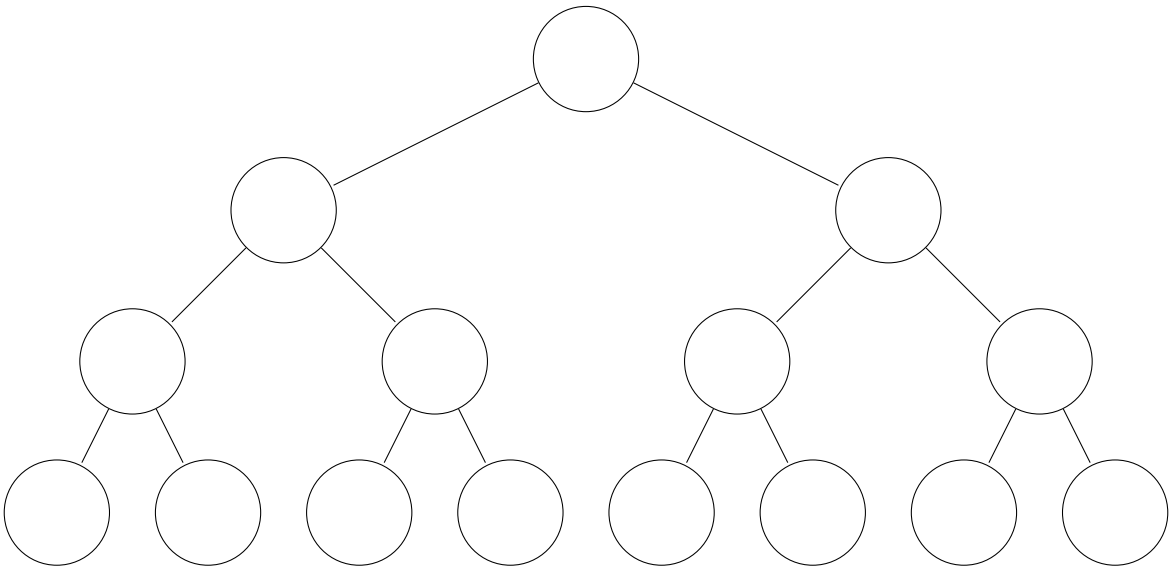
Self-Balancing Text String Trees

CPSC 490 Senior Project

Billy Zhong



Computer Science
Yale University
Advisor: Michael Fischer
May 10, 2022

1 Introduction

In a June 1979 technical report, Michael Fischer and Richard Ladner introduced the problem of maintaining "sticky pointers" within packed linked-list implementation of text strings [1].

Abstractly, a text string is a mutable string which supports two operations: INSERT and DELETE. INSERT allows a new string to be inserted at a particular index in the text string, while DELETE removes a specified amount of characters from the text string beginning at a particular index. A sticky pointer is an object that points to a particular character in a text string and continues to "stick" to that character even after the text string has been modified and the character has changed index. At any point in time, a sticky pointer can be used to find the current index of the character that it points to, if it still exists.

The Fischer-Ladner technical report describes a novel data structure to update and maintain sticky pointers on a text string, which we call a **sticky pointer forest**. The data structure implements a text string as a forest of binary trees whose leaves are linked together linearly. Each node in the forest represents a distinct substring of the text string and a sticky pointer is a tuple containing a pointer to a node and the position of the character within the substring represented by the node. The sticky pointer forest supports the INSERT and DELETE operations of text strings as well as provides an algorithm called FIND to return the updated node and position of the character stuck by a sticky pointer. The Fischer-Ladner report shows that the worst-case runtime of FIND is *linear* in the number of INSERT and DELETE operations performed on the text string.

In Fall 2021, we augmented the sticky pointer forest using elements of self-balancing binary trees in order to achieve a worst-case runtime for FIND that is *logarithmic* in the number of INSERT and DELETE operations performed on the text string. We referred to this extended data structure as the self-balancing sticky pointer forest. However, this data structure did not support the efficient translation between node-position pairs and the indices of the character which they pointed to. More specifically, the translation required an asymptotic runtime that was *linear* in the number of INSERT and DELETE operations. This means that there was no efficient way to create sticky pointers from indices of characters or get the index of a character pointed to by a sticky pointer.

In this report, we will extend self-balancing binary search trees to form a novel data structure to maintain text strings and sticky pointers more efficiently. In addition to INSERT, DELETE, and FIND, this data structure will also include two new operations to translate between node-position pairs and indices of characters, which will have a runtime that is *logarithmic* with respect to the number of INSERT and DELETE operations. With these new operations, the data structure will not only maintain the same logarithmic runtimes for INSERT and DELETE, but also allow for the logarithmic creation of sticky pointers from indices of characters and a modified, logarithmic FIND which returns the index of the stuck character, rather than a node-position pair. In the following section, we will describe the semantic foundations of our new data structure.

2 Text Strings and Characters

2.1 Definitions

A **segment** is an ordered pair of an ID and a string. For a segment s, we refer to its ID using the notation $\hat{s}$ and its string using the notation $\overline{s}$. Two segments are considered **equivalent** if they have equal IDs and lexicographically equivalent strings. A **text string** is a linearly ordered list of segments $S = [s_0, s_1, \ldots, s_{n-1}]$. The **string representation** $\overline{S}$ of a text string S is the concatenation the strings of the segments in order — that is, $\overline{S} = \overline{s_0}\ \overline{s_1} \ldots \overline{s_{n-1}}$. The **offset** of a segment $s_k \in S$ describes the total length of the strings of the segments that precede s_k in S. It is mathematically defined as

$$\textsc{Offset}(s_k) = \sum_{j=0}^{k-1} |\overline{s_j}|$$

A **character** within a text string is an ordered pair (s_k, p), such that $s_k \in S$ is a segment and p is a position within the string of the segment such that $0 \leq p < |\overline{s_k}|$. We say that a character (s_k, p) is a character of the segment s_k in S. Two characters $(s_k, p), (s_l, q)$ considered **equivalent** if $s_k = s_l$ and $p = q$. The **absolute index** of a character in S describes the index of the character within the string representation $\overline{S}$. For a character (s_k, p), the absolute index is defined as

$$\textsc{Abs}(s_k, p) = \textsc{Offset}(s_k) + p$$

The **displacement** between two characters in S describes the difference between their absolute indices. If (s_k, p) and (s_l, q) are characters of S, then

$$\begin{aligned}\textsc{Displacement}((s_k, p), (s_l, q)) &= \textsc{Abs}(s_l, q) - \textsc{Abs}(s_k, p) \\ &= (\textsc{Offset}(s_l) + q) - (\textsc{Offset}(s_k) + p)\end{aligned}$$

Additionally, we extend the displacement definition to also define the displacements between two segments and between a segment and a character. The displacement between two segments is equivalent to the displacement between the first characters of the two segments. Similarly, the displacement between a character and a segment is equivalent to the displacement between the character and the first character of the segment. Notice that because the first character of a segment s_k is given by $(s_k, 0)$, the absolute index of the first character is exactly the offset of s_k and so the displacement between two segments $s_k, s_l \in S$ is given by

$$\begin{aligned}\textsc{Displacement}(s_k, s_l) &= \textsc{Displacement}((s_k, 0), (s_l, 0)) \\ &= \textsc{Abs}(s_l, 0) - \textsc{Abs}(s_k, 0) \\ &= \textsc{Offset}(s_l) - \textsc{Offset}(s_k)\end{aligned}$$

Similarly, the displacement between a character (s_k, p) and a segment $s_l \in S$ is given by

$$\begin{aligned}
\textsc{Displacement}((s_k, p), s_l) &= \textsc{Displacement}((s_k, p), (s_l, 0)) \\
&= \textsc{Abs}(s_l, 0) - \textsc{Abs}(s_k, p) \\
&= \textsc{Offset}(s_l) - (\textsc{Offset}(s_k) + p) \\
\textsc{Displacement}(s_l, (s_k, p)) &= \textsc{Displacement}((s_l, 0), (s_k, p)) \\
&= \textsc{Abs}(s_k, p) - \textsc{Abs}(s_l, 0) \\
&= (\textsc{Offset}(s_k) + p) - \textsc{Offset}(s_l)
\end{aligned}$$

2.2 Operations on Text Strings

2.2.1 Split

Given a segment $s_k \in S$ and an index p such that $0 < p < |\overline{s_k}|$, $\textsc{Split}(S, s_k, p)$ creates two new segments s_L and s_R such that the following conditions are satisfied:

1. The IDs of s_L and s_R are encoded (deterministically and uniquely) to reflect the fact that they are left and right halves of s_k — that is, $\hat{s_L} = \langle \hat{s_k}, L \rangle$ and $\hat{s_R} = \langle \hat{s_k}, R \rangle$.
2. The strings of s_L and s_R partition the string of s_k at index p — that is, $\overline{s_k} = \overline{s_L}\ \overline{s_R}$ where $|\overline{s_L}| = p$

and returns a new text string

$$S' = [s_0, \ldots, s_{k-1}, s_L, s_R, s_{k+1}, \ldots, s_{n-1}]$$

Notice that the string representation of the new text string is lexicographically equivalent to the string representation of the original text string — that is, $\overline{S} = \overline{S'}$.

2.2.2 InsertBefore

Given an existing segment $s_k \in S$ and a segment to be inserted s', $\textsc{InsertBefore}(S, s_k, s')$ returns a new text string S' that is equivalent to S with s' immediately preceding s_k.

$$S' = [s_0, \ldots, s_{k-1}, s', s_k, \ldots, s_{n-1}]$$

2.2.3 InsertAfter

Given an existing segment $s_k \in S$ and a segment to be inserted s', $\textsc{InsertAfter}(S, s_k, s')$ returns a new text string S' that is equivalent to S with s' immediately following s_k.

$$S' = [s_0, \ldots, s_k, s', s_{k+1}, \ldots, s_{n-1}]$$

2.2.4 Delete

Given an existing segment $s_k \in S$, $\textsc{Delete}(S, s_k)$ returns a new text string S' that is equivalent to S with s_k deleted from the list.

$$S' = [s_0, \ldots, s_{k-1}, s_{k+1}, \ldots, s_{n-1}]$$

3 Histories

3.1 Definition

A **history** is a pair of sequences, one containing text strings and one containing operations, that track the transformation of a text string over time. Defined inductively, a history $H = ([S_0, S_1, \dots S_m], [\sigma_0, \sigma_1, \dots, \sigma_{m-1}])$ is a pair of sequences such that such that:

1. $S_0 = [\ ]$ is the empty text string
2. An operation σ_i is one of the operations on text strings defined above in section 1.2 or a no-op operation.
3. S_{t+1} is the result of applying the operation σ_t to the text string S_t.

We call an element S_t of H the **version** of the text string at time t. Additionally, we refer to the last element of the sequence S_m as the **most recent version** of the text string. The operation σ_t that transforms S_t to S_{t+1} is known as the operation performed at time t. As a small caveat, we require that the only operation that can be performed on an empty text string $S_t = [\ ]$ is an $\textsc{Insert}(S_t, s')$ operation, which merely populates S_t and returns $S_{t+1} = [s']$. Additionally, if s' is inserted into S_t, we require that s' have an ID that is distinct from every existing segment in S_t.

3.2 Character Equivalence across Time

Now that we have developed the notion that text strings are mutable strings that transform over time due to text string operations, we will formulate the concept of character equivalence across time.

3.2.1 Split

If the operation $\textsc{Split}(S_t, s_k, p)$ is performed at time t on the text string

$$S_t = [s_0, \dots, s_{k-1}, s_k, s_{k+1}, \dots, s_{n-1}]$$

to transform the text string into

$$S_{t+1} = [s_0, \dots, s_{k-1}, s_L, s_R, s_{k+1}, \dots, s_{n-1}]$$

then the character (s_l, q) in S_t is **equivalent across time** to the following character in S_{t+1}, given by

$$(S_t, s_l, q) \approx \begin{cases} (S_{t+1}, s_l, q) & \text{if } s_l \neq s_k \\ (S_{t+1}, s_L, q) & \text{if } s_l = s_k \text{ and } 0 \leq q < p \\ (S_{t+1}, s_R, q-p) & \text{if } s_l = s_k \text{ and } p \leq q < |\overline{s_l}| \end{cases}$$

We use the notation (S_t, s_k, p) to denote the character (s_k, p) in S_t. No characters are inserted or deleted in a $\textsc{Split}$ operation.

3.2.2 InsertBefore

If the operation InsertBefore(S_t, s_k, s') is performed at time t on the text string

$$S_t = [s_0, \ldots, s_{k-1}, s_k, s_{k+1}, \ldots, s_{n-1}]$$

to transform the text string into

$$S_{t+1} = [s_0, \ldots, s_k, s', s_{k+1}, \ldots, s_{n-1}]$$

then the character (s_l, q) in S_t is equivalent to the following character in S_{t+1}, given by

$$(S_t, s_l, q) \approx (S_{t+1}, s_k, p) \text{ for all characters } (s_l, q) \text{ in } S_t$$

Additionally, the characters (S_{t+1}, s', q) for $0 \leq q < |s'|$ are introduced at time $t+1$. The same equivalences are true for the operations Insert and InsertAfter.

3.2.3 Delete

If the operation Delete(S_t, s_k) is performed at time t on the text string

$$S_t = [s_0, \ldots, s_{k-1}, s_k, s_{k+1}, \ldots, s_{n-1}]$$

to transform the text string into

$$S_{t+1} = [s_0, \ldots, s_{k-1}, s_{k+1}, \ldots, s_{n-1}]$$

then the character (s_l, q) in S_t is equivalent to the following character in S_{t+1}, given by

$$(S_t, s_l, q) \approx (S_{t+1}, s_l, q) \text{ if } s_l \neq s_k$$

The characters (S_t, s_k, p), where $0 \leq p < |\overline{s_k}|$, are deleted and do not exist at time $t+1$.

3.3 Segment and Character Relations across Time

For $t_a \leq t_b$, a segment $s_l \in S_{t_b}$ is considered a **sub-segment** of a segment $s_k \in S_{t_a}$ if for all characters (S_{t_b}, s_l, q) where $0 \leq q < |\overline{s_l}|$, there exists a character (S_{t_a}, s_k, p) with $0 \leq p < |\overline{s_k}|$ such that $(S_{t_a}, s_k, p) \approx (S_{t_b}, s_l, q)$. Less formally, s_l is a sub-segment of s_k if s_l was derived from s_k through a series of Split operations.

The **initial segment** of a segment $s_l \in S_{t_b}$ is the segment $s_k \in S_{t_a}$ such that the following conditions are true:

1. s_k was introduced at time t_a by an Insert operation performed on S_{t_a-1} at time $t_a - 1$. Equivalently, s_k is not a sub-segment of any segment but itself.
2. s_l is a sub-segment of s_k

The **original instance** of a character is the earliest representative of the equivalence class of that character, induced by the character equivalence across time equivalence relation. More specifically, the original instance of the character $(s_l, q)_{t_b}$ in a history H is the character (S_{t_a}, s_k, p) such that the following conditions are true:

1. s_k is the initial segment of s_l
2. $(S_{t_a}, s_k, p) \approx (S_{t_b}, s_l, q)$

4 Sub-histories

4.1 Definition

Let s' be a segment inserted at time t' in history H. The **sub-history** $H_{\hat{s'}}$ is a history that tracks the transformation of a text string containing only s' and its sub-segments over time through SPLIT operations. $H_{\hat{s'}}$ is defined with respect to H as follows:

1. If history $H = ([S_0, \ldots, S_m], [\sigma_0, \ldots, \sigma_{m-1}])$ contains m versions and $m-1$ operations, then sub-history $H_{\hat{s'}} = ([R_0, \ldots, R_m], [\pi_0, \ldots, \pi_{m-1}])$ also contains m versions and $m-1$ operations.

2. No operation is performed in $H_{\hat{s'}}$ prior to time $t'-1$ — that is, π_t is no-op and $R_t = [\,]$ for $0 \leq t < t' - 1$.

3. At time $t'-1$, an INSERT$(R_{t'-1}, s')$ operation is performed in $H_{\hat{s'}}$ to yield $R_{t'} = [s']$.

4. For time $t \geq t'$, if σ_t is a SPLIT(S_t, s, p) operation and s is a sub-segment of s', then π_t is a SPLIT(R_t, s, p) operation. If σ_t is any other operation, π_t is no-op.

4.2 Properties

Proposition 1. Let s' be a segment inserted at time t' in history H. For $t \geq t'$ in sub-history $H_{\hat{s'}} = ([R_0, \ldots, R_m], [\pi_0, \ldots, \pi_{m-1}])$, the string representation of R_t is exactly the string of s' — that is, $\overline{R_t} = \overline{s'}$.

Proof. We will show the proposition through induction over time. First, recall from the definition of the sub-history $H_{\hat{s'}}$ that $R_{t'} = [s']$. Thus at time t', we know that $\overline{R_{t'}} = \overline{s'}$. Now let $t \geq t'$ be arbitrary and suppose that $\overline{R_t} = \overline{s'}$. From the definition of the sub-history $H_{\hat{s'}}$, π_t can be one of two operations:

> *Case 1.* Suppose that π_t is no-op. Then it follows that $R_{t+1} = R_t$ and thus $\overline{R_{t+1}} = \overline{R_t} = \overline{s'}$.
>
> *Case 2.* Suppose that π_t is a SPLIT operation on R_t. Recall from the definition of the SPLIT operation (see section 1.2.1) that the string representation of the resulting text string is lexicographically equivalent to the string representation of the original text string. Thus, $\overline{R_{t+1}} = \overline{R_t} = \overline{s'}$.

So we have shown exhaustively that $\overline{R_{t+1}} = s'$. Thus by induction, it follows that for $t \geq t'$ in sub-history $H_{\hat{s'}}$, $\overline{R_t} = \overline{s'}$. □

Proposition 2. Let s' be a segment inserted at time t' in history H. For $t \geq t'$ with history $H = ([S_0, \ldots, S_m], [\sigma_0, \sigma_1, \ldots, \sigma_{m-1}])$ and sub-history $H_{\hat{s'}} = ([R_0, \ldots, R_m], [\pi_0, \ldots, \pi_{m-1}])$, if $s \in S_t$ is a sub-segment of s', then $s \in R_t$.

Proof. We will show the proposition through induction over time. At time $t = t'$, there is only one sub-segment of s' in $S_{t'}$: s' itself. From the construction of the sub-history $H_{\hat{s'}}$, we know that $R_{t'} = [s']$ and thus $s' \in R_{t'}$ as well. Now let $t \geq t'$ be arbitrary and suppose if $s \in S_t$ is a sub-segment of s', then $s \in R_t$. Let us consider the possible operations for σ_t:

Case 1. Suppose that σ_t is an INSERT operation or a DELETE or SPLIT operation on a segment that is not a sub-segment of s', or no-op. Then the set of segments in S_{t+1} that are sub-segments of s' is no different from the set of segments in S_t that are sub-segments of s'. Additionally, π_t must be no-op by construction and thus the set of segments in R_{t+1} that are sub-segments of s' is no different from the set of segments in R_t that are sub-segments of s'. Let $s \in S_{t+1}$ be an arbitrary sub-segment of s'. s must necessarily also be in S_t. By the inductive hypothesis, we know that $s \in R_t$. Thus, $s \in R_{t+1}$.

Case 2. Suppose that σ_t is a DELETE operation on a sub-segment of s'. Then S_{t+1} has one less sub-segment of s'. π_t is no-op by construction. Thus, R_{t+1} has the same sub-segments of s' as R_t. Let $s \in S_{t+1}$ be an arbitrary sub-segment of s'. s must necessarily also be in S_t. By the inductive hypothesis, we know that $s \in R_t$. Thus, $s \in R_{t+1}$.

Case 3. Suppose that σ_t is $\text{SPLIT}(S_t, s, p)$ and s is a sub-segment of s'. Then the added s_L and s_R are sub-segments of s'. $\pi_t = \text{SPLIT}(R_t, s, p)$ by construction, and so $s_L, s_R \in R_{t+1}$ as well.

So we have shown exhaustively that if $s \in S_{t+1}$ is a sub-segment of s', then $s \in R_{t+1}$. Thus by induction, it follows that for $t \geq t'$, if $s \in S_t$ is a sub-segment of s', then $s \in R_t$. □

5 Sticky Pointers

5.1 Definition

A **sticky pointer** to a character is an object that can be used to identify the character in the most recent version of the text string that is equivalent to the stuck character. A sticky pointer to the character (S_{t_b}, s_l, q) in a history H is the ordered pair $(\hat{s_k}, p)$ where (S_{t_a}, s_k, p) is the original instance of (S_{t_b}, s_l, q). We say that a character is pointed to or stuck by a sticky pointer $(\hat{s_k}, p)$ if that character is equivalent to the character (S_{t_a}, s_k, p).

5.2 Operations

5.2.1 Create

Given a text string in a history along with its corresponding sub-histories and a character in the text string, CREATE returns a sticky pointer to the character. More specifically, if (S_{t_b}, s_l, q) is the character in text string S_{t_b}, then $\text{CREATE}(S_{t_b}, s_l, q)$ returns the sticky pointer $(\hat{s_k}, p)$ such that (S_{t_a}, s_k, p) is the original instance of (S_{t_b}, s_l, q).

5.2.2 Find

Given a sticky pointer, FIND returns the character pointed to by the sticky pointer within the most recent version of the text string, if it exists. More specifically, given sticky pointer $(\hat{s_k}, p)$, $\text{FIND}(\hat{s_k}, p)$ returns (S_m, s_l, q) where (S_m, s_l, q) is a character in the most recent version of the text string S_m and $(S_t, s_k, p) \approx (S_m, s_l, q)$. If there is no such character, FIND

returns the information that the character does not exist in the most recent version of the text string.

6 Implementation Details

Now that we have established the semantic framework of our data structure, we will discuss how the data structure and corresponding operations are specifically implemented. Please refer to the Appendix for any pseudocode.

6.1 Segment Nodes

The most basic unit of our novel data structure is the segment node. Segment nodes represent the segments found in text strings. Each node contains a field for the ID and a field for the string of the segment.

6.2 Characters

We represent characters of segments with an object containing a field for a pointer to a segment node and a field for the position of the particular character within that segment. We say a character is invalid or does not exist if its segment pointer is null.

6.3 Text String Trees

Text string trees are binary trees consisting of segment nodes, which represent text strings. Each segment node in a text string tree has a link to a parent node, a left child, and a right child. Additionally, each link has an associated weight, which represents the displacement between the current node and its parent, left child, and right child, respectively. Text string trees are characterized by two key properties:

1. Traversing the text string tree in-order yields the segment nodes in the order specified by the text string.

2. The total weight of the path from the segment node representing segment s_k to the segment node representing the segment s_l is equal to the displacement of s_l from s_k, or $\textsc{Displacement}(s_k, s_l)$.

Additionally, text string trees do not maintain a reference to the root of the tree, but rather maintain a reference to the segment node representing the head segment of the text string. Because of the ordering of the text string tree, this is also known as the left-most segment node in the tree.

6.4 Histories and Sub-histories

A history and its corresponding sub-histories are implemented as a singular object. The object contains the most recent version of the text string of the history, as well as a map

from an ID to the most recent version of the text string of the sub-history corresponding to that ID. Additionally, each segment node in the text strings of the history and the sub-histories are given a new *dual* link. This links a segment node in the text string of the history to its equivalent node in the text string of the corresponding sub-history and vice versa (if it exists). The two key properties of the history object are as follows:

1. Every segment node in the most recent version of the text string of the history has a non-null dual link to the equivalent node in the most recent version of the text string of the corresponding sub-history.

2. If a segment node in the most recent version of the text string of the corresponding sub-history has a null dual link, then its equivalent node in the most recent version of the text string of the history has been deleted.

6.5 Sticky Pointers

A sticky pointer is implemented as an object containing a field for an ID and a field for the position of the stuck character.

6.6 Operations on Characters

6.6.1 GetAbsoluteIndex

Given a character c in a text string, GetAbsoluteIndex(c) will return the absolute index of c in the text string. The algorithm computes the total weight of the path from the segment node pointed to by the character to the left-most node of the tree, through the root of the tree. The algorithm returns the sum of the negation of this weight and the original position of the character within the segment.

6.6.2 GetDual

Given a character c in a text string in a history, GetDual(c) will return the equivalent character in the corresponding sub-history, or vice versa. The algorithm returns a new character which points to the dual of the segment node of the original character and has the same position as the original character.

6.7 Operations on Text String Trees

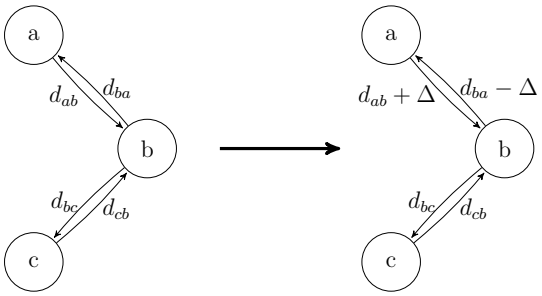
6.7.1 Locate

Given the absolute index i of a character in a text string ts, Locate(ts, i) will return the character with that absolute index, or return the information that such a character does not exist. The algorithm begins at the head of the tree and traverses to the root, while maintaining the offset to the current segment by computing the total weight of the path from the head to the current node. It then traverses down the tree. If i is less than the offset of the current node, it traverses left. If i is greater than the sum of the offset and the length of the text of the current node, it traverses right. If i falls between these two values,

then the character falls within the current segment. The current segment, along with the difference between i and the offset of the current segment is returned.

6.7.2 ModifyAncestorDisp

If a segment node s is inserted or deleted, or the string of the segment node changes in length by Δ, then ModifyAncestorDisp(s, Δ) is called to adjust the weights of the links representing the displacement between segments to reflect that change. The algorithm traverses up the tree from s to the root. If it encounters a node n such that s is in n's right subtree, but n is a left child of its parent p, then s falls between n and p. Thus, the weight of the links between n and p are increased or decreased in absolute value by Δ. The same is true if the algorithm encounters a node n such that s is in n's left subtree, but n is a right child of its parent p. An illustration of the modified links is presented below:



where $d_{xy} =$ Displacement(x, y).

6.7.3 InsertSegment

Given a new segment n to insert as the in-order predecessor/successor of an existing segment s of a text string, InsertSegment(s, n, lr) inserts n as a leaf of the tree such that it is the in-order predecessor/successor of s, where lr specifies whether n will precede or succeed s. The algorithm inserts n as the right-most/left-most segment of the left/right subtree of s and calls ModifyAncestorDisp$(n, n.\text{text.length})$ to adjust the displacements of the tree to reflect the insertion of n.

6.7.4 SplitSegment

Given a segment s to split and a position p to split s at, SplitSegment(s, p) will partition s into two segment nodes and return a reference to the second segment node, if a split has taken place. If $p = 0$, no split takes place and a null pointer is returned. If $p > 0$, then s's string t is split into two substrings $l = t[0 : p]$ and $r = t[p : t.\text{length}]$. s's string is replaced by l and ModifyAncestorDisp$(s, l.\text{length} - t.\text{length})$ is called to reflect the shortening. Additionally, a new segment n is created with the same ID as s and its string is set to r. InsertSegment$(s, n, \text{successor})$ is called to insert n as the in-order successor of s.

6.7.5 DeleteSegment

Given a segment s to delete from a text string, DeleteSegment(s) will remove s from the text string tree. The algorithm performs the segment node deletion the same way a binary seach tree deletes its nodes. It repeatedly swaps s with its in-order successor or in-order predecessor and adjusts the weights of the tree using ModifyAncestorDisp until s becomes a leaf. It then calls ModifyAncestorDisp($s, -s$.length) and detaches s from the tree.

6.8 Operations on Histories

6.8.1 CreateStickyPointer

Given a character c such that c is in the most recent version of the text string of a history, CreateStickyPointer(c) will return a sticky pointer that is stuck to that character. The algorithm begins by getting the dual of c. It will then call GetAbsoluteIndex(GetDual(c)). The algorithm will return the ID of the segment pointed to by c and the computed absolute index of the dual.

6.8.2 SplitDuals

Given a character c in the most recent version of the text string of a history, SplitDuals(c) splits the most recent version of the text string of a history at c and the most recent version of the text string of the corresponding sub-history at GetDual(c). The algorithm calls SplitSegment($c.seg, c.pos$) to split the text string of the history at c into s_L and s_R and also calls SplitSegment(GetDual(c).seg, GetDual(c).pos) to split the text string of the corresponding sub-history at GetDual(c) into s'_L and s'_R. s_L and s'_L are made duals of each other and s_R and s'_R are made duals of each other.

6.8.3 InsertString

Given a character c in the most recent version of the text string of a history and a string t to insert, InsertString(c, t) will insert t immediately preceding c in the most recent version of the text string. The algorithm begins by generating a unique ID u for t. Then it creates a new sub-history by creating a new text string and storing the text string in the map under the entry for u. The algorithm then creates two identical segments n, n' containing the u and t. u and u' are made duals of each other. The algorithm calls SplitDuals(c) to split the segment pointed to by c into s_L and s_R and split its dual. The algorithm then inserts n as the in-order successor of s_L by calling InsertSegment($s_L, n,$ successor) and inserts n' as the head of the text string of the new sub-history.

6.8.4 DeleteString

Given two characters c_s, c_e in the most recent version of the text string such that c_e follows c_s, DeleteString(c_s, c_e) removes the string beginning from c_s (inclusive) and ending with c_e (exclusive) from the most recent version of the text string. The algorithm begins by calling SplitDuals(c_s) to split the segment pointed to by c_s into a_L and a_R and SplitDuals(c_e)

to split the segment pointed to by c_e into b_L and b_R. The algorithm then iterates over the segments between a_R (inclusive) and b_L (inclusive) as defined by ordering of the text string, and for each segment, disconnects it from its dual and deletes it.

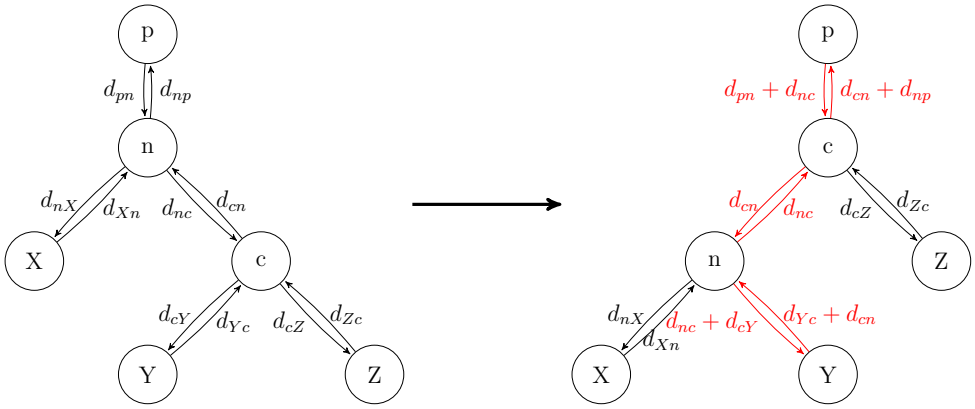
6.9 Operations on Sticky Pointers

6.9.1 Find

Given a sticky pointer sp within a history h, FIND(h, sp) will return the character pointed to by the sticky pointer within the most recent version of the text string, if it exists. The algorithm begins by locating the character with the absolute index equal to the position of sp, in the sub-history with the ID of sp. It then retrieves the dual of that character and returns it if it exists. If it doesn't exist, then the character was deleted.

6.10 Rebalancing

Because INSERTSEGMENT and SPLITSEGMENT insert new segments as leaves of the text string tree and DELETESEGMENT swaps nodes so that only a leaf is deleted from the text string tree, it follows that text string trees can be rebalanced using existing rotation-based, self-balancing binary search tree implementations while preserving the order of the text string. We extend segment nodes to include local balance information and after each insertion or deletion of a leaf, we update the balance information of the ancestors and rebalance according to the inherited balanced tree implementation. Because rotations do not alter the in-order ordering of the tree or insert or delete nodes, we do not need to alter any information other than the weights of the modified links following rotation. As a result, we must only modify a constant number of weights. Below, a left rotation on the nodes n and c is illustrated, with the modified links and their new weights highlighted in red:



where $d_{ab} =$ DISPLACEMENT(a, b). We can quickly see that the newly appointed link weights are appropriate due to the properties of the displacement function. For example, for the newly appointed link weight from p to c:

$$\begin{aligned} d_{pn} + d_{nc} &= \textsc{Displacement}(p, n) + \textsc{Displacement}(n, c) \\ &= \textsc{Offset}(n) - \textsc{Offset}(p) + \textsc{Offset}(c) - \textsc{Offset}(n) \\ &= \textsc{Offset}(c) - \textsc{Offset}(p) \\ &= \textsc{Displacement}(p, c) \end{aligned}$$

It follows that rotations only require a constant number of operations and have a constant runtime. Thus, we can efficiently bound the height of the text string tree so that it is *logarithmic* in the number of segment nodes that it contains. The number of segment nodes within the text string tree is linearly proportional to the number of Insert operations performed because each Insert operation can introduce at most 2 segments. Thus, it follows that the height of the tree is logarithmic in the number of Insert operations performed. Additionally, every operation on text string trees traverse the tree up and down at most once, and so it follows that every operation on text string trees has a worst-case runtime of $O(h)$, where h is the height of the tree. Now that the text string tree is balanced and the height is logarithmic in the number of Insert operations, it follows that every operation on self-balancing text string trees can be performed with a worst-case runtime of $O(\log n)$, where n is the number of Insert operations.

7 Conclusion

In this report, we have augmented self-balancing binary search trees to form a novel data structure to efficiently maintain text strings and sticky pointers. Our data structure leverages rotation-based self-balancing binary search tree implementations to bound the height of text string trees so that they are *logarithmic* in the number of Insert operations performed. Because Insert, Delete, and Find traverse the height of text string trees a constant number of times, it follows that Insert, Delete, CreateStickyPointer, and Find all have asymptotic worst-case runtimes that are logarithmic in the number of Insert operations performed. Additionally, we introduced two new operations to facilitate the translation between characters and their absolute indices: Locate, GetAbsoluteIndex. The algorithms for these two operations also traverse the height of text string trees a constant number of times so they also have logarithmic worst-case runtimes. This is a marked improvement over the self-balancing sticky pointer forests presented in Fall 2021, which could only support the implementation of these two operations with a linear worst-case runtime without additional structure.

8 Appendix

8.1 GetAbsoluteIndex

Algorithm 1 Compute the Absolute Index of a Character within the Text String

```
1: function GetAbsoluteIndex(c)
2:     curr = c.seg
3:     disp = c.pos
4:     while curr has a parent do
5:         disp −= curr.parent_disp
6:         curr = curr.parent
7:     end while
8:     while curr has a left child do
9:         disp −= curr.lchild_disp
10:        curr = curr.lchild
11:    end while
12:    return disp
13: end function
```

8.2 GetDual

Algorithm 2 Get the Dual of a Character within a History

```
1: function GetDual(c)
2:     return Character(c.seg.dual, c.pos)
3: end function
```

8.3 Locate

Algorithm 3 Locate a Character within the Text String

```
1: function LOCATE(ts, i)
2:     if ts is empty then
3:         return Character(NULL, −1)
4:     end if
5:     curr = ts.head
6:     disp = 0
7:     while curr has a parent do
8:         disp += curr.parent_disp
9:         curr = curr.parent
10:    end while
11:    while curr is in ts do
12:        if disp ≤ i ≤ disp + curr.text.length then
13:            return Character(curr, i − disp)
14:        else if i < disp then
15:            disp += curr.lchild_disp
16:            curr = curr.lchild
17:        else if disp + curr.text.length <= i then
18:            disp += curr.rchild_disp
19:            curr = curr.rchild
20:        end if
21:    end while
22:    return Character(NULL, −1)
23: end function
```

8.4 ModifyAncestorDisp

Algorithm 4 Modify the displacements of a text string tree to reflect a modified segment

```
 1: function ModifyAncestorDisp(s, Δ)
 2:     curr = s
 3:     if curr doesn't have a parent then
 4:         return
 5:     end if
 6:     parent = curr.parent
 7:     while parent has a parent do
 8:         gparent = parent.parent
 9:         if parent is a left child of gparent and curr is a right child of parent then
10:             gparent.lchild_disp −= Δ
11:             parent.parent_disp += Δ
12:         else if parent is a right child of gparent and curr is a left child of parent then
13:             gparent.rchild_disp += Δ
14:             parent.parent_disp −= Δ
15:         end if
16:         curr = parent
17:         parent = gparent
18:     end while
19: end function
```

8.5 InsertSegment

Algorithm 5 Inserts a new segment as the in-order predecessor/successor of a segment

```
1: function InsertSegment(s, n, lr)
2:     if lr is predecessor then
3:         if there is no left subtree of s then
4:             s.lchild = n
5:             s.lchild_disp = −n.text.length
6:             n.parent = s
7:             n.parent_disp = n.text.length
8:         else
9:             descendant = right-most segment of left subtree of s
10:            descendant.rchild = n
11:            descendant.rchild_disp = descendant.text.length
12:            n.parent = descendant
13:            n.parent_disp = −descendant.text.length
14:        end if
15:    else if lr is successor then
16:        if there is no right subtree of s then
17:            s.rchild = n
18:            s.rchild_disp = s.text.length
19:            n.parent = s
20:            n.parent_disp = −s.text.length
21:        else
22:            descendant = left-most segment of right subtree of s
23:            descendant.lchild = n
24:            descendant.lchild_disp = −n.text.length
25:            n.parent = descendant
26:            n.parent_disp = n.text.length
27:        end if
28:    end if
29:    ModifyAncestorDisp(n, n.text.length)
30: end function
```

8.6 SplitSegment

Algorithm 6 Splits a segment into two segments at a particular position

```
1: function SplitSegment(s, p)
2:     if p = 0 then
3:         return NULL
4:     end if
5:     t = s.text
6:     l = t[0 : p]
7:     r = t[p : t.length]
8:     s.text = l
9:     ModifyAncestorDisp(s, l.length − t.length)
10:    n = Segment(s.ID, r)
11:    InsertSegment(s, n, successor)
12: end function
```

8.7 DeleteSegment

Algorithm 7 Deletes a segment from the text string

```
1: function DeleteSegment(s)
2:     while s is internal do
3:         if s has a right child then
4:             descendant = left-most segment of right subtree of s
5:         else if s has a left child then
6:             descendant = right-most segment of left subtree of s
7:         end if
8:         Swap(s, descendant)
9:         ModifyAncestorDisp(descendant, descendant.text.length - s.text.length)
10:        ModifyAncestorDisp(s, s.text.length - descendant.text.length)
11:    end while
12:    parent = s.parent
13:    if s is a left child of parent then
14:        parent.lchild = NULL
15:        parent.lchild_disp = 0
16:    else if s is a right child of parent then
17:        parent.rchild = NULL
18:        parent.rchild_disp = 0
19:    end if
20:    s.parent = NULL
21:    s.parent_disp = 0
22: end function
```

8.8 CreateStickyPointer

Algorithm 8 Create a Sticky Pointer to a character

```
1: function CreateStickyPointer(c)
2:     ID = c.seg.ID
3:     dual = GetDual(c)
4:     abs_index = GetAbsoluteIndex(dual)
5:     return StickyPointer(ID, abs_index)
6: end function
```

8.9 Find

Algorithm 9 Compute the character pointed to by the sticky pointer

```
1: function Find(h, sp)
2:     dual = Locate(h.sub_text_string[sp.ID], sp.pos)
3:     c = GetDual(dual)
4:     return c
5: end function
```

References

[1] M.J. Fischer and R.E. Ladner. Data structures for efficient implementation of sticky pointers in text editors. Technical Report 79-06-08, Department of Computer Science, University of Washington, June 1979.