CPE 593 Project - Rope-Like Text Buffer

Jayden Pereira jpereir2@stevens.edu
Nathan Renner nrenner@stevens.edu
Edgar Castaneda-Vargas ecastane@stevens.edu

I. Abstract

This data structure report goes into detail about what a Rope structure is and why use it over a string. Furthermore, the team's process for creating this data structure is discussed by explaining the different functions used to provide the Rope's functionality. Included are diagrams to aid the understanding of what a Rope is and the different functions the team was able to create. The team then benchmarked the times of the functions for both Rope and string, and drew conclusions based on the results. This report fully covers every aspect of the project in terms of both successes and shortcomings and shows a full understanding of the data structure the team created through our final project.

II. Introduction

A Rope data structure is a type of binary tree where every node can have up to two children nodes. What makes a Rope different from a regular binary tree is that only the leaf nodes contain the string as well as the length of the string which is also known as the weight. The parent nodes do not hold a string, but they hold the sum of the weights of their leaf nodes in their respective subtree. The question now becomes why use this structure over a regular string. One of main advantages of Rope is that it is much faster in terms of inserting and deleting text when compared to string arrays. The complexity of strings to complete these tasks is O(n) while the complexity of a Rope is

O(logn) which is considerably faster and is why Ropes are used in applications like text editors. Moreover, in order to do copying operations on a string array, O(n) extra memory is required, but this is not the case for a Rope.

While Ropes are incredibly fast, the structure has its disadvantages. When it is not being operated upon, more storage space is required than a simple string. This is due to storing the parent nodes. Moreover when doing operations on small strings, a Rope is significantly slower than a string array. As a result, a Rope is not a substitute for a string in all situations. A Rope fully shines when dealing with large strings. As the string grows, the trade off of requiring more storage is reduced, and the operations become faster in comparison. Thus, a Rope is better when dealing with a large size of data that is being modified on a regular basis while a string is better with smaller data and fewer operations.

In order to compare the Rope data structure with the string array, the team was tasked with benchmarking several operations. These include loading a 100MB file of text into the Rope structure, searching for the kth byte as wells as searching for a string from byte m to byte n, replacing a string from byte m to n, inserting, deleting, and appending up to 10KB of text , and saving the entire text to a file. The team used Unix's built-in time function to get each test's runtime and compare the times between the Rope functions and the strings equivalent functions.

III. Method

Given our set of tasks, we first decided to tackle how to algorithmically create and self-balance the BST-like structure. Since the documentation and resources found online

covered mostly the manipulation of the Rope, we started off building the structure by hand in order to spot patterns that could be turned into a method for creation.

However, we first needed to establish a rule within our implementation of the data structure in order to start developing creation methods. Based on the reading found, the leaf nodes (which we refer to as 'chunks') were often seen with variable length with no discernable advantage to be found, as adding a chunk size randomizer would actually make the code slightly more complex. Instead, we decided to standardize the chunk size to 10 bytes, such that we would reduce the overall height of the tree slightly while not introducing a large amount of complexity if iterating through to the last byte in the chunk if we increased the chunk size to be larger. This removal of randomized chunk size additionally did not interfere with the search algorithm discussed later.

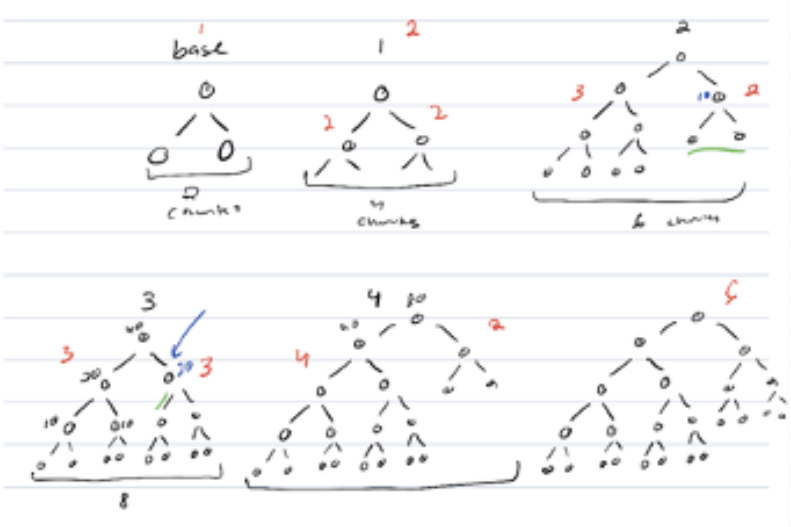

Figure 1: Hand-drawn Rope creation

The most difficult aspect when creating the Rope stems from having to self-balance the tree, as the time complexity of searching, insertions, and deletions are all based on having a balanced tree. The strength of a BST in general, is in iterating vertically through the tree instead of horizontally through each chunk.

We took a look at possibly integrating a Red-Black, AVL, and even a Splay BST into our structure, but the main problem stemmed from the rotations these algorithms did to the branches of the trees. Since each in-order leaf node contains the continuous chain of characters for the entire string, we needed to balance the tree *while maintaining string continuity*.

We then found a pattern in our hand-drawn construction: two cases emerged on the basis of left and right subtree heights. The first case is when the left and right subtree heights are of equal height; in this scenario, a new root can be made with a calculated left subtree weight (with the formula: *weight = (2 ^ height) * chunk size*) and a new subtree gets added to the right pointer of the new root (as shown in figure 2).

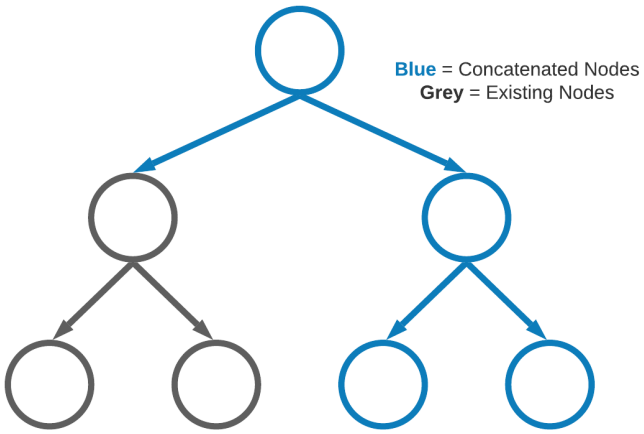

Figure 2: Case 1: Existing tree of equal subtree height

The second case is when the two branch heights are unequal and thus, to try and balance the tree, the current right subtree becomes the left pointer of the new right subtree (as shown in figure 3).

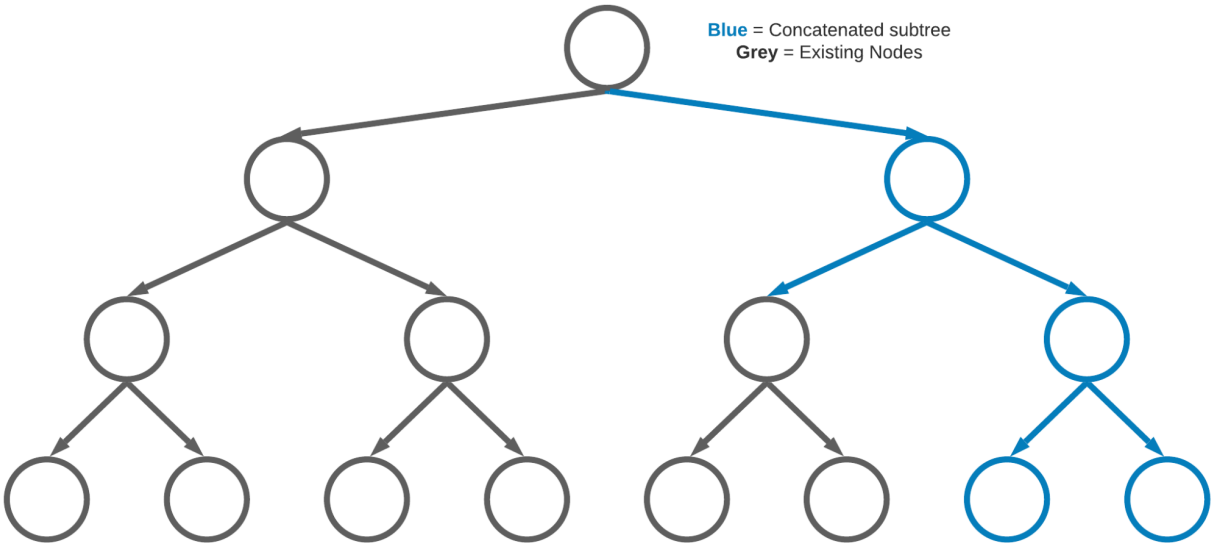

Figure 3: Case 2: Existing tree of unequal subtree height

Our next method we needed to implement is a way to actually traverse the tree. Since this structure is different from a BST with information only contained in the leaf nodes, we needed a unique way to grab specific bytes from a file as large as 100MB. In order to do so, we utilize the weights assigned to each node, as the weight value is actually the number of bytes contained in the left-handed subtree of the node. Following the example on figure 4, in order to find the 26th byte of a 40-byte string, the initial index is 26 and the weight of the root is 20. Since the index is higher, the new search index is the original index minus the weight and the search then recurses to the right pointer. Now the index is 6 and the weight of the node is 10, since the index is lower than the weight, the search just continues to the left pointer with no change to the index. In the example, the search ends with an index of 6 and on a leaf node – which means, iterating to the 6th character in the chunk yields the 26th byte of data overall.

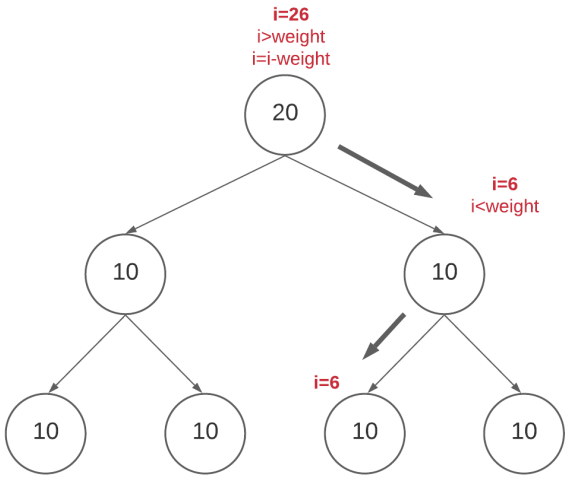

Figure 4: Search algorithm walkthrough

Now that the Rope is created and can be traversed, the last method needed to be developed is manipulating the tree with insertions and deletions. When initially researching, the intended way to accomplish this task was splitting apart the tree and reconstructing the tree while self-balancing. As the operation was incredibly complex and late into development we discovered our problem with balancing large trees, we decided to approach the insertion algorithms with some compromise – knowing that we could do better but trying to come up with a solution that maintained string continuity and kept the tree as balanced possible to lose the least amount of performance. Instead of breaking subtrees into individual nodes and refactoring the tree, possibly messing up by not having the correct weights assigned to nodes, we decided to just create a new right subtree – with the parent node's left pointer being the subtree that was inserted and the right pointer the existing right subtree. This new subtree is then placed at the

right pointer of the root (as shown in figure 5).

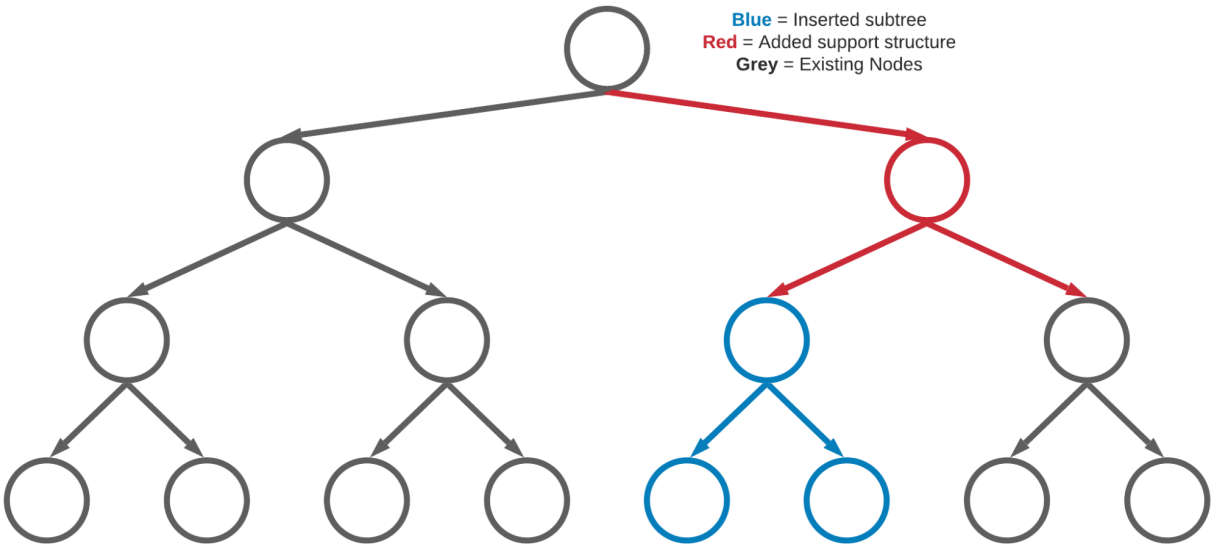

Figure 5: Implementation of insertion method

The additional tasks were all variations of these three methods in some way, from searching for specific indices to remove nodes or creating larger subtrees to insert into the tree.

IV. Results

To benchmark our Rope structure, we conducted a series of uniform tests, to maintain consistency. We tested the Rope structure along with the string library to compare our results to, along with reading in an 100MB file to test the structures on. To test overall performance, we performed the following tasks in order (*see figure: 6*):

- Create a new Rope with a specified filename.
- Search for a specific byte.
- Search for a specific string.

- Replace a specific string (which is commented out for an error to be explained later in this section).
- Insert a string in the middle of the file.
- Delete specific bytes.
- Save the data to a new file.

```
// Create rope
Rope rope;
rope.createRopeWithFile(fileName);
rope.searchForByte(k, rope.root); // getting kth byte
rope.getString(m, n, rope.root); // get a string from byte m to n
//rope.replaceString(m, n, stringOfXLength(n-m), rope.root); // replace string from m to n - ERROR (segmentation fault)
rope.insertStringAtMiddle(stringOfXLength(x), rope.root); // insert X bytes in middle
rope.deleteBytes(x, rope.root); // deleting X bytes from end
rope.saveAsFile(rope.root); // saving text to file

// Create String
/*
string test = loadFileIntoString(fileName);
test[k - 1]; // getting kth byte
test.substr(m, n - m); // get a string from byte m to n
//test.replace(m, n - m, stringOfXLength(n-m)); // replace string from m to n
test.insert(50000000, stringOfXLength(x)); // insert X bytes in middle
test.erase(test.end() - x, test.end()); // deleting X bytes from end
saveStringIntoFile(test); // saving text to file
 */
```

Figure 6: Benchmarking code

To maintain uniformity, the same values were used for all variables, which were: k = 200,000, m = 200,000, n = 350,000, and x = 50,000. In order to run the test, Unix's built in time function was used alongside the executable file to find the time it takes the program to run. After we performed the tests, the resulting times are shown below (figure 7).

Test	Rope	String
1	9.226	10.634
2	9.029	10.706
3	8.831	10.956
4	8.835	10.787
5	8.844	10.938
Average	**8.953**	**10.804**

Figure 7: Results table

The benchmarking results show that the Rope structure has a 17.13% decrease in overall time compared to the normal C++ string. This shows our effort was a success, even if the string is not operating at 100% efficiency.

The tests were not a complete success, however, as we had a problem along the way: `replaceString()` keeps producing a segmentation fault. Below is how the method is called, with variable "n" being larger than "m", and `stringOfXLength()` being a predefined helper method.

```
rope.replaceString(m, n, stringOfXLength(n-m), rope.root);
```

This method works by looping through the start and end bytes of the string needed to be replaced, and calling `replaceByte()` during each iteration. `replaceByte()` uses the same code as `searchForByte()` (which works as intended), but instead of returning the byte in question, it simply replaces the byte. For whatever reason, the code would

start to work as intended, replacing the first “x” amount of bytes, but randomly threw a segmentation fault regardless of the size of the string to replace. The team heavily debugged this issue, but could not find a common pattern of which to find a solution. This could possibly be with the way in which C++ handles the (up to) millions of recursive calls, but this is without certainty.

V. Conclusion

After further testing, we had started to notice in small file sizes the construction method worked well - however, when testing on large files like our main 100MB test, the Rope started to become skewed towards its left side. This helps to understand, while we did indeed increase the speed in comparison to the string library, the overall efficiency of the structure was limited in part due to the exact implementation of the data structure. Work was done to reverse engineer other Rope libraries (such as Ropey) in order to remedy these detriments, but the complexity of the unique BST posed a considerable challenge.

This project could be further expanded upon by redesigning and implementing more effective Rope creation and splitting methods. A Rope that can self-balance more effectively on creation will benefit more from the time complexity advantage a BST has on basic character arrays. Additionally, a properly implemented splitting method would create new replace string and delete string methods that would run in a much quicker time.

Overall, we were able to successfully complete most of the tasks given, and even though some of the functionality was not done in the most efficient way, we were able to

create a data structure that held, manipulated, and saved a 100 MB string and was faster than a standardized library by over 17%.

V. References

Agarwal, Umang. “Rope Data Structure.” *Medium*, Underrated Data Structures and Algorithms, 17 Apr. 2020, medium.com/underrated-data-structures-and-algorithms/rope-data-structure-e623d7862137.

Bagwell, Philip, and Tiark Rompf. “RRB-Trees: Efficient Immutable Vectors.” *Infoscience*, 1 Jan. 1970, infoscience.epfl.ch/record/169879.

Boehm, Hans-j., et al. *Ropes: an Alternative to Strings*. www.cs.rit.edu/usr/local/pub/jeh/courses/QUARTERS/FP/Labs/CedarRope/rope-paper.pdf.

Farinier, Benjamin, et al. *Mergeable Persistent Data Structures*. hal.inria.fr/hal-01099136v2/document.

Kalra, Ishmeet Singh. “Rope: the Data Structure Used by Text Editors to Handle Large Strings.” *OpenGenus IQ: Learn Computer Science*, OpenGenus IQ: Learn Computer Science, 27 Aug. 2019, iq.opengenus.org/rope-data-structure/.

Vegdahl, Nathan. “Ropey.” *GitHub*, 30 Dec. 2020, github.com/cessen/ropey.