Strings to Ropes: Tying the Knots for Text Optimization

Adithya Raman

INTRODUCTION

Strings are a cornerstone of many programming languages. After all, it makes sense that the data type representing human language is prevalent, no matter the language. Therefore, strings should have a far better implementation than they currently have. In many languages, strings are often kept as an array of characters and face issues such as immutability. Everyday operations like concatenation, insertion, and deletion often require O(n) runtime, leading to poor performance.

Furthermore, there is excessive memory fragmentation in languages where we have to deal with immutability, such as Java. When the scale of the application increases, the use of conventional strings becomes impractical. This is where the use of Ropes becomes increasingly useful. Ropes have a tree-based structure and are incredible at addressing the challenges of regular string implementation, reducing many operations to O(log n) or better.

ROPES

How Ropes Work?

Unlike the array-type structure, strings with Ropes take a string and represent it as a self-balancing binary tree. Each leaf node of a rope contains a short fragment of what the string was. This allows the operations in the leaf nodes to run in O(1) runtime because every leaf node is small and compact at every level. This small and compact setup makes core operations such as concatenations and insertion much easier by making it so that the rope's O(n) runtime is the time it takes to traverse through the tree, typically O(log n). Each internal node stores the length of the string in its left subtree. Storing these lengths in the internal nodes allows for an efficient BST style of searching for a specific index in the string. Much like how a BST reorganizes values based on

which values are greater on the right and smaller on the left, Ropes organize positions based on the lengths of the substrings.

Furthermore, Ropes have self-rebalancing properties, much like an AVL tree. When a height difference is greater than one between any node's left and right subtree, the tree will rebalance itself to keep the BST-type property. This ensures the tree will always have O(log n) run time for its operations.

Core Operations and Their Efficiency

The rope data structure is best implemented when many strings constantly have to deal with modification. When caring about performance, traditional strings often suffer when doing widespread operations. These include concatenations, insertion, deletion, substringing, and splitting. When looking at these functions individually, we can see a drastic difference in O(n) runtimes for all of these functions.'

To start, let us begin with concatenation. In typical array-based implementations, this requires an O(n) runtime. This is because it requires us to create a new array and then traverse through all the characters in each string we are concatenating and add them to the new string. However, we can change this from O(n) time to constant when we use a rope. This is because we never have to go through every character and add it to anything new. Instead, all that needs to be done is create a new node and set the strings as its children. No copying must be done, which runs in an O(1) runtime. The only thing that happens is creating one node and setting two pointers to the parts we want to concatenate.

Before moving on to insertion and deletion, it is important to explain one of the most important operations in the rope data structure called splitting, which allows insertion and deletion to run much more efficiently than the array-based implementation that regular arrays or immutable strings have to go through. The rope goes through the tree to find the target index and then reconstructs two subtrees representing the original string's left and right sides. This works better than original string structures because it does not rely on allocating new memory and copying characters. Instead, it uses manipulating pointers to split in O(log n) runtime. This optimization makes insertion and

deletion much more efficient than in standard string implementations.

To insert a substring, a rope needs to split at the desired index and then introduce the substring to the rope. Finally, all it needs to do is concatenate the three parts. All these operations run in O(log n) time. Deletion works much in the same way using the same principle as insertion. First, it splits the rope at the start and the end of the region to be split and then keeps the remaining two parts at the end of the pointers, essentially discarding the center. All parts run in O(log n) time, much like insertion. Their insertion and deletion require O(n) time than array-based implementation. This is because it involves shifting characters over or creating an entirely new array and all the characters into the new one.

Indexing is the only area where the traditional array-based implementation tends to have the advantage over Ropes. In an array-based implementation, accessing a character at a specific index takes O(1) runtime because of the ability to direct access into the container-type structure of arrays. This makes the array-based structure ideal when we randomly grab a character. Compared to Ropes that require a tree traversal to find an index, they require O(log n) time to index, which is far slower. However, practically, this advantage holds no bearing regarding functionality. Regarding practical applications, editing strings, concatenating, inserting, and deleting are far more important than indexing. The rope trades slower indexing for much faster modifying power, which is far more helpful in real-use cases.

Real World Implementation

The rope has already been used in real-world applications, mainly regarding text editing and text editors. Due to their quick ability to insert, delete, and concatenate, especially compared to normal string implementations. This allows files spanning thousands to millions of characters to have edits in them in O(log n) time. This makes the rope ideal for platforms requiring constant text-based editing, such as Google Docs or Microsoft Word documents. Furthermore, creating undo and redo systems becomes far more convenient because the Ropes can help keep track of the changes by changing tree nodes rather than the entire string. The rope does an excellent job in situations where it can take a large string and break it down into easy to work with chunks that

allow for quick and fast text editing.

Disadvantages of Ropes

Ropes are a fantastic data structure when it comes to manipulating large strings. However, it is important to mention the disadvantages. One of the main drawbacks is that it is far more challenging to implement, especially compared to regular array-based structures built into most languages. Ropes require building and maintaining a binary tree and keeping track of the weights of the subtree, along with going through and rebalancing the tree to maintain its log performance. This makes the structure far more complex to implement and debug than regular string implementations, which often do not require user work. Adding on, it has a tremendous amount of memory overhead compared to standard string implementations because it only stores the strings but also pointers and additional data, such as the weights at each internal node. Finally, Ropes do wonderfully when they are manipulating large strings, but when we scale down to smaller strings and have to do less editing, Ropes become far more inefficient as the amount of overhead required outweighs the benefits when it is at such a small scale or requires little to no change. Ropes are powerful tools but have tradeoffs unsuitable for every string application. However, when the scale increases and the need to edit arises there, it does a far better job at doing those jobs than the original implementation.

String Immutability

Immutability is a crucial part of programming. When a program has something immutable, you cannot change the data of the object you have created; instead, when modifying something, a new version of that object is created with those changes added. This allows the prevention of multiple different bugs related to modifying shared data. With the data being immutable, a lot of the risks associated with them are prevented. This also allows for easier tracking of how data changes over time as the old data remains untouched. Ultimately, it is important because it provides predictability and stability when writing code.

Ropes are a natural fit for immutable languages due to their very design. Strings in many languages are written to be immutable, most famously in Java. While this immutability ensures a safe and predictable outcome, it also introduces many performance concerns, such as those

mentioned earlier. Ropes help to provide a way to work with a large number of frequently modified strings without worrying about the performance drawbacks of just using the strings in Java. They deal with the small immutable fragments, and instead of copying entire strings, they reuse the existing fragments consistently. This allows the blending of the safety that the immutability of strings brings to the table with nearly optimal performance. Furthermore, memory overhead issues become far less apparent when massive amounts of editing need to be done in an immutable language, allowing them to work far more efficiently than normal string implementation.

Testing Plan

In Boehm, Atkinson, and Plass' paper they show the performance times of ropes compared to other systems. In the paper they implement the ropes as part of the C library as Cedar ropes. In their implementation the ropes have a tremendously faster running time especially when compared to other string types. My goal is to be able to implement this in a different language that being Java because it is important to harness the power of the rope along with the immutability of strings in Java.

The testing plan aims to evaluate the performance, memory saving with immutability, and the correctness of the rope compared to regular string implementation. In order to ensure that a fair and accurate comparison occurs, three primary areas of focus will be focused on throughout the testing process. The first is the performance time between the rope implementation and the string implementation. The second is to ensure that both produce identical results when asked to perform the same task. The final area of focus is to see how each implementation works under varying degrees of intensity in terms of text size and editing. This will enable the rope to be tested to its fullest extent and determine whether it does a better job than strings in most real-world applications of string editing. In the case of this testing, the operations that will be used are the ones listed above, including insert, delete, split, and concatenate. This testing will also be done in Java to ensure that we take into account the immutability of strings. We will compare the rope with the built-in Java string objects and their methods.

The first step before testing is to create three different tiers of input data. The first one will

represent a small text about a paragraph with a few hundred characters. The second will represent a medium text, which will be a few thousand characters and have a length of about a page. The final representation will be a large text with hundreds of thousands of characters, about the length of a real-world document. Each file will be in plain text only, and we will pull all three from preexisting public domain works. These files will be saved and used throughout the testing process to represent the different levels of workloads.

The next step before testing will be creating a set list of operations, such as a script that will be applied to the rope and the original string implementation. The script will go as follows

1. Insert into a position
2. Split to divide the text
3. Concatenate two parts together
4. Delete at a position
5. Repeat depending on edit density

This will be the script followed for all trials for all sizes of texts. Depending on the edit density, it will repeat these operations a different number of times. For a low editing representation, it will only run 5 times, for a medium, it will run 20 times, and for a significant representation, it will run 100 times. This will allow me to evaluate how each structure deals with the different levels of modification workloads.

Finally, when testing is ready to begin, we will run the script for the three editing types at least three times for each string implementation and each text size. This will result in 54 run trials in total. This will ensure the most accurate and precise data. For each step of the procedure, the testing will be on a 14-core MacBook Pro with an M3 Max chip and 36GB of memory. Furthermore, all measurements of time will be done with System.nanoTime() to ensure precision when it comes to time, and will use Java's built-in Runtime API to track the total memory consumption of both implementations. This will ensure consistency, fairness, and precision when collecting data for the testing. It is also important that both results are in the exact string. This is vital as if both cannot

come up with the same final string, the performance does not matter, and the structure needs to be reworked to ensure that they produce the same string. This will mean restarting the testing to ensure the Ropes are implemented correctly.

Based on everything known about Ropes, Ropes although slower and memory dense for smaller texts will be far better than the regular string implementation for medium to large texts for both memory and performance time.

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