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

Richard Cole† Tsvi Kopelowitz‡ Moshe Lewenstein§

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 [5, 8, 9, 10]. 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. If one wants to output all appearances of the query an additive cost of $O(occ)$ is sufficient, where occ is the size of the output.

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 $O(m + \log |\Sigma|)$. Here also an additive $O(occ)$ is sufficient if one desires to output all appearances of the query.

We also consider the online version of indexing, where the indexing structure continues to update the text online and queries are answered in tandem. In this variant we create a cross between a suffix tree and a suffix list (a dynamic variant of suffix array) to be called a *suffix trist*; it supports queries in $O(m + \log |\Sigma|)$. The suffix trist also uses space and text update time that are no more than 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 [5, 12, 13, 14] and suffix array [8, 9, 10, 11] 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 [5, 8, 9, 10].

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, Σ is the alphabet, $|\Sigma|$ is the alphabet size and occ is the number of occurrences of the query. In suffix arrays the time is

*Results from this paper have appeared as an extended abstract in ICALP 2006.

†Department of Computer Science, Courant Institute, NYU.

‡Dept. of Computer Science, Bar-Ilan U., 52900 Ramat-Gan, Israel

§This research was supported by the Israel Science Foundation (grant no. 1848/04)

$O(m + \log n + occ)$. In [2] it was shown that the search time of $O(m + \log n + occ)$ is possible also on suffix trees. 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 ways 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 at most $|\Sigma|$ 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 [11] for full details).

It is easy to construct a data structure that will yield optimal $O(m)$ time to answer queries. 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 $O(n|\Sigma|)$. Also, it should be noted that this method assumes Σ is comprised of $\{1, 2, ..., |\Sigma|\}$ as we need to be able to random access locations in the array based on the current character, and so we need our alphabet to be proper indices. While this can be overcome using renaming schemes, it will provide an extra $O(m \log |\Sigma|)$ time for the query process, as one would need to rename each of the characters in the pattern.

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. However, our solution uses some $|\Sigma|$ length arrays to allow for quick navigation, and as such we are also confined to using alphabets of the form $\Sigma = \{1, 2, ..., |\Sigma|\}$

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 [12] was the first online construction. It was a reverse construction (in the sense mentioned above). Ukkonen's algorithm [13] 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 [7].

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

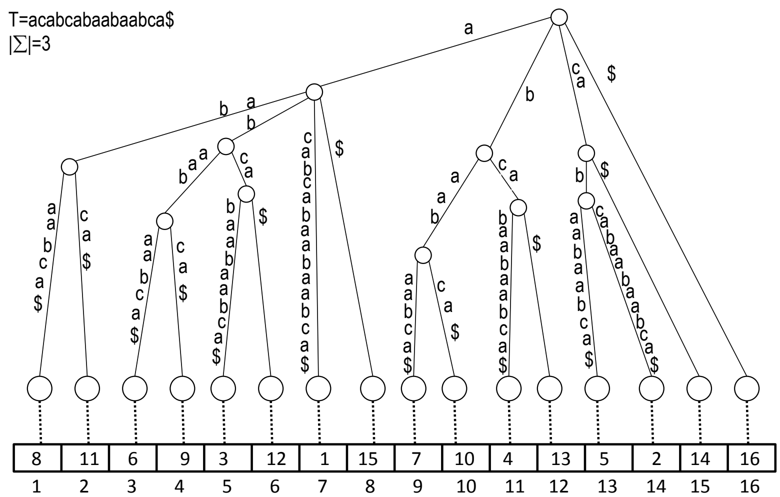

Figure 1: A suffix tree with a suffix array at leaves.

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*.

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. It is assumed 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. Location j of the suffix array is sometimes referred to as the location of S^{i_j} (instead of the location of i_j).

Let $ST(S)$ and $SA(S)$ denote the suffix tree and suffix array of S, respectively. As with all suffix array constructions to date, assume 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. See Figure 1.

This connection between suffix arrays and suffix trees will now be sharpened.

For strings R and R' 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. 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 the children of a node in a suffix tree are assumed to be 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, 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 one can state the following.

Observation 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 in lexicographic order. 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 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.

Obviously, suffix intervals are intervals of the suffix array. Note that, as mentioned above, for a node v in a suffix tree, $[L(v), R(v)]$ is a suffix interval and the interval is called *v's suffix interval*. Also, by Observation 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 this interval is called the *(i, j)-suffix interval*.

3 Suffix Trays

The suffix tray is elaborated on next. The suffix tray will use the concept of suffix intervals from the previous section which, as has been seen, is common to both suffix arrays and suffix trees.

For suffix trays special nodes are created, which correspond to suffix intervals. These nodes are called *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.

Secondly, the idea of the space-inefficient $O(n|\Sigma|)$ suffix tree solution mentioned in the introduction is used. Maintain $|\Sigma|$-length arrays for a selected subset of nodes, a subset that contains no more than $\frac{n}{|\Sigma|}$ nodes, which maintains the $O(n)$ space bound. To choose this selected subset of nodes define the following.

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

See Figure 2 for an illustration of the different node types.

The following property of branching-σ-nodes is crucial to our result.

Lemma 1 *Let S be a string of size n over an alphabet Σ and let $ST(S)$ be its suffix tree. The number of branching-σ-nodes in $ST(S)$ is $O(\frac{n}{|\Sigma|})$.*

Proof: The number of σ-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 σ-nodes and

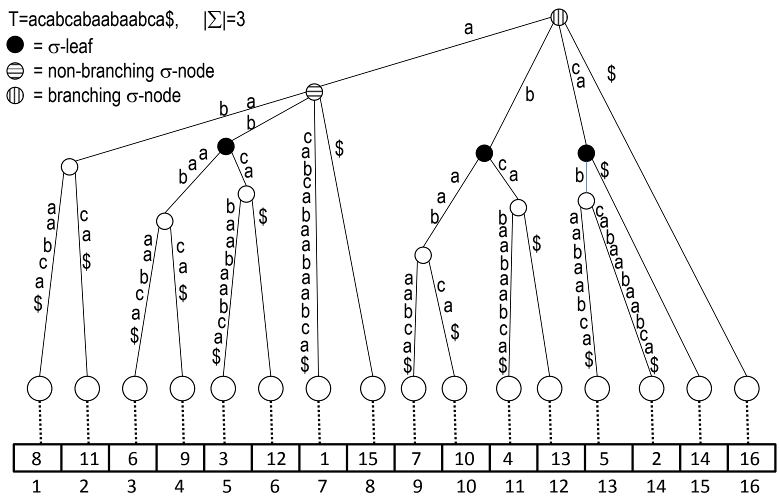

Figure 2: A suffix tree with different node types.

contracted onto the branching-σ-nodes and σ-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 one can afford to maintain arrays at every branching-σ-node which will be very helpful in answering queries as shall be seen in subsection 3.2.

3.1 Suffix Tray Construction

A *suffix tray* is constructed from a suffix tree as follows. The suffix tray contains all the σ-nodes of the suffix tree. Some suffix interval nodes are also added to the suffix tray as children of σ-nodes. Here is how each σ-node is converted from the suffix tree to the suffix tray.

- σ-leaf u: u becomes a suffix interval node with suffix interval $[L(u), R(u)]$.
- non-leaf σ-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 σ-nodes. Then u will be in the suffix tray with interleaving suffix interval nodes and σ-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-σ-node u in the suffix tray, maintain an array of size $|\Sigma|$, denoted by A_u, that contains the following data. For every child v of u that is a σ-node, location τ in A_u where τ 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 σ-node u which is not a branching-σ-node and not a σ-leaf, i.e. it has exactly one child v which is a σ-node, store the first character τ on the edge (u, v), which is called the *separating character*. See Figure 3 for an example of a suffix tray.

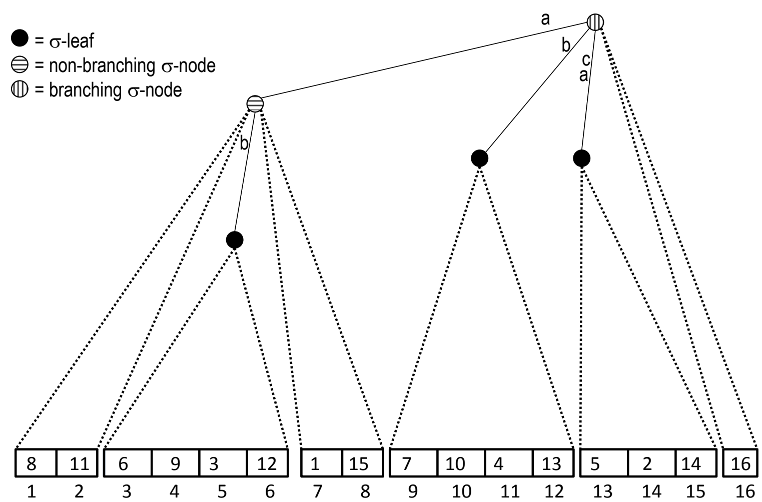

Figure 3: A suffix tray (on running example) with chunks of suffix array at the bottom.

The suffix tray is now claimed to be of linear size.

Lemma 2 *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-σ-node the auxiliary information is of constant size.

The auxiliary information held in each branching-σ-node is of size $O(|\Sigma|)$. By Lemma 1 there are $O(\frac{n}{|\Sigma|})$ branching-σ-nodes. Hence, the overall size is $O(n)$. ▮

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 [5, 8, 9, 10], so can suffix trays.

3.2 Navigating on Index Queries

It is time to turn to the important feature of suffix trays, answering index queries.

Upon receiving a query $P = p_1 ... p_m$, begin by traversing the suffix tray from the root. Assume that the suffix tray has been traversed with $p_1 ... p_{i-1}$ and needs to continue with $p_i ... p_m$. At each branching-σ-node u 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 constant time. For other σ-nodes that are not σ-leaves and not branching-σ-nodes, compare p_i with the separator character τ. Recall that these nodes have only one child v that is a σ-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 $p_i < \tau$, navigate to u's left interval. If $p_i > \tau$, navigate to u's right interval. If $p_i = \tau$, navigate to the only child of u that is a σ-node. If u is a σ-leaf then it is at u's suffix interval.

To search within a suffix interval $[j, k]$. 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 3 *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 Observation 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 σ-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 σ-leaves. As none of the children of v are σ-nodes this is a special case of the (i, j)-suffix interval. ∎

By the discussion above and Lemma 3 the running time for answering an indexing query is $O(m + \log |\Sigma|)$. Summarizing the discussion of the whole section the following is claimed.

Theorem 1 *Let S be a length n string over an alphabet Σ. 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 Suffix Trists - The On-line Scenario

In this section the problem of how to build an indexing structure that supports both text extensions and supports fast(er) indexing is addressed. It is shown that if there exists an on-line 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 one can construct an on-line 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, the on-line linear-space suffix-tree construction is treated as a suffix-tree oracle that provides 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 resented is called the Suffix Trist.

The suffix trist imitates the suffix trays. The σ-nodes and branching-σ-nodes are still used in the suffix tree, and the method for answering indexing queries is similar. However, new issues arise in the on-line 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-σ-nodes become σ-nodes, and σ-nodes become branching-σ-nodes).

4.1 Balanced Indexing Structures and Suffix Trists

Since the data is arriving online, the data structures representing the text needs to be dynamic. On the other hand, there is a need for data structures that work efficiently in the worst case. Therefore, special dynamic data structures over the lexicographically ordered collection of the suffixes are considered. The two data structures to be used are: (1) a *suffix list* which is a doubly linked list of the lexicographically ordered suffix collection, and (2) a *balanced indexing structure, BIS* for short, which is a balanced binary search tree over the lexicographically ordered suffix collection. Amir et.al. in [1] augmented the suffix list with the dynamic order maintenance data structure of Dietz and Sleator [4]. The BIS contains a suffix list within. Moreover, it contains the longest common prefix (LCP) data structure from [6] in order to allow indexing queries to be answered efficiently. Amir et.al. [1] showed how the BIS can be updated in $O(\log n)$ time for every *text extension*. Moreover, a BIS supports indexing queries in time $O(m + \log n)$. The $\log n$ term in the indexing query time follows from the height of the tree.

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

However, things are not as simple as they seem. Insertion of a suffix aS into a BIS for a string S assumes that all the suffixes of S are in the BIS, or more specifically assumes that the suffix S itself is in the BIS. This may not be the case if the BIS is limited to a suffix interval, which contains only part of the suffixes. This problem is addressed and solved in the next subsection.

Also, one still needs to deal with the problem of nodes changing status. The solution presented is a direct deamortized solution and is presented in Section 5.

4.2 Inserting New Nodes into BISs

When a text extension from S to aS is performed, 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 the on-line suffix tree is given, it is shown how to update the suffix trist using the suffix tree (updated by the oracle). The problems are (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 are handled in Section 5. The focus is on solving (1) while mentioning that (2) can be easily solved by BIS tricks in $O(\log |\Sigma|)$ time.

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

Observation 2 *For a node u in a suffix tree, if u is not a σ-node, then all of the leaves in u's subtree are in the same BIS.*

Proof: Being that the each BIS corresponds to an interval of suffixes in the suffix list, and that each of u's leaves are in the same interval (as otherwise it would have been a σ-node), then all of the leaves in u's subtree must be 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, consider the following 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 σ-node, then from Observation 2 it is known 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) one can find the root of the BIS which needs to include the new node u. If v is a σ-node, then the root of the appropriate BIS can be located in constant time: if v is a branching-σ-node, then the BIS is found in constant time from the array in v (it will be guaranteed that this holds in the on-line setting as well). If v is a σ-leaf then there is only one possible BIS. Otherwise, v is a non-leaf σ-node but not a branching-σ-node, and the correct BIS of the two possible BISs can be found 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). At first u is ignored 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 it is shown how to update the trist to include v, u can be added as was done 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-σ-node. Moreover, the number of leaves in v's subtree is the same as the number of leaves in the subtree rooted at w' (as u is currently being ignored). So, if w' is not a σ-node, v is not a σ-node, and otherwise, v is a σ-node with a separating character that is the first character of the label of edge (v, w'). 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 σ-node, the detection of such an event taking place is detailed.

5.1 Detecting a new σ-node

Let u be a new σ-node and let v be its parent. Just before u becomes a σ-node, (1) v must have already been a σ-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, one will be able to detect when a new σ-node is created by maintaining 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 are associated 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, the counter of v_k in the BIS needs to be increased, 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 was explained in the previous section the newly inserted leaf is ignored), as that child previously maintained the number of leaves in its subtree. Furthermore, from now on only the size of the subtree of the newly inserted node is updated, and not the size of its child subtree.

5.2 Updating the New σ-Node

Let u be the new σ-node (which is, of course, a σ-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 σ-node there was a suffix interval node w that was an (i, j)-suffix interval with a BIS representing it.

Updating the new σ-node will require two things. First, the BIS is split into 3 parts; two new BISs and the new σ-leaf that separates between them. Second, for the new σ-leaf the separating character is added (easy) and a new set of counters for the children of u is created(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 $O(\log |\Sigma|)$ one 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, one can split the BIS into the three desired parts in $O(\log |\Sigma|)$ time. Fortunately, one can find the two nodes in the BIS in $O(\log |\Sigma|)$ time using $leaf(u)$ directly from the BIS using the suffix order pointers.

Creating the new counters is done as follows. Denote the children of u by $u_1, ..., u_k$. 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 methods of binary search trees. It is now shown that there is enough time to update all the counters of $u_1, ..., u_k$ before one of them becomes a σ-node, while still maintaining the $O(\log |\Sigma|)$ bound per update.

Specifically, the counters will be updated during the first k insertions into the BIS of u (following the event of u becoming a σ-node). At each insertion one of the counters is updated. What is required is for the counters to be completely updated prior to the next time they will be used, i.e. in time to detect a new σ-node occurring in the subtree of u. The following lemma is precisely what is needed.

Lemma 4 *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 σ-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 σ-node (it should have already been a σ-node).∎

5.3 When a σ-Leaf Loses its Status

The situation where a σ-leaf becomes a non-leaf σ-node is actually a case that has been covered in the previous subsection. Let v be a σ-leaf that is about to change its status to a non-leaf σ-node. This happens because one of its children v_k is about to become a σ-leaf. Note that just before the change v is a suffix interval node. As in the previous subsection The BIS representing the suffix interval needs to be split 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 σ-Node Becomes a Branching-σ-Node

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

Just before becoming a branching-σ-node v contained a separating character τ, 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 σ-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 σ-leaf v_i, (3) a suffix interval node w_L, (4) a σ-node v_j, and (5) a suffix interval node x. 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 one must use a different approach and spread the construction over some time. A pseudo-amortized solution is given and then it is mentioned how to (really) deamortize it. The following lemma allows this.

Lemma 5 *From the time that w becomes a branching-σ-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-σ-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-σ-node must have at least $2|\Sigma|$ leaves in its subtree, as it has at least two children that are σ-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-σ-node, at least $|\Sigma|$ leaves need to be added into its subtree, as required.∎

This yields the pseudo-amortized result, as one 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 5 that on any given search path at most one branching-σ-node construction is charged, even if several branching-σ-nodes are encountered.

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

Begin 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, one more element is added to the array A_v. Lemma 5 assures that A_v will be constructed before a branching-σ-node that is a descendant of v but not of v_i or v_j is handled.

This scheme allows 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 σ-nodes. This issue is overcome as follows. Continue to maintain the initial separating character τ of v and another separator τ', the first character on the edge (v, v_i), until A_v is fully constructed. If v is encountered during a traversal for a query and the continuation of the traversal is to either v_i or v_j then one can discover this in constant time from τ' or τ. For the rest of the children of v that are σ-nodes, maintain them all in a BST, so that when answering an indexing query, one 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 it is guaranteed by Lemma 5 that if one needs to use the BST of the children that are σ-nodes, then one will not encounter any more branching-σ-nodes afterwards. Thus, at most another $O(log|\Sigma|)$ is added to the query time.

There is one more loose end that needs to be dealt with. When other children of v (other, as opposed to v_i and v_j) become σ-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, A_v is defined slightly differently then the static case. Each entry in A_v will point to the edge whose label begins with the character of that entry, if such an edge exists. If no such edge exists, simply put a NIL. This still allows to spend constant time per branching-σ-node when answering an indexing query. However, when the edge pointed by the appropriate location (during the process of answering a query) is met, the node v' on the other side of the edge is examined. If v' is a σ-node, continue to traverse from there. If v' is not a σ-node, then one 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 σ-node, and its parent is already a branching-σ-node, no more changes are required.

Theorem 2 *Let S be a string over an alphabet Σ. 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] R. Cole and M. Lewenstein. Multidimensional matching and fast search in suffix trees. *Proc. of the Symposium on Discrete Algorithms (SODA)*, 851-852, 2003.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

[4] P.F. Dietz, D.D. Sleator. Two Algorithms for Maintaining Order in a List. In *Proc. of Symposium on Theory of Computing (STOC)*, 1987, 365–372.

[5] M. Farach-Colton, P. Ferragina, S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. of the ACM*, 47(6): 987–1011, 2000.

[6] G. Franceschini and R. Grossi. A General Technique for Managing Strings in Comparison-Driven Data Structures. In *Proc. 31st Intl. Col. on Automata, Languages and Programming (ICALP)*, LNCS 3142, 606–617, 2004.

[7] 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, 372–381, 1999.

[8] J. Kärkkäinen, P. Sanders and S. Burkhardt. Linear work suffix array construction. In *J. of the ACM*, 53(6): 918-936, 2006.

[9] D.K. Kim, J.S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2-4): 126-142, 2005.

[10] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4): 143-156, 2005.

[11] 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.

[12] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.

[13] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[14] P. Weiner. Linear pattern matching algorithm. *Proc. 14th IEEE Symposium on Switching and Automata Theory*, 1–11, 1973.