# Chain Structure for the Representation of Large Text Files

Noah Malhi
*dept. Computer Engineering*
*Stevens Institute of Technology*
nmalhi@stevens.edu

Quentin Jimenez
*dept. Computer Engineering*
*Stevens Institute of Technology*
qjimenez@stevens.edu

Prashant Kumar
*dept. ECE*
*Stevens Institute of Technology*
pkumar14@stevens.edu

November 2022

**Abstract**

*Many data structures exist for managing text-based data. The string is the most basic, but has time-complexity limitations when inserting and deleting. Innovations upon this design include Rope, a data structure designed to circumvent the flaws inherent to the string. This project is a further iteration of the rope data structure called chain, a hybrid concept combining elements from strings and ropes to make an algorithm with optimal time-complexities for all tasks. This is accomplished by using an m-way tree with memory pools at the leaves to store the text data, allowing for quick indexing as well as insertion and deletion. These memory pools also compress the memory used, decreasing the number of pointers required to store the same information. The chain is useful in similar applications as the rope, for text-editing and data storage, and improves upon the speed and memory usage of rope.*

*Keywords: Chain, Rope, Text Files, M-ways-Tree, Optimization*

## 1 Introduction

Chain is an efficient tree data structure used to represent extremely large text files. It was innovated by Professor Dov Kruger at Stevens Institute of Technology and was created as an alternative for the Rope data structure. Originally, representation of text files was done using "string" class. "String" being an array-based objects which stores each letter in order succeeding each other in memory. This method is simple, yet has some major disadvantages. The main issue is that when the file is of a very large size, insertions in the middle become very costly. The insertion with "String" is worst-case O(n). To get past this inefficiency Hans Boehm, Russ Atkinson and Michael Plass discovered the rope data structure. In 1995 they realized their research in a paper named "Ropes: an Alternative to Strings". This discovery led to a much better way of representing large text files, but there is always room for improvement [HP95]. In this project Chain is a way that takes a lot of the same concepts of Rope, but fixes some of its own disadvantages. Where Rope is focused around inserting words in a tree, Chain focuses on inserting the lines as a whole. The purpose of this project was to create a Chain design and see how it handles increasing file size.

## 2 Ropes

The rope data structure [HP95] was created as an alternative to strings, looking to improve efficiency in areas where strings were lacking. Originally, strings were linear arrays of characters, inheriting the problems native to array editing. Letters were stored positionally, for instance to store the word "Hello", it would be converted into a char array ['H', 'E', 'L', 'L', 'O']. Since this occupies a contiguous space in memory, any edits would require copying and creating an entirely new object, making the time-complexity of any operation significantly large.

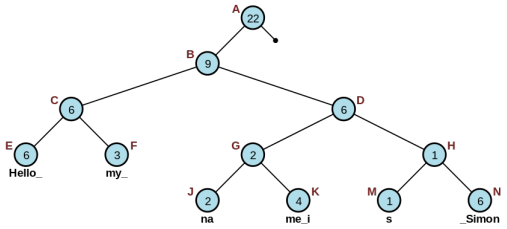


Fig. 1: Rope Structure

Ropes improve upon these inefficiencies, primarily by not using a contiguous and pre-allocated memory size. The backbone of a rope is a binary tree, allowing it to use pointers to allocate new memory when required. Each node of the tree contains a value called the weight, which represents the number of characters stored to the left of that node [HP95]. Any node that does not have children is called a leaf and contains a string. When a full string is inserted into the rope, it is divided into different sub-trees and stored in the leaves of the binary tree.

In order to retrieve a character from the rope, the position is calculated from the weights of the tree. Starting at the root node, if the position is greater than the weight, the weight is subtracted from the position and it proceeds to the right child. If it is less, it moves to the left child and maintains the same value. This is repeated until the pointer reaches a leaf node, where the position that remains can return the character requested. While this is slower than indexing a normal string, improvements are made with other functions like insertion and deletion.

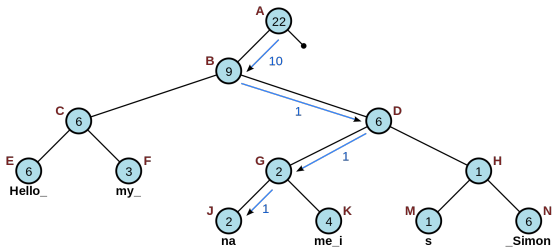


Fig. 2: Rope Structure Traversal

Insertion and deletions with ropes require two other functions to work, split and concatenation. These work hand in hand, split cuts a rope into two and returns 2 new ropes, updating the weights as it goes. Concatenation does the opposite, combining two ropes together and returning a single unified rope. To insert a new element into a rope, it is split where the insertion is desired, the new element is concatenated with one of the new ropes, and then all of the ropes are concatenated back into a single rope. While seemingly more complex than a traditional insertion with a string, the time complexity of the rope is O(logn) or O(n) at its worst, compared to the standard O(n) for a conventional string.

Deletion works similarly, but uses two splits and a concatenation instead. To delete an element from a rope, the node before the desired leaf to be deleted is split from the main rope. The leaf is then split from the new rope and then it is concatenated back to the original rope without the deleted element, re-balancing the weights as necessary. This has a time complexity of O(logn) compared to O(n) for a string, a significant processing improvement. [HP95]

Due to the speed of insertion and deletions, ropes are often used for text-editing programs, allowing changes to be made quickly. However, the rope begins to break down as the size increases, requiring pointers for almost every character in the tree. With this, the rope makes significant improvements to certain areas of string manipulation, but leaves room for improvement with future structures. [HP95]

# 3 Advantages of Chain

The chain structure is an attempt to improve upon the rope structure, aiming to further decrease the time-complexity of insertion and deletion, while also improving other methods of the algorithm. It does this by somewhat returning to the original idea of the string, using pre-allocated chunks of memory to increase efficiency. Using memory pools, characters can be represented as offsets from the beginning, decreasing the number of pointers used to store strings. Along with this, unlike rope that inserts words into nodes, Chains inserts the entire line. This results in a much smaller tree structure compared to Rope.

While it does come with advantages over the rope, it still does maintain some of the inefficiencies inherent to structures based on a single chunk of memory. If the size of the string exceeds the amount of space within the memory pool, the data will have to be copied and reallocated to a new chunk. This significantly increases the time-complexity of the algorithm, but only if the data exceeds the memory. To work around this, the initial memory chunk should be large enough to prevent having to be reallocated. Also, although maintaining lines rather than the individual words in the nodes reduces the total amount of nodes, it also makes it much more difficult to manipulate the lines. Word insertion and deletion can be

done in a simpler fashion in Rope due to its word composition. Manipulating the lines in Chain is a much tougher task that is not within the scope of this project.

This specific implementation of a chain uses an m-way tree as the base. This is a benefit in the Chain implementation as well, as it also reduces the depth of the tree. M-way trees will be discussed in the next section.

# 4 Chain Structure

## 4.1 Chain Classes

Before diving into the Chain structure it is important to discuss the node classes the make up the Chain. Displayed below is the C++ definitions for InternalNode and LeafNode. The following is then a brief description of these classes and how they are utilized.

```
class LeafNode {
    uint32_t count;
    string lines[M];

    friend class Chain;
};

class InternalNode {
    bool isLeafNode;
    int count[M] = {0};
    int nodeIndexFlag;

    InternalNode *nextNode[M];
    InternalNode *parentNode;
    LeafNode *nextLeaf[M];

    friend class Chain;
};
```

### 4.1.1 Leaf Node

The first type of node is a Leaf Node. A leaf node consists of a count and an array of M lines. M is a constant positive integer determines how many lines can be held in each leaf node, or more generally speaking how many children each node can have. The count is used to keep track of how many lines are present in the node. This number may range from 1 to M. A leaf node cannot be the root of the tree and is may only be pointed to from an Internal node.

### 4.1.2 Internal Node

An Internal Node makes up the brunt of the Chain structure. The root of the Chain is always an Internal node and it may point up to M number of Internal Nodes or M number of Leaf Nodes. Each Internal node also keeps track of its parent node as well as its index in the parent node for back-tracing purposes that will be mentioned later. The Boolean isLeafNode is utilized to help traverse down the Chain to reach an InternalNode that points to leaf nodes.

## 4.2 M-ary trees

A Chain based M-ary tree constitutes of a root node with M nodes and M child nodes. A binary tree is the special case where M = 2, and a ternary tree is another case with M = 3 that limits its children to three. The child nodes could be internal nodes or leaf nodes. Internal Nodes are rooted at the Root Node or other Internal Nodes. Leaf Nodes branch out from the Internal Nodes and they in turn have M pointers to Strings. These strings are complete sentences as opposed to words of a fixed length in case of a Rope data structure. Some primitive rules for M-ary trees can be laid out as follows:

The root node points to internal nodes or leaf nodes and the node elements represent the number of leaf nodes pointing to strings. The root node can point to M internal nodes or M Leaf nodes each pointing to M elements. Pointers to strings are only held at the leaf nodes. A perfect M-ary tree is a one which has all its leaf nodes full and at the same level. The height of a perfect M-ary tree given by h can be calculated as h=logMn, where n represents the total number of lines to be held by the leaf nodes. Internal node is an internal node pointing to leaf node or another internal node. ParentNode is a parent to the leaf node or internal node. The parentNode could either be an internal node or a root node.

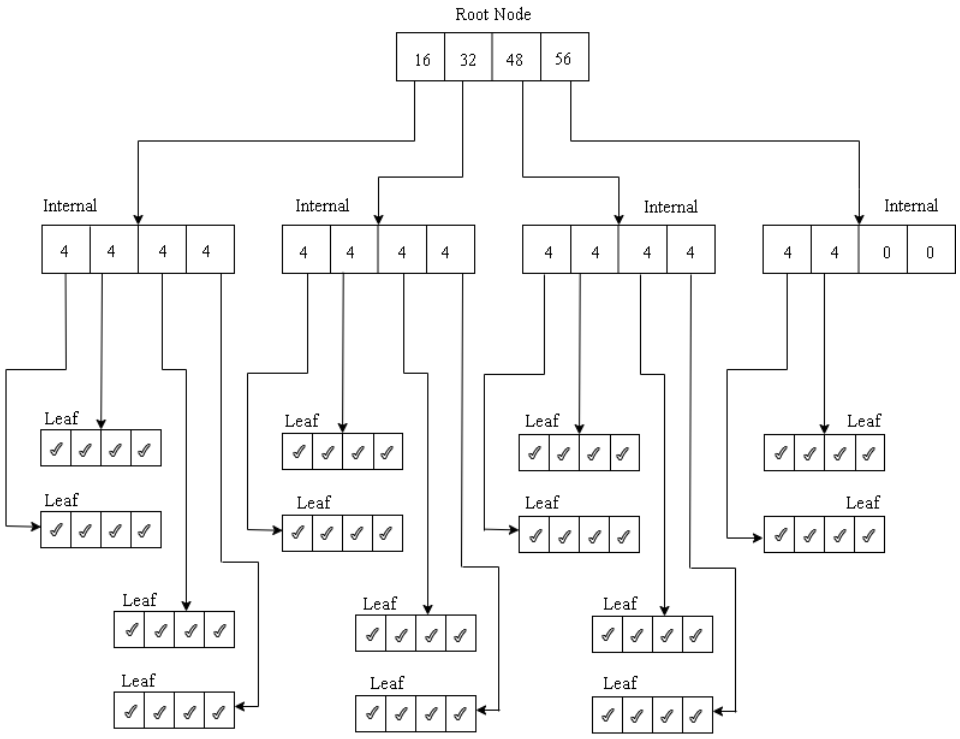


Fig. 3: Chain Data Structure using a Four Way Tree

Fig. 3 represents a Chain data structure with M=4. The tree's height is three and has a root node, internal nodes, and leaf nodes. The root node points to four internal nodes namely internal1, internal2, internal3, and internal4. These internal nodes point to various leaf nodes which finally point to strings. The elements in the root node carry the count of the leaves which point to strings. Similarly, the elements in the internal node too keep a count of the leaves pointing to strings. The consecutive root element keeps a cumulative count of the non-empty leaf nodes. This helps in insertion at a particular position without having to add all previous root nodes values to arrive at the correct node for insertion. The last root node element's count represents the total number of strings in the tree structure.

# 5 Insertion

Insertion is a basic functionality to any data structure. The insertion function was split further into three sub-functions, namely beginning, end, and middle. These methods will be discussed in detail in the upcoming sub-sections.

## 5.1 Split Functionality

To achieve insertion at any user defined input, we need the ability to increase the tree height. Providing node split functionality to nodes helps achieve this aim. The function inputs to the node split function are the nodes themselves. If the root node is to be split, it is done by splitting the root node further into M internal nodes as depicted in Fig. 3. Internal nodes are split further into M internal nodes as shown in Fig. 4. The values are copied from the parent node into the first sub-node formed by splitting. The count of the parentNode is then updated by backtracking (backtrace).

## 5.2 Beginning Insertion

Insertion at the beginning of the tree is achieved by checking for the count in the root node. If the root node count at the first place (index = 0) is equal to M raised to (h1), it implies that the first of the M segments is full. To create additional space, the internal node at level h-1 is split into M sub-internal nodes. The values are then copied to the next right index of the leaf nodes. An example of insertion at the beginning is depicted in Fig. 4. M has been set to 4, yielding M internal nodes from the root node namely internal1, internal2, internal3, and internal4. Assuming an beginning state of Fig. 3, the initial position is at the root node and the tree is traversed until the last layer of internal nodes i.e internal1. As internal1 is full, internal1 is split into internal11 and internal12. It is to be noted

here that setting M=4 enables the ability to have four internal nodes originating from internal1. After creating the new internal nodes, the values in the leaf nodes are copied one place to the right beginning from the last value in the leaf nod. This creates a leaf node at the first position of internal11 pointing to nullptr. The new value is then inserted at the first leaf node. Insertion at the beginning is of the complexity O(logn+ M2). For large text files M2 is negligible, hence the overall amortized complexity is O(logn).

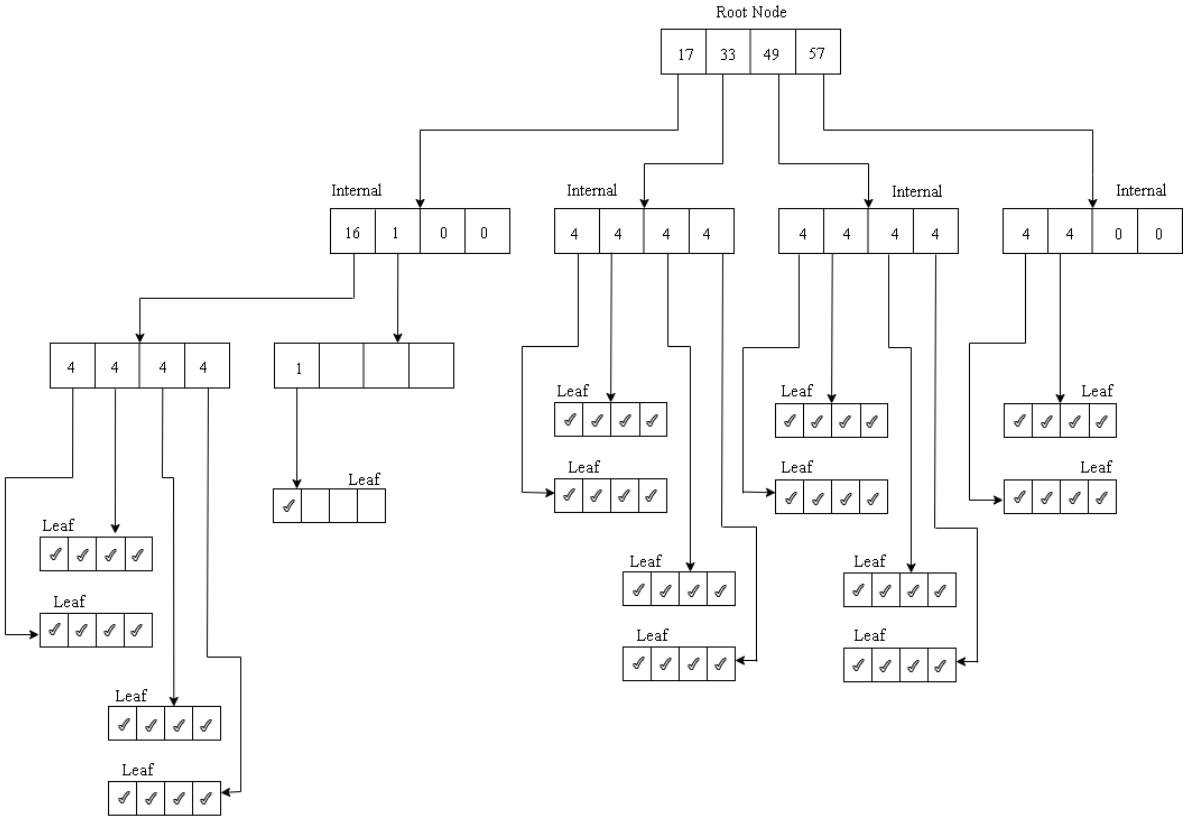


Fig. 4: Chain Insert Beginning

## 5.3 End Insertion

To insert at the end, the last non-full node needs to be found. It starts at the root node checking from position 0 to position M. Fig. 3 represents a structure where adding to the end occurs. Again, the basis for referencing the End Insertion can be assumed as Fig. 3, which is used as a base structure. If End Insertion is the very first insertion into the Chain, the root node would be a single InternalNode that has M=4 leaf nodes and its parentNode is set to nullptr. The leaf nodes contain up to four lines each thus at most one InternalNode contains 16 lines. If all of slots are full, a split operation would then take place at the Mth position (due to end insert). In the case of splitting for an end insertion, four new internal nodes are created with the node being split set as the head. The first node farthest to the left maintains the values of the head node and its parentNode is set to be equal to the head. The second node has the new line inserted into nextNode[0] and the parentNode is also pointed towards the new head. The third and fourth internalNodes are created but left empty with their parentNode being set to head. An important step in this split is to also set the indexFlag for all of the nodes in which this flag is the nodes position from the head node. InternalNode node1 would be give the index of 0, node2 1, and so on. Setting the parentNode and setting the index is especially important to the chain structure. This is because after the split of the node and the insertion of the line, a backtrace needs to be done from the newly added nodes back to the top of the tree. This is done to update the counts of each node up until the top. This gives the data structure an accurate count of full leaf nodes in the tree and the ability to access specific line numbers.

### 5.4 Middle Insertion

Fig. 5 shows the process of insertion in the middle of the tree. This function takes two arguments, an integer user argument as the position and the string to be inserted. Ten lines of string are inserted at position 25 in the base structure in Fig.3. The process of middle insertion is like that of insertion at the beginning of the tree.

The process is started by looking through the root node with the position to insert. As discussed above, the root node carries the cumulative count of full leaf nodes. Since 25 is greater than 16 and less than 32, it indexes into the second internal node internal2. The program then looks for empty leaf nodes within internal2. Since there are none, internal2 is split into internal21 and internal22 using the split node function.

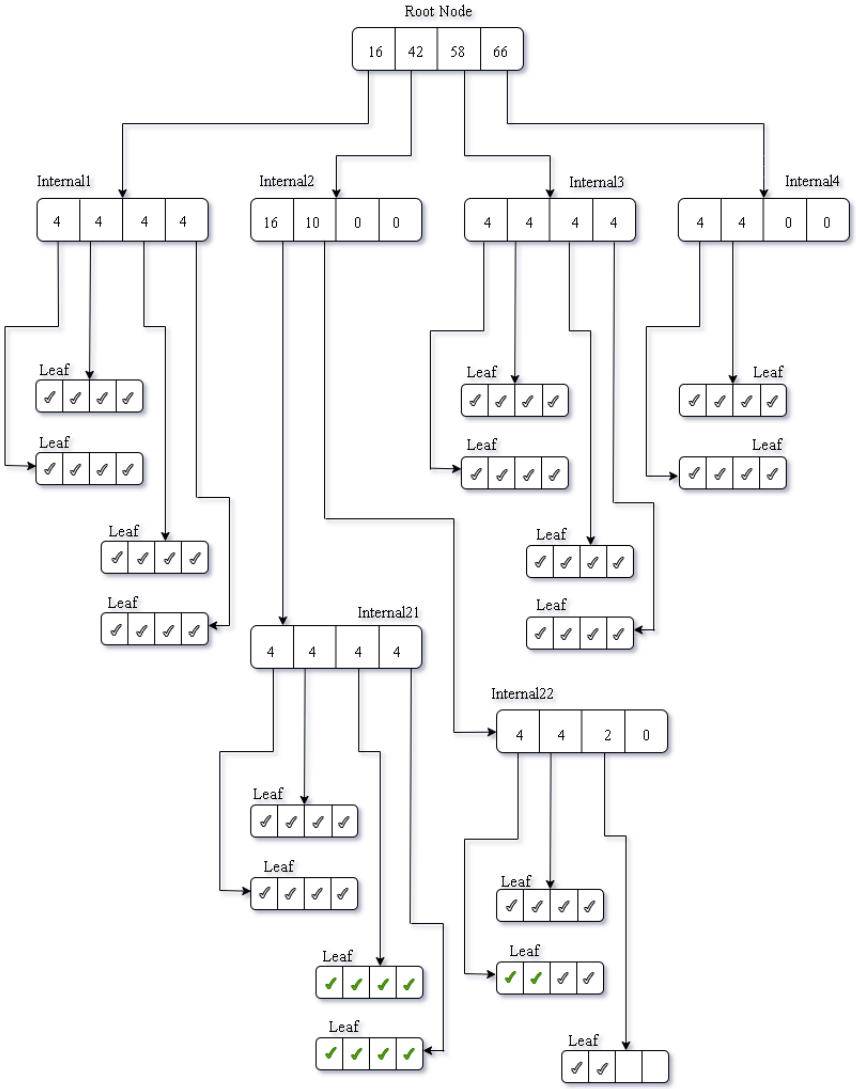


Fig. 5: Chain Insert Middle

After creating the new internal nodes, the values in the leaf nodes are copied one place to the right beginning from the last value in the leaf node. This process creates an empty leaf node at the first location of internal21. The new values are then inserted into the newly emptied leaf node. The same process is repeated ten times to insert all strings into position 25.

## 6 Deletion

Fig. 6 represents a sample of deleting five lines at position 40. First, it is necessary to locate position 40 based on the cumulative count stored in root node. Since 40 is greater than 32 and less than 48, the algorithm traverses to internal3 which holds the leaf node at position 40. The

pointer from the leaf node is set to nullptr and all the remaining leaf nodes are shifted one place to the left starting from the left leaf node. This process creates an empty leaf node at the end of the internal node internal3. The count of the internal nodes and root nodes are changed using the backtrace function. A similar process is repeated five times and the respective internal nodes and root nodes are updated.

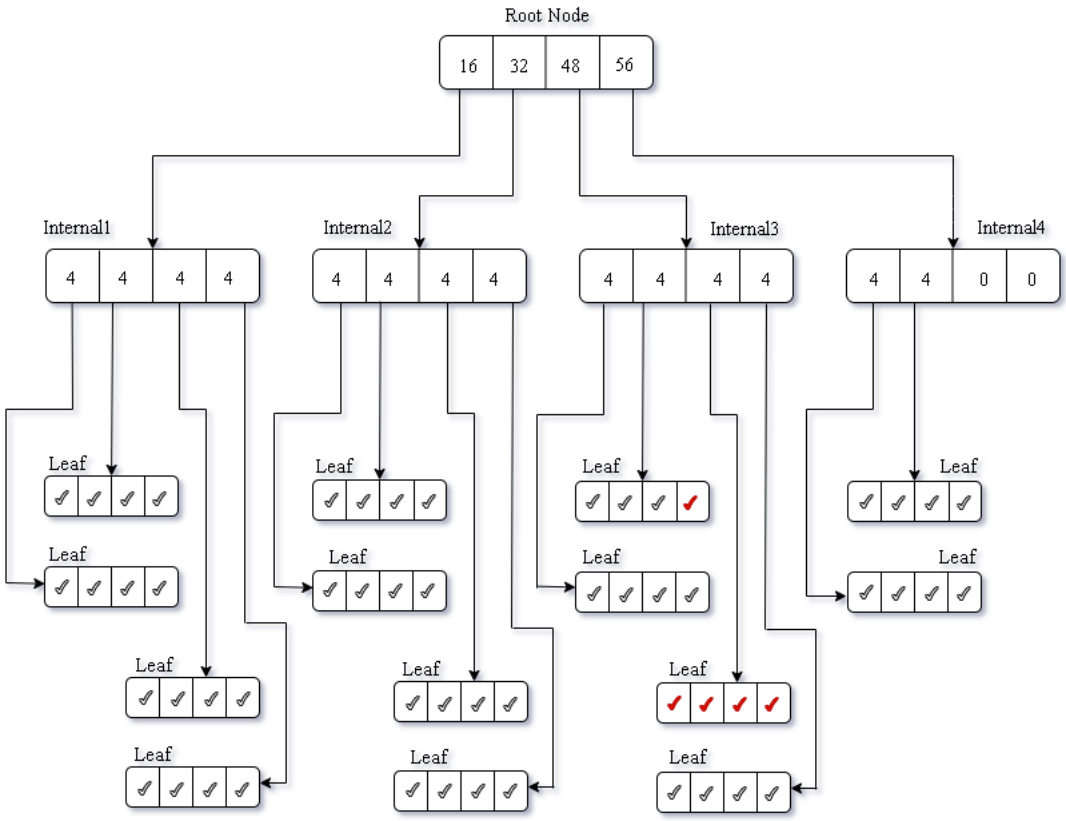


Fig. 6: Chain Remove

# 7 Future Optimizations

## 7.1 Memory Pools

Initially in the scope of this project was the use of Memory Pools with the Chain structure. Memory pools are allocated blocks of memory that use dynamic allocation [PhD16]. By allocating a large amount of memory beforehand saves a lot of time overall and allocating individual chunks of memory is faster than the typically C++ "new" operator. It is faster than new as there is no added overhead for allocating these chunks while using the memory pool. This of course is with the exception of running out of space in the initial allocation. When it comes to memory, it also is safe from fragmentation in which memory is continuously allocated and deallocated resulting in the memory being split up with small gaps. Lastly another advantage is that the pointer size within the pool is 32 bits rather than the normal 64 bits.

The Chain structure would make use of memory pools in two different ways. The first being the aforementioned benefits of the memory pool. The second reasoning would be the ability to index into the memory pool for fast access times. The proposed solution when implementing the Chain structure is to insert each line into the memory pool while maintaining its offset into memory. When the line needs to be retrieved, all that is needed is the offset to locate the line in memory.

## 7.2 Tree Balancing

An important concept not covered in this projects scope was the ability to balance the tree. Balancing is an important functionality because it can help reduce the time to traverse the tree. If one side of the tree contained the majority of the

lines, then the tree structure would not be utilized to its fullest potential as the traversal from the root to the leaves would be greater. There are a few different situations that require the tree to be balanced. First off when reading in a large file. In this case insertEnd() is used to insert all of the lines in order. With very large text files the Chain structure will continuously be split at its furthest right node as the structure is filled. This results in a very unbalanced tree in which the right side is over-saturated. Another common case where balancing may be needed is when lines are removed from the Chain. If a large amount of lines are removed, such as 1000 lines, this may leave the tree unbalanced.

When it comes to balancing there are quite a few struggles that come into the scene. First of all, there needs to be a way to maintain the order of the tree when a balancing action occurs. If insertEnd is called and the tree is right side heavy, the tree needs to redistribute but also keep the order that the leaves are in. Without this, the ability to correctly find a specific line number to insert, remove or search is completely impossible unless each node in the tree is search for the line.

One potential change that can be made in the future it to create the Chain data structure using B-Trees. B-trees are very similar to M-way-trees except for important factors. B-trees have the ability to balance itself and every node in the tree is at least half full [AB94]. In the case of the Chain structure that would mean that each leafNode that exists would have to be at least half full, M / 2 lines. Initially this research utilized b-tree's but it was not in the correct adaptation for Chain nor in its scope. However, a B-tree implementation is very much a potential solution to creating a balanced Chain structure.

# 8 Conclusion

Data structures have always evolved and been built upon each other, and the chain is no different. First there was the string, which allowed for sequential storage of text. While simple to implement, the string is highly inefficient when it comes to editing, especially with insertions and deletions. The rope builds on this design, using a binary tree to store text data in pieces. Using non-contiguous memory the rope is able to improve in areas where the string lacks, working faster when inserting and deleting text. Although it is faster in these areas, it is still outperformed by the string when doing certain function like indexing.

The chain takes concepts from both designs and builds upon them to improve on the inefficiencies in their structure. Instead of a binary tree like the rope, the chain uses an m-way tree to reduce the pointers required to represent the same amount of text. However, it utilizes a contiguous block of memory similar to a string through memory pools, allowing it to be indexed faster than a rope. Both of these combine to increase efficiency in all areas.

This project represents the current progress toward the chain concept, and with further implementation the data structure may become fully realized. There were many difficulties encountered, one of the biggest was deciding which structure to base the chain on. Initially, attempts were made to use a b-tree instead, but it did not function well within the scope of this design. There were also difficulties in the implementation of the memory pool in conjunction with the m-way tree, which lead to it being removed from the final version of this project.

Overall the chain concept still has a way to go before it is complete. Fixing of the multi-level functionalities needed to really complete the Chain. In its current status, only 64 lines are able to be written in the middle whereas for Insert End it is working completely. The insert end function is able to find the end position and insert. and also it has the ability to split numerous times if the leaf nodes are full. Insert Middle, Insert End, and remove are all working on this basis. Implementation of the memory pools would be next, allowing the chain to be indexed more efficiently. While good progress was made on this design, there is still much more to be done to complete the Chain.

# References


[AB94] B. B. Madan A. Varsheya and M. Balakrishnan. *Concurrent search and insertion in K-Dimensional Height Balanced Trees.* Proceedings of 8th International Parallel Processing Symposium, 1994. DOI: 10.1109/IPS.1994.288202.

[HP95] R. Atikinson H.J Boehm and M. Plass. *Ropes: an Alternative to Strings.* Vol. 25(12). SOFTWARE-PRACTICE AND EXPERIENCE. 1995.

[PhD16] Benjamin Qochuk PhD. *Memory Pool with C++ Implementation.* 2016. URL: https://iq.opengenus.org/memory-pool/. (accessed: 12.14.2022).