Strings to Ropes: Unraveling the Knots

Adithya Raman

INTRODUCTION

Ropes was the data structure that I chose for my project. Their ability to contain an entire string and do all the essential string operations faster than a standard string implementation would benefit the grand scheme of things, especially considering how vital the string is to everyday language. I implemented and tested it in Java using IntelliJ Community Edition. Overall, this experience was exciting and taught me much about my coding skills and the steps to making the most optimized data structure possible to accomplish a specific task. In my case, it was being able to make the most efficient use of text editing, optimizing the ability to edit.

Before starting this project, my understanding of data structures came mostly from things done in my Computer Science curriculum. That means understanding how and why a data structure works and leaving it at that. It was purely theoretical. Coming up with an implementation was something unlike what I had done before. It was not just using a pre-existing implementation to solve a task. Instead, it was like I was making a building; I knew what the final product should be and what it should do, but other than that, I was on my own to devise how to do it. I came to appreciate the ins and outs of being a developer of a structure. Even though I had an idea of what I needed to do, imagining having to go through and think about what I would need to trade off to get the structure to behave how I wanted it to was challenging and made me respect the engineers who wrote the optimized libraries.

TESTING

After creating and testing with Ropes, I came to many conclusions about ropes and how they work in text optimization. To start with, let's talk about the experience I had while creating the

Ropes. The process was very heavily demanding to develop. When I wrote my original paper, I said that one of the core problems with ropes is the tremendous amount of overhead in creating and implementing the class. After going through the process of making it, the con was entirely justified as a con. The con is extra emphasized when we compare it to the built-in string implementation, which requires no overhead and no implementation whatsoever.

The fact that the functions were so interdependent was often a point of frustration for me while working on the implementation . The constant interdependence made it harder to debug, mainly because if a function was not working correctly, it was not immediately clear whether that function was the issue or whether it was not a case tested for in the other functions. For example, a small error in the weight calculations would cause a different problem in concatenation, which would then cause a failure in the substring, making it very hard to trace back to the root of the problem.

Seven main functions needed to be created. Those functions were split, concatenate, substring, insert, remove, rebalance, and rotate right and left. The rebalancing functions will be focused on later, but the other five functions were highly codependent, especially with split. This is the function that caused me the most trouble. Split as predicted in the original paper would anchor the functions. Insert and remove use it by splitting at a specific index and then either concatenating the first half with the new second half or by breaking the beginning and ending and removing the middle from the tree.

On paper, the implementation sounds pretty simple. However, getting down to the implementation, it was a lot trickier than first expected. The reason was that it had three main parts behind it. First, it had to traverse the entire tree to find the splitting position recursively. Next, it had to divide the split point into two ropes while ensuring everything was maintained. Finally, it needed to be rebuilt and rebalanced from then on. The splitting had issues logically at all three parts and required multiple test cases to ensure it would work properly.

Once the splitting was correctly created, the insert, remove, substring, and concatenate functions were much more trivial. The only issue that came about was when there was an error in any of the four functions, I would immediately check if split was the baseline cause because of how deeply

integrated it was as part of the code. After the creation of all the functions when I started running the test cases it was not producing the same output as the strings at the start. It would only produce the proper result for the smallest text file size. Once the text size got to medium and large it started not producing the same values.

```
/Users/adithyaraman/Library/Java/JavaVirtualMachines/openjdk-22.0.1/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=64405:/Applica
String Time: 6240250
String Memory Used: 1441888
Rope Time: 56250
Rope Memory: 1462288
Equals each other

String Time: 257166
String Memory Used: 1495120
Rope Time: 251042
Rope Memory: 1504392
Does not equal each other

String Time: 1505459
String Memory Used: 1572032
Rope Time: 141959
Rope Memory: 1592752
Does not equal each other


Process finished with exit code 0
```

This error was both immediately frustrating but also showed me that despite everything working just fine on their own in isolation, that when they interacted with each other there were subtle bugs that occurred. This forced me to create smaller test cases combining multiple of the functions together to highlight where and what exactly were causing the errors to occur. If there was an important thing I learned from this part of the implementation it would be that once a function has been created it is important to test it with the previous functions made also if both are accessing the same place. Otherwise you run the risk of your new creations messing up your old ones and vice versa. Everything needs to hold a certain level of cohesion.

Ropes relies heavily on a balanced binary tree structure, which requires rebalancing, much like an AVL tree. My prior experience with AVL Trees proved invaluable to this Ropes implementation because it would have taken far longer than it originally did without the foundation I already had in AVL trees. The balancing borrows heavily from the AVL tree, where it maintains the tree through height invariants. In particular, it rotates much like an AVL tree. However, these concepts in ropes brought forth newer issues. The tree needed to maintain the weights and the order of the tree. However, in an AVL tree case, it would only affect performance. For a rope, if the balancing were not done correctly, it would not only affect performance but also corrupt the overall storage of

the string. This created a unique challenge that went beyond standard AVL tree implementation. However, it was important that this balancing was done so that I could accomplish everything in O(log n) time.

RESULTS

The results of my testing shocked me. The main thing I was testing in my code was the runtime; the results both surprised me and did not surprise me simultaneously. What was not surprising about my code was that, on average, the ropes did a far better job than the original string implementation. What shocked me was how well it did on small text, and that, on average, the original string implementation took far longer on the smaller texts than the larger ones.

RUNTIME OF REGULAR STRING IMPLEMENTATION(IN NANOSECONDS)

Small Text Small Edits	Medium Text Small Edits	Large Text Small Edits	Small Text Medium Edits	Medium Text Medium Edits	Large Text Medium Edits	Small Text Large Edits	Medium Text Large Edits	Large Text Large Edits
284625	3674875	105583	3540000	271666	473208	3666000	420750	1902708
190750	3518375	100541	3709250	294875	500666	3585667	433250	1775167
187000	3515417	92000	3294500	243625	517375	3813750	443042	1856000

RUNTIME OF ROPE IMPLEMENTATION(IN NANOSECONDS)

Small Text Small Edits	Medium Text Small Edits	Large Text Small Edits	Small Text Medium Edits	Medium Text Medium Edits	Large Text Medium Edits	Small Text Large Edits	Medium Text Large Edits	Large Text Large Edits
61875	72166	55667	143916	124500	271167	527291	146791	442167
52834	64875	48041	231667	135875	267166	386916	179541	484875
60500	83459	45416	140875	123000	270959	545292	172375	524709

When I wrote my original paper, I expected it to be slower when doing fewer edits on a small text and worse overall for small and medium texts. However, the results show that almost the opposite is true. In smaller and medium texts, the runtime is far greater. After doing a bit of research on this, it comes down to the updates that have been made for Java. The JVM has been optimized so that better things can be done to ensure that the fastest runtime is possible for larger

strings. As I mentioned in my previous paper, a string copies over values for insertion, deletion, and concatenation by remaking the entirety of the string and adding it to the new creation. It has been optimized by reusing arrays, pooling its memory, and using its own internal substrings. But that is dependent on the density of the editing. Simply put, five characters for a 50-character text are worth more than five for a 5000-character text. As a result, when it comes down to editing, it can't use the same optimizations for the 50-character text because of how much sheer percentage those five characters hold in the context in the entirety of the operation.

MEMORY OF REGULAR STRING IMPLEMENTATION(IN BYTES)

Small Text Small Edits	Medium Text Small Edits	Large Text Small Edits	Small Text Medium Edits	Medium Text Medium Edits	Large Text Medium Edits	Small Text Large Edits	Medium Text Large Edits	Large Text Large Edits
1449680	1532392	1650016	1474560	1528776	1605616	1480400	1526712	1605448
1449680	1532392	1650016	1474560	1528776	1605616	1480400	1526712	1605448
1449680	1532392	1650016	1474560	1528776	1605616	1480400	1526712	1605448

MEMORY OF ROPE IMPLEMENTATION(IN BYTES)

Small Text Small Edits	Medium Text Small Edits	Large Text Small Edits	Small Text Medium Edits	Medium Text Medium Edits	Large Text Medium Edits	Small Text Large Edits	Medium Text Large Edits	Large Text Large Edits
1472352	1554024	1757312	1495776	1543304	1638264	1503768	1530520	1612488
1472352	1554024	1757312	1495776	1543304	1638264	1503768	1530520	1612488
1472352	1554024	1757312	1495776	1543304	1638264	1503768	1530520	1612488

Another part that surprised me about my testing was the memory results. Every single run had the same memory result at the end of it. It was to be expected as I had it running through a scripted thing of edits that although would pick out a random point was not changing the amount it was changing the text by. In my original paper, I predicted that it would be less memory efficient at small sizes, but as it grew, its memory overhead would not be as bad as it would have a little bit less than the strings implementation. What the data shows, however, is that in every case, the strings

were slightly more memory efficient than the ropes implementation. Part of this surprised me, as the rope structure would do a far better job with the mixture of immutability. However, I believe this is not entirely due to the Rope itself, but some may be due to the implementation.

To be more efficient, there is a lot more that needs to be done to the ropes implementation that I did not know of. On further research, two parts make it far easier to reduce the memory size, which allows the Rope to be more memory efficient than the strings. One of them is memory pooling. In my implementation, there was very little reusing of nodes. That meant that too many temporary nodes could have been copied over whenever a new substring was created. The second was the way I went about balancing the tree. As I mentioned earlier, I chose an AVL style of balancing the tree. However, this method is very aggressive, so nodes would be created to ensure the correct string is made at the end. I mainly chose the AVL style because I was familiar with it, and I knew it ensured that I would use a BST-style structure to make all operations be O(log n). A less aggressive method of balancing would do a far better job of optimizing the Rope's memory. If I were to start this project from scratch again, I would invest more time optimizing my memory usage by reusing nodes and coming up with a more lightweight way that is less aggressive for the balancing.

CONCLUSION

The overall results of the testing goes as follows. The amount of time it took to implement the structure was incredibly large. It was something that required a lot of time and dedication to accomplish. As for runtime it fully accomplished what it set out to do. It was faster and was able to be much more efficient than the string implementation. As for the memory it was overall slightly more memory intensive than the regular string implementation. My overall conclusion about using ropes depends on the type of machine you use and overall the number of edits you want to make. My machine works quite fast and makes the changes quickly, both with the rope and string implementations. This made it so that the optimization for 100 edits was not as beneficial as it would be overall because the implementation was a headache on top of having to use it. The slower the machine, the more valuable the ropes will become in Java. The need for ropes decreases significantly as the machine can process and go faster. And since memory shows that Ropes are

slightly more so than the original string implementation, that is the only case I see using Ropes. If the user is making constant edits where the number cant be quantified and could be as large as possible then I can say it is worth it for them to use Ropes on a higher end machine. Personally, what I can confidently say is that Ropes have their place, but only get their true power when the number of edits gets so exponentially large that the log(n) timing starts to matter, or when the machine is so slow that any optimization in time is incredibly beneficial. However, for an overall general-purpose application without the constant editing in mind, the trade-offs between development time and complexity and performance overall mean ropes are an optimization with a more niche use case rather than an overall replacement for strings altogether.

BIBLIOGRAPHY

Boehm, H.-J., Atkinson, R. and Plass, M. (1995), Ropes: An alternative to strings. Softw: Pract. Exper., 25: 1315-1330. https://doi.org/10.1002/spe.4380251203

Agarwal, U. (2020, April 17). *Rope Data Structure*. Medium. https://medium.com/underrated-data-structures-and-algorithms/rope-data-structure-e623d7862137