel("Avg. Time / Operation (ns)")
if len(names) > 2:
    plt.title(names[2])

plt.savefig(name + '.png',
            dpi=300,
            format='png',
            bbox_extra_artists=(lg,),
            bbox_inches='tight',
```

```
            pad_inches = 0.5)

#plt.show()
```

C Spreadsheets

Since there is no way to put Excel / Google Sheets spreadsheets into LaTeX, the PDF export will begin on the next page.