# Suffix Trays and Suffix Trists: Structures for Faster Text Indexing

Richard Cole[1] and Tsvi Kopelowitz[2] Moshe Lewenstein[2]

[1] New York University
cole@cs.nyu.edu
[2] Bar-Ilan University
{kopelot,moshe}@cs.biu.ac.il

**Abstract.** Suffix trees and suffix arrays are two of the most widely used data structures for text indexing. Each uses linear space and can be constructed in linear time [3, 5–7]. However, when it comes to answering queries, the prior does so in $O(m \log |\Sigma|)$ time, where $m$ is the query size, $|\Sigma|$ is the alphabet size, and the latter does so in $O(m + \log n)$, where $n$ is the text size. We propose a novel way of combining the two into, what we call, a *suffix tray*. The space and construction time remain linear and the query time improves to $O(m + \log |\Sigma|)$.
We also consider the online version of indexing, where the indexing structure continues to update the text online and queries are answered in tandem. Here we suggest a *suffix trist*, a cross between a suffix tree and a suffix list. It supports queries in $O(m + \log |\Sigma|)$. The space and text update time of a suffix trist are the same as for the suffix tree or the suffix list.

## 1 Introduction

*Indexing* is one of the most important paradigms in searching. The idea is to preprocess a text and construct a mechanism that will later provide answer to queries of the form "does a pattern $P$ occur in the text" in time proportional to the size of the *pattern* rather than the text. The suffix tree [3, 9–11] and suffix array [5–8] have proven to be invaluable data structures for indexing.

Both suffix trees and suffix arrays use $O(n)$ space, where $n$ is the text length. In fact for alphabets from a polynomially sized range, both can be constructed in linear time, see [3, 5–7].

The query time is slightly different in the two data structures. Namely, in suffix trees queries are answered in $O(m \log |\Sigma| + occ)$, where $m$ is the length of the query, $\Sigma$ is the alphabet, $|\Sigma|$ is the alphabet size and $occ$ is the number of occurrences of the query. In suffix arrays the time is $O(m + \log n + occ)$. For the rest of this paper we assume that we are only interested in one occurrence of the pattern in the text, and note that we can find all of the occurrences of the pattern with another additive $occ$ cost in the query time.

The differences in the running times follows from the different way queries are answered. In a suffix tree queries are answered by traversing the tree from

the root. At each node one needs to know how to continue the traversal and one needs to decide between $|\Sigma|$ options which are sorted, which explains the $O(\log|\Sigma|)$ factor. In suffix arrays one performs a binary search on all suffixes (hence the $\log n$ factor) and uses longest common prefix (LCP) queries to quickly decide whether the pattern needs to be compared to a specific suffix (see [8] for full details).

It is easy to construct a data structure with optimal $O(m)$ query time. This can be done by simply putting a $|\Sigma|$ length array at every node of the suffix tree. Hence, when traversing the suffix tree with the query we will spend constant time at each node. However, the size of this structure is $O(n|\Sigma|)$.

The question of interest here is whether one can construct an $O(n)$ space structure that will answer queries in time faster than the query time of suffix arrays and suffix trees. We indeed propose to do so with the *Suffix Tray*, a new data structure that extracts the advantages of suffix trees and suffix arrays by combining their structures. This yields an $O(m + \log|\Sigma|)$ query time.

We are also concerned with texts that allow online update of the text. In other words, given an indexing structure supporting indexing queries on $S$, we would also like to support extensions of the text to $Sa$, where $a \in \Sigma$. We assume that the text is given in reverse, i.e. from the last character towards the beginning. So, an indexing structure of our desire when representing $S$ will also support extensions to $aS$ where $a \in \Sigma$. We call the change of $S$ to $aS$ a *text extension*. The "reverse" assumption that we use is not strict, as most indexing structures can handle online texts that are reversed (e.g. instead of a suffix tree one can construct a prefix tree and answer the queries in reverse. Likewise, a prefix array can be constructed instead of a suffix array).

Online constructions of indexing structures have been suggested previously. McCreight's suffix tree algorithm [9] was the first online construction. It was a reverse construction (in the sense mentioned above). Ukkonen's algorithm [10] was the first online algorithm that was not reversed. In both these algorithms text extensions take $O(1)$ amortized time, but $O(n)$ worst-case time. In [1] an online suffix tree construction (under the reverse assumption) was proposed with $O(\log n)$ worst-case text extensions. In all these constructions a full suffix tree is constructed and hence queries are answered in $O(m \log|\Sigma|)$ time. An on-line variant of suffix arrays was also proposed in [1] with $O(\log n)$ worst-case for text extensions and $O(m + \log n)$ for answering queries. Similar results can be obtained by using the results in [4].

The problem we deal with in the second part of the paper is how to build an indexing structure that supports both text extensions and supports fast(er) indexing. We will show that if there exists an online construction for a linear-space suffix tree such that the cost of adding a character is $O(f(n, |\Sigma|))$ ($n$ is the size of the current text), then we can construct an online linear-space data-structure for indexing that supports indexing queries in time $O(m + \log|\Sigma|)$,

where the cost of adding a character is $O(f(n, |\Sigma|) + \log |\Sigma|)$. We will call this data structure the *Suffix Trist*[3].

## 2 Suffix Trees, Suffix Arrays and Suffix Intervals

Consider a text $S$ of length $n$ and let $S^1, \cdots, S^n$ be the suffixes of $S$. Two classical data structures for indexing are the suffix tree and the suffix array. We assume that the reader is familiar with the suffix tree. Let $S^{i_1}, ..., S^{i_n}$ be the lexicographic ordering of the suffixes. The suffix array of $S$ is defined to be $SA(S) =< i_1, ..., i_n >$, i.e. the indices of the lexicographic ordering of the suffixes. We will sometimes refer to location $j$ of the suffix array as the location of $S^{i_j}$ (instead of the location of $i_j$).

We let $ST(S)$ and $SA(S)$ denote the suffix tree and suffix array of $S$, respectively. As with all suffix array constructions to date, we make the assumption that every node in a suffix tree maintains its children in lexicographic order. Therefore, the leaves ordered by an inorder traversal correspond to the suffixes in lexicographic order, which is also the order maintained in the suffix array. Hence, one can view the suffix tree as a tree over the suffix array.

We will now sharpen this connection between suffix arrays and suffix trees.

For strings $R$ and $R'$ we say that $R <_L R'$ if $R$ is lexicographically smaller than $R'$. $leaf(S^i)$ denotes the leaf corresponding to $S^i$ in $ST(S)$, the suffix tree of $S$. We define $L(v) = SA^{-1}(i)$ if $leaf(S^i)$ is the leftmost leaf of the subtree of $v$, i.e. $L(v)$ is the location of $S^i$ in the suffix array. Note that since we assume that the children of a node in a suffix tree are maintained in lexicographic order, it follows that for all $S^j$ such that $leaf(S^j)$ is a descendant of $v$, $S^i \leq_L S^j$. Likewise, we define $R(v) = SA^{-1}(i)$ if $leaf(S^i)$ is the rightmost leaf of the subtree of $v$. Therefore, for all $S^j$ such that $leaf(S^j)$ is a descendant of $v$, $S^i \geq_L S^j$. Hence, the interval $[L(v), R(v)]$ is an interval of the suffix array which contains exactly all the suffixes $S^j$ for which $leaf(S^j)$ is a descendant of $v$.

Moreover, under the assumption that the children of a node in a suffix tree are maintained in lexicographic ordering we can state the following.

**Lemma 1.** *Let $S$ be a string and $ST(S)$ its suffix tree. Let $v$ be a node in $ST(S)$ and let $v_1, ..., v_r$ be its children. Let $1 \leq i \leq j \leq r$, and let $[L(v_i), R(v_j)]$ be an interval of the suffix array. Then $k \in [L(v_i), R(v_j)]$ if and only if $leaf(S^k)$ is in one of the subtrees rooted at $v_i, ..., v_j$.*

This leads us to the following concept.

**Definition 1.** *Let $S$ be a string and $\{S^{i_1}, ..., S^{i_n}\}$ be the lexicographic ordering of its suffixes. The interval $[j, k] = \{i_j, ..., i_k\}$, for $j \leq k$, is called a* suffix interval.

[3] The name Suffix Trist is derived from the combination of suffix trees and suffix lists, the dynamic version of the suffix array. Of course, one may argue that it may be preferable to work with prefix arrays and prefix lists. Then one would receive Prefix Prays and Prefix Priests.

Obviously, suffix intervals are intervals of the suffix array. We note that, as mentioned above, for a node $v$ in a suffix tree, $[L(v), R(v)]$ is a suffix interval and we call the interval *$v$'s suffix interval*. Also, by Lemma 1 for $v$'s children $v_1, ..., v_r$ and for any $1 \leq i \leq j \leq r$, $[L(v_i), R(v_j)]$ is a *suffix interval* and we call this interval the *$(i, j)$-suffix interval*.

## 3 Faster Indexing on Static Text

### 3.1 Suffix Trays

We now elaborate on the suffix tray. The suffix tray will use the concept of suffix intervals from the previous section which, as we have seen, is common to both suffix arrays and suffix trees.

For suffix trays we will create special nodes, which correspond to suffix intervals. We call these nodes *suffix interval nodes*. Part of the suffix tray will be a suffix array. Each suffix interval node can be viewed as a node that maintains the endpoints of the interval within the complete suffix array.

Second, we use the idea of the space-inefficient $O(n|\Sigma|)$ suffix tree solution mentioned in the introduction. We maintain $|\Sigma|$ length arrays at a selected subset of nodes, a subset that contains no more that $\frac{n}{|\Sigma|}$ nodes, which maintains the $O(n)$ space bound. To choose this selected subset of nodes we define the following.

**Definition 2.** *Let $S$ be a string over alphabet $\Sigma$. A node $u$ in $ST(S)$ is called a $\sigma$-node if the number of leaves in the subtree of $ST(S)$ rooted at $u$ is at least $|\Sigma|$. A $\sigma$-node $u$ is called a branching-$\sigma$-node, if at least two of $u$'s children in $ST(S)$ are $\sigma$-nodes and is called a $\sigma$-leaf if all its children in $ST(S)$ are not $\sigma$-nodes.*

Note that if a node $u$ is a $\sigma$-node, then all of its ancestors are $\sigma$-nodes. This also implies that in a $\sigma$-leaf's subtree there are no $\sigma$-nodes. The following property of branching-$\sigma$-nodes is crucial to our result.

**Lemma 2.** *Let $S$ be a string of size $n$ over an alphabet $\Sigma$ and let $ST(S)$ be its suffix tree. The number of branching-$\sigma$-nodes in $ST(S)$ is $O(\frac{n}{|\Sigma|})$.*

*Proof.* The number of $\sigma$-leaves is at most $\frac{n}{|\Sigma|}$ because (1) they each have at least $|\Sigma|$ leaves in their subtree and (2) their subtrees are disjoint. Let $T$ be the tree induced by the $\sigma$-nodes and contracted onto the branching-$\sigma$-nodes and $\sigma$-leaves only. Then $T$ is a tree with $\frac{n}{|\Sigma|}$ leaves and with every internal node having at least 2 children. Hence, the lemma follows. □

This means that we can afford to maintain arrays at every branching-$\sigma$-node which will be very helpful in answering queries as we shall see in subsection 3.3.

### 3.2 Suffix Tray Construction

A *suffix tray* is constructed from a suffix tree as follows. The suffix tray contains all the $\sigma$-nodes of the suffix tree. We also add some suffix interval nodes to the suffix tray as children of $\sigma$-nodes. Here is how each $\sigma$-node is converted from the suffix tree to the suffix tray.

- $\sigma$-leaf $u$: $u$ becomes a suffix interval node with suffix interval $[L(u), R(u)]$.
- non-leaf $\sigma$-node $u$: Let $u_1, ..., u_r$ be $u$'s children in the suffix tree and $u_{l_1}, ..., u_{l_x}$ be the subset of children that are $\sigma$-nodes. Then $u$ will be in the suffix tray with interleaving suffix interval nodes and $\sigma$-nodes, i.e. $(1, l_1 - 1)$-suffix interval node, $u_{l_1}$, $(l_1 + 1, l_2 - 1)$-suffix interval node, $u_{l_2}$, ..., $u_{l_x}$, $(l_x + 1, r)$-suffix interval node.

At each branching-$\sigma$-node $u$ in the suffix tray we maintain an array of size $|\Sigma|$, denoted by $A_u$, that contains the following data. For every child $v$ of $u$ that is a $\sigma$-node, location $\tau$ in $A_u$ where $\tau$ is the first character on the edge $(u, v)$, points to $v$. The rest of the locations in $A_u$ point to the appropriate suffix-interval node, or to a NIL pointer if no such suffix interval exists.

At each $\sigma$-node $u$ which is not a branching-$\sigma$-node and not a $\sigma$-leaf, i.e. it has exactly one child $v$ which is a $\sigma$-node, we store the first character $\tau$ on the edge $(u, v)$, which we call the *separating character*.

We now claim that the suffix tray is of linear size.

**Lemma 3.** *Let $S$ be a string of size $n$. Then the size of the suffix tray for $S$ is $O(n)$.*

*Proof.* The suffix array is clearly of size $O(n)$ and the number of suffix interval nodes is bounded by the number of nodes in $ST(S)$. Also, for each non branching-$\sigma$-node the auxiliary information is of constant size.

The auxiliary information held in each branching-$\sigma$-node is of size $O(|\Sigma|)$. By Lemma 2 there are $\frac{n}{|\Sigma|}$ branching-$\sigma$-nodes. Hence, this all is of size $O(n)$. □

Obviously, given a suffix tree and suffix array, a suffix tray can be constructed in linear time (using depth-first searches, and standard techniques). Since both suffix arrays and suffix trees can be constructed in linear time for alphabets from a polynomially sized range [3, 5–7], so can suffix trays.

### 3.3 Navigating on Index Queries

We now turn to the important feature of suffix trays, answering index queries.

Upon receiving a query $P = p_1...p_m$ we begin traversing the suffix tray from the root. Assume that we have traversed the suffix tray with $p_1...p_{i-1}$ and need to continue with $p_i...p_m$. At each branching-$\sigma$-node $u$ we access location $p_i$ of the array $A_u$ in order to know which suffix tray node to navigate to. Obviously, since this is an array lookup this takes us constant time. For other $\sigma$-nodes that are not $\sigma$-leaves and not branching-$\sigma$-nodes we compare $p_i$ with the separator character $\tau$. Recall that these nodes have only one child $v$ that is a $\sigma$-node.

Hence, in the suffix tray the children of $u$ are (1) a suffix interval node to the left of $v$, say $u$'s left interval, (2) $v$, and (3) a suffix interval node to the right of $v$, say $u$'s right interval. If $p_i < \tau$ we navigate to $u$'s left interval. If $p_i > \tau$ we navigate to $u$'s right interval. If $p_i = \tau$ we navigate to the only child of $u$ that is a $\sigma$-node. If $u$ is a $\sigma$-leaf then we are at $u$'s suffix interval.

To search within a suffix interval $[j, k]$ we apply the standard suffix array search beginning with boundaries $[j, k]$. The time to search in this structure is $O(m + \log I)$, where $I$ is the interval size. Hence, the following is important.

**Lemma 4.** *Every suffix interval in a suffix tray is of size $O(|\Sigma|^2)$.*

*Proof.* Consider an $(i, j)$-suffix interval, i.e. the interval $[L(v_i), R(v_j)]$ which stems from a node $v$ with children $v_1, ..., v_r$. Note that by Lemma 1 the $(i, j)$-suffix interval contains the suffixes which are represented by leaves in the subtrees of $v_i, ..., v_j$. However, $v_i, ..., v_j$ are not $\sigma$-nodes (by suffix tray construction). Hence, each subtree of those nodes contains at most $|\Sigma| - 1$ leaves. Since $j - i + 1 \leq |\Sigma|$ the overall size of the $(i, j)$-suffix interval is $O(|\Sigma|^2)$.

A suffix interval $[L(v), R(v)]$ is maintained only for $\sigma$-leaves. As none of the children of $v$ are $\sigma$-nodes this is a special case of the $(i, j)$-suffix interval. □

By the discussion above and Lemma 4 the running time for answering an indexing query is $O(m+\log |\Sigma|)$. Summarizing the discussion of the whole section we can claim the following.

**Theorem 1.** *Let $S$ be a length $n$ string over an alphabet $\Sigma$. The suffix tray of $S$ is (1) of size $O(n)$, (2) can be constructed in $O(n + construct_{ST}(n, \Sigma) + construct_{SA}(n, \Sigma))$ time (where $construct_{ST}(n, \Sigma)$ and $construct_{SA}(n, \Sigma)$ are the times to construct the suffix tree and suffix array) and (3) supports indexing queries (of size $m$) in time $O(m + \log |\Sigma|)$.*

## 4 The Online Scenario

In this section we deal with the problem of how to build an indexing structure that supports both text extensions and supports fast(er) indexing. We show that if there exists an online construction for a linear-space suffix tree such that the cost of adding a character is $O(f(n, |\Sigma|))$ ($n$ is the size of the current text), then we can construct an online linear-space data-structure for indexing that supports indexing queries in time $O(m + \log |\Sigma|)$, where the cost of adding a character is $O(f(n, |\Sigma|) + \log |\Sigma|)$. During the construction, we will treat the online linear-space suffix-tree construction as a suffix-tree oracle that provides us with the appropriate updates to the suffix tree. Specifically, the best known current construction supports text extensions in $O(\log n)$, see introduction. As already mentioned, the data structure we present is called the Suffix Trist.

The suffix trist imitates the suffix trays. We still use $\sigma$-nodes and branching-$\sigma$-nodes in the suffix tree, and the method for answering indexing queries is similar. However, new issues arise in the online model:

– Suffix arrays are static data structures and, hence, do not support insertion of new suffixes.
– The status of nodes changes as time progresses (non-$\sigma$-nodes become $\sigma$-nodes, and $\sigma$-nodes become branching-$\sigma$-nodes).

### 4.1 Balanced Indexing Structures and Suffix Trists

In order to solve the first of the two problems we turn to a dynamic variant of suffix arrays which can be viewed as a structure above a suffix list.

The Balanced-Indexing-Structure, BIS for short, was introduced in [1]. BIS is a binary search tree over the suffixes where the ordering is lexicographic. In [1] it was shown how the BIS can be updated in $O(\log n)$ time for every *text extension*, where $n$ is the current text size. Moreover, a BIS supports indexing queries in time $O(m + \log n)$, where $m$ is the query size.

This leads to the following idea for creating a *Suffix Trist* instead of a suffix tray. Take a separate BIS for every suffix interval. Since the suffix intervals are of size $O(|\Sigma|^2)$ the search time in those small BISs will be $O(m + \log |\Sigma|)$.

However, things are not as simple as they seem. Insertion of suffix $aS$ into a BIS for a string $S$ assume that we have all the suffixes of $S$ in the BIS, or more specifically assumes that the suffix $S$ itself is in the BIS. This may not be the case if we limit the BIS to a suffix interval, which contains only part of the suffixes. Nevertheless, in our case there is a way to circumvent this problem. We describe the solution in the next subsection.

Also, we still need to deal with the second problem of nodes changing status. Our solution is a direct deamortized solution and is presented in Section 5.

### 4.2 Inserting New Nodes into BISs

When we perform a text extension from $S$ to $aS$, the suffix tree is updated to represent the new text $aS$ (by our suffix-tree oracle). Specifically, a new leaf, corresponding to the new suffix, is added to the suffix tree, and perhaps one internal node is also added. If such an internal node is inserted, then that node is the parent of the new leaf and this happens in the event that the new suffix diverges from an edge (in the suffix tree of $S$) at a location where no node previously existed. In this case an edge needs to be broken into two and the internal node is added at that point.

Since we assume that the online suffix tree is given to us, what we need to show is how to update the suffix trist using the suffix tree (updated by the oracle). The problem is (1) to find the correct BIS in which to insert the new node and (2) to actually insert it into this BIS. Of course, this may change the status of internal nodes, which we handle in Section 5. We focus on solving (1) and mention that (2) can be solved by BIS tricks in $O(\log |\Sigma|)$ time, which we defer to a full version, but mention that they are similar to what appears in [1] and hence we find it somewhat less interesting here.

The following lemma which we state without proof will be useful and follows from the definition of suffix trists.

**Lemma 5.** *For a node $u$ in a suffix tree, if $u$ is not a $\sigma$-node, then all of the leaves in $u$'s subtree are in the same BIS.*

It will also be handy to maintain a pointer $leaf(u)$ to some leaf in $u$'s subtree for every node $u$ in the suffix tree. This variant can easily be maintained under text extensions using standard techniques.

In order to find the correct BIS in which the new node is to be inserted we consider two cases. First, consider the case where the new leaf $u$ in the suffix tree is inserted as a child of an already existing internal node $v$. If $v$ is not a $\sigma$-node, then from Lemma 5 we know that $leaf(v)$ and $u$ need to be in the same BIS. By traversing up from $leaf(v)$ to the root of the BIS (in $O(\log |\Sigma|)$ time) we can find the root of the BIS which needs to include the new node $u$. If $v$ is a $\sigma$-node, then we can locate the root of the appropriate BIS in constant time: if $v$ is a branching-$\sigma$-node, then we can find the BIS in constant time from the array in $v$ (we will guarantee that this holds in the online setting as well). If $v$ is a $\sigma$-leaf then there is only one possible BIS. Otherwise, ($v$ is a non-leaf $\sigma$-node but not a branching-$\sigma$-node) we can the find correct BIS of the two possible BISs by examining the separating character maintained in $v$.

Next, consider the case where the new leaf $u$ in the suffix tree is inserted as a child of a new internal node $v$. Let $w$ be $v$'s parent, and let $w'$ be $v$'s other child (not $u$). We first ignore $u$ completely by treating the new tree as the suffix tree of $S$ where the edge $(w, w')$ is broken into two, creating the new node $v$. After we show how to update the trist to include $v$, we can add $u$ as we did in the case that $v$ was already an internal node. In order to determine the status of $v$, note that $v$ cannot be a branching-$\sigma$-node. Moreover, note that the number of leaves in $v$'s subtree is the same as the number of leaves in the subtree rooted at $w'$ (as we are currently ignoring $u$). So, if $w'$ is not a $\sigma$-node, $v$ is not a $\sigma$-node, and otherwise, $v$ is a $\sigma$-node with a separating character that is the first character of the label of edge $(v, w')$. Note that the entire process takes $O(\log |\Sigma|)$ time, as required.

## 5 When a Node Changes Status

Before explaining how to update a node that becomes a $\sigma$-node, we must explain how to detect that this event has taken place. This is explained next.

### 5.1 Detecting a new $\sigma$-node

Let $u$ be a new $\sigma$-node and let $v$ be its parent. Just before $u$ becomes a $\sigma$-node, (1) $v$ must have already been a $\sigma$-node and (2) $u \in \{v_i, ..., v_j\}$ and is associated with an $(i, j)$-suffix interval represented by a suffix interval node $w$ that is a child of $v$ in the suffix trist. Hence, we will be able to detect when a new $\sigma$-node is created if we maintain counters for each of the (suffix tree) nodes $v_i, ..., v_j$ to count the number of leaves in their subtrees (in the suffix tree). These counters are maintained in a binary search tree, which we associate with the BIS, and each counter $v_k$ is indexed by the the first character on the edge $(v, v_k)$.

The update of the counters can be done as follows. When a new leaf is added into a given BIS of a suffix trist at suffix interval node $w$ of the BIS, where $u$ is the parent of $w$, we need to increase the counter of $v_k$ in the BIS, where $v_k$ is the one node (of the nodes of $v_i, ..., v_j$ of the suffix interval of the BIS) which is an ancestor of the new leaf. The counter of $v_k$ can be found in $O(\log |\Sigma|)$ (as there are at most $|\Sigma|$ nodes in the binary search tree for the counters). The appropriate counter is found by searching with the character that appears on the new suffix at the location immediately after $|label(v)|$; the character can be found in constant time by accessing it from a pointer from node $v$ into the text.

Note that when a new internal node was inserted into the suffix tree as described in the previous section, it is possible that the newly inserted internal node is now one of the nodes $v_i, ..., v_j$ for an $(i, j)$-suffix interval. In such a case, when the new node is inserted, it copies the number of leaves in its subtree from its only child (as we explained in the previous section we ignore the newly inserted leaf), as that child was previously maintaining the number of leaves in its subtree. Furthermore, from now on we will only update the size of the subtree of the newly inserted node, and not the size of its child subtree.

### 5.2 Updating the New $\sigma$-Node

Let $u$ be the new $\sigma$-node (which is, of course, a $\sigma$-leaf) and let $v$ be its parent. As discussed in the previous subsection $u \in \{v_i, ..., v_j\}$ where $v_i, ..., v_j$ are the children of $v$ (in the suffix tree) and, as discussed in the previous subsection, just before becoming a $\sigma$-node there was a suffix interval node $w$ that was an $(i, j)$-suffix interval with a BIS representing it.

Updating the new $\sigma$-node will require two things. First, we need to split the BIS into 3 parts; two new BISs and the new $\sigma$-leaf that separates between them. Second, for the new $\sigma$-leaf we will need to add the separating character (easy) and to create a new set of counters for the children of $u$ (more complicated).

The first goal will be to split the BIS that has just been updated into three - the nodes corresponding to suffixes in $u$'s subtree, the nodes corresponding to suffixes that are lexicographically smaller than the suffixes in $u$'s subtree, and the nodes corresponding to suffixes that are lexicographically larger than the suffixes in $u$'s subtree.

As is well-known, for a given a value $x$, splitting a BST, balanced suffix tree, into two BSTs at value $x$ can be implemented in $O(\text{h})$ time, where $h$ is the height of the BST. The same is true for BISs (although there is a bit more technicalities to handle the auxiliary information). Since the height of BISs is $O(\log |\Sigma|)$ we can split a BIS into two BISs in $O(\log |\Sigma|)$ time and by finding the suffixes (nodes in the BIS) that correspond to the rightmost and leftmost leaves of the subtree of $u$, we can split the BIS into the three desired parts in $O(\log |\Sigma|)$ time. Fortunately, we can find the two nodes in the BIS in $O(\log |\Sigma|)$ time using $leaf(u)$, the length of $label(u)$, and the auxiliary data in the BIS. We leave the full details for the full version.

We now turn to creating the new counters. Denote the children of $u$ by $u_1, ..., u_k$. We first note that the number of suffixes in a subtree of $u_i$ can be

counted in $O(\log|\Sigma|)$ time by a traversal in the BIS using classical tricks of binary search trees. We now show that we have enough time to update all the counters of $u_1, ..., u_k$ before one of them becomes a $\sigma$-node, while still maintaining the $O(\log|\Sigma|)$ bound per update.

Specifically, we will update the counters during the first $k$ insertions into the BIS of $u$ (following the event of $u$ becoming a $\sigma$-node). At each insertion we update one of the counters. What is critical for this to be done in time for the counters to be useful, i.e. in time to detect a new $\sigma$-node occurring in the subtree of $u$. The following lemma is precisely what is needed.

**Lemma 6.** *Let $u$ be a node in the suffix tree, and let $u_1, ..., u_k$ be $u$'s children (in the suffix tree). Say $u$ has just become a $\sigma$-node. Then at this time, the number of leaves in each of the subtrees of $u$'s children is at most $|\Sigma| - k + 1$.*

*Proof.* Assume by contradiction that this is not the case. Specifically, assume that child $v_i$ has at least $|\Sigma| - k + 2$ leaves in its subtree at this time. Clearly, the number of leaves in each of the subtrees is at least one. So summing up the number of leaves in all of the subtrees of $u_1, ..., u_k$ is at least $|\Sigma| - k + 2 + k - 1 = |\Sigma| + 1$, contradicting the fact that $u$ just became a $\sigma$-node (it should have already been a $\sigma$-node). □

### 5.3 When a $\sigma$-Leaf Loses its Status

The situation where a $\sigma$-leaf becomes a non-leaf $\sigma$-node is actually a case that we have covered in the previous subsection. Let $v$ be a $\sigma$-leaf that is about to change its status to a non-leaf $\sigma$-node. This happens because one of its children $v_k$ is about to become a $\sigma$-leaf. Note that just before the change $v$ is a suffix interval node. As in the previous subsection we will need to split the BIS representing the suffix interval into three parts, and the details are exactly the same as in the previous subsection. As before this is done in $O(\log|\Sigma|)$ time.

### 5.4 When a $\sigma$-Node Becomes a Branching-$\sigma$-Node

Let $v$ be a $\sigma$-node that is changing its status to a branching-$\sigma$-node. Just before it changes its status it had exactly one child $v_j$ which was a $\sigma$-node. The change in status must occur because another child (in the suffix tree), say $v_i$, has become a $\sigma$-leaf (and now that $v$ has two children that are $\sigma$-nodes it has become a branching-$\sigma$-node).

Just before becoming a branching-$\sigma$-node $v$ contained a separating character $\tau$, the first character on the edge $(v, v_j)$, and two suffix interval nodes $w$ and $x$, corresponding to the left interval of $v$ and the right interval of $v$, respectively. Now that $v_i$ became a $\sigma$-leaf $w$ was split into three parts (as described in subsection 5.2). Assume, without loss of generality, that $v_i$ precedes $v_j$ in the list of $v$'s children. So, in the suffix trist the children of $v$ are (1) a suffix interval node $w_L$, (2) a $\sigma$-leaf $v_i$, (3) a suffix interval node $w_L$, (4) a $\sigma$-node $v_j$, and (5) a suffix interval node $x$. We will denote with $B_1, B_2$ and $B_3$ the BISs that represent the suffix interval nodes $w_L, w_R$ and $x$.

The main problem here is that constructing the array $A_v$ takes too much time, so we must use a different approach and spread the construction over some time. We first give a pseudo-amortized solution and then mention how to (really) deamortize it. The following lemma allows us this time.

**Lemma 7.** *From the time that $w$ becomes a branching-$\sigma$-node, at least $|\Sigma|$ insertions are required into $B_1, B_2$ or $B_3$ before any node in the subtree of $v$ (in the suffix tree) that is not in the subtrees of $v_i$ or $v_j$ becomes a branching-$\sigma$-node.*

*Proof.* Clearly, at this time, any node in the subtree of $v$ (in the suffix tree) that is not in the subtrees of $v_i$ or $v_j$ has fewer than $|\Sigma|$ leaves in its subtree. On the other hand, note that any branching-$\sigma$-node must have at least $2|\Sigma|$ leaves in its subtree, as it has at least two children that are $\sigma$-nodes, each contributing at least $|\Sigma|$ leaves. Thus, in order for a node in the subtree of $v$ (in the suffix tree) that is not in the subtrees of $v_i$ or $v_j$ to become a branching-$\sigma$-node, at least $|\Sigma|$ leaves need to be added into its subtree, as required. □

This yields the pseudo-amortized result, as we can always amortize the $A_v$ construction over its insertions into $B_1, B_2$ and $B_3$. The crucial observation that follows from Lemma 7 that on any given search path we charge for at most one branching-$\sigma$-nodes construction, even if we go through several branching-$\sigma$-nodes.

The reason that we call this a pseudo-amortized result is because the $A_v$ construction charges on future insertions that may not occur. So, we take a lazy approach to solve this problem and this also yields the deamortized result.

We start by using the folklore trick of initializing the array $A_v$ in constant time. Then every time an insertion takes place into one of $B_1, B_2$ or $B_3$ we add one more element to the array $A_v$. Lemma 7 assures us that $A_v$ will be constructed before we begin to handle a branching-$\sigma$-node that is a descendant of $v$ but not of $v_i$ or $v_j$.

This scheme allows us to construct $A_v$ while maintaining the $O(\log |\Sigma|)$ time bound. However, it is still unclear how an indexing query should be answered when encountering $v$ on the traversal of the suffix tree. This is because on the one hand $A_v$ might not be fully constructed, and on the other hand, as time progresses, $v$ might have a non-constant number of children that are $\sigma$-nodes. We overcome this issue as follows. We continue to maintain the initial separating character $\tau$ of $v$ and another separator $\tau'$, the first character on the edge $(v, v_i)$, until $A_v$ is fully constructed. If we are at $v$ during a traversal for a query and the continuation of the traversal is to either $v_i$ or $v_j$ then we can discover this in constant time from $tau'$ or $\tau$. For the rest of the children of $v$ that are $\sigma$-nodes, maintain them all in a BST, so that when answering an indexing query, we can discover the appropriate place to continue in $O(log|\Sigma|)$ time (as there are only $|\Sigma|$ children). This does not affect the time it takes to answer an indexing query as we are guaranteed by Lemma 7 that if we need to use the BST of the children that are $\sigma$-nodes, then we will not encounter any more branching-$\sigma$-nodes afterwards. Thus, we at most add another $O(log|\Sigma|)$ to the query time.

There is one more loose end that we need to deal with. When other children of $v$ (other, as opposed to $v_i$ and $v_j$) become $\sigma$-nodes during the construction of $A_v$,

this can affect many of the locations of $A_v$. Specifically, updating accordingly could take too much time (or might require too many insertions in order to complete it). In order to solve this problem we define $A_v$ in a slightly different way as opposed to the static case in order to support this. Each entry in $A_v$ will point us to the edge whose label begins with the character of that entry, if such an edge exists. If no such edge exists, we simply put a NIL. This still allows us to spend constant time per branching-$\sigma$-node when answering an indexing query. However, when we go on to the edge pointed by the appropriate location (during the process of answering a query), we look at the node $v'$ on the other side of the edge. If $v'$ is a $\sigma$-node, we continue to traverse from there. If $v'$ is not a $\sigma$-node, then we can find the appropriate BIS by following $leaf(u)$, and traversing up to the root of the BIS in $O(\log|\Sigma|)$ time. Now, when a new node becomes a $\sigma$-node, and its parent is already a branching-$\sigma$-node, no more changes are required.

**Theorem 2.** *Let $S$ be a string over an alphabet $\Sigma$. The suffix trist of $S$ is (1) of size $O(n)$, (2) supports text extensions in time $O(\log|\Sigma|) + extension_{ST}(n,\Sigma))$ time (where $extension_{ST}(n,\Sigma)$is the time for a text extension in the suffix tree) and (3) supports indexing queries (of size $m$) in time $O(m + \log|\Sigma|)$.*

## References


1. A. Amir, T. Kopelowitz, M. Lewenstein, and N. Lewenstein. Towards Real-Time Suffix Tree Construction. *Proc. of Symp. on String Processing and Information Retrieval (SPIRE)*, 67-78, 2005.
2. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.
3. M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.
4. R. Grossi and G. F. Italiano. Efficient techniques for maintaining multidimensional keys in linked data structures. In *Proc. 26th Intl. Col. on Automata, Languages and Programming (ICALP)*, LNCS 1644, pages 372–381, 1999.
5. Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03)*, LNCS 2719, pages 943–955, 2003.
6. D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. *Proc. of 14th Symposium on Combinatorial Pattern Matching*, 186-199, LNCS 2676, 2003.
7. P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Proc. of 14th Symposium on Combinatorial Pattern Matching*, 200-210, LNCS 2676, 2003.
8. U. Manber and E.W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. on Computing*, 22(5):935-948, 1993.
9. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.
10. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
11. P. Weiner. Linear pattern matching algorithm. *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.