On the Sorting-Complexity of Suffix Tree Construction *

Martin Farach-Colton† Rutgers University

Paolo Ferragina‡ Università di Pisa, Italy

S. Muthukrishnan§ AT&T Shannon Labs

Abstract

The suffix tree of a string is the fundamental data structure of combinatorial pattern matching. We present a recursive technique for building suffix trees that yields optimal algorithms in different computational models. Sorting is an inherent bottleneck in building suffix trees and our algorithms match the sorting lower bound. Specifically, we present the following results.

1. Weiner [Wei73], who introduced the data structure, gave an optimal $O(n)$-time algorithm for building the suffix tree of an n-character string drawn from a constant-size alphabet. In the comparison model, there is a trivial $\Omega(n \log n)$-time lower bound based on sorting, and Weiner's algorithm matches this bound. For integer alphabets, the fastest known algorithm is the $O(n \log n)$ time comparison-based algorithm, but no super-linear lower bound is known. Closing this gap is the main open question in stringology [Gal85]. We settle this open problem by giving a linear time reduction to sorting for building suffix trees. Since sorting is a lower-bound for building suffix trees, this algorithm is time-optimal in every alphabet model. In particular, for an alphabet consisting of integers in a polynomial range we get the first known linear-time algorithm.
2. All previously known algorithms for building suffix trees exhibit a marked absence of locality of reference, and thus they tend to elicit many page faults (I/Os) when indexing very long strings. They are therefore unsuitable for building suffix trees in secondary storage devices, where I/Os dominate the overall computational cost. We give a linear-I/O reduction to sorting for suffix tree construction. Since sorting is a trivial I/O-lower bound for building suffix trees, our algorithm is I/O-optimal.

1 Introduction

Given a string $S \in \Sigma^n$, the *suffix tree* T_S of S is the compacted trie of all the suffixes of $S\$$, $\$ \notin \Sigma$. The suffix tree is the basic data structure in combinatorial pattern matching because of its many elegant uses. Furthermore, it has a compact $O(n)$ space representation that can be constructed in $O(n)$ optimal time for constant-size alphabet [Wei73]. The original construction and its analysis are nontrivial. Some effort has been spent on producing simplified linear time algorithms [CS85, McC76], though all such efforts have been variants of the original approach of Weiner.

The construction of suffix trees remains an active area of research [Kos94, SV94, Har97], but several issues remain open. In this paper we introduce a new suffix tree construction algorithm and address some important issues left open.

*Part of the results contained in this paper have appeared in [Far97, FM96, FFM98]. We fully describe the results from [Far97] in Section 3 and those from [FFM98] in Section 4. We describe the underlying ideas from [FM96]; the main result there has many PRAM-specific complications that obscure the combinatorial structure in our approach, and we do not present those details here.

†Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. (*farach@cs.rutgers.edu*, *http://www.cs.rutgers.edu/~farach*). Supported by NSF Career Development Award CCR-9501942, NATO Grant CRG 960215, NSF/NIH Grant BIR 94-12594-03-CONF and an Alfred P. Sloan Research Fellowship.

‡Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. (*ferragin@di.unipi.it*, *http://www.di.unipi.it/~ferragin*). Part of this work has been done while the author was a Post-Doc student at Max-Planck-Institut für Informatik, Saarbrücken (Germany), and has been partially supported under Italian MURST project "Algorithms on Large Data Sets: Science and Engineering".

§AT&T Labs-Research, Florham Park, NJ 07932. Work was done when the author was at Bell Labs, Lucent Technologies. *muthu@research.att.com.*

1.1 Our Approach

Almost all previous algorithms inserted one suffix at a time into a growing suffix tree. In this paper we will follow another natural approach that consists of building suffix trees via *divide and conquer*. In particular, our approach is as follows.

1. We will show how to recursively compute the compacted trie of all suffixes of S beginning at odd positions. We call this tree T_o, the *odd tree*.
2. From T_o we will show how to compute the *even tree* T_e, the compacted trie of suffixes of S beginning at even positions.
3. Finally, we will show how to merge T_o and T_e into the whole suffix tree T_S.

It is not difficult to implement the first two steps efficiently, although their algorithmic details vary from one computational model to another. The last merging step is, however, recognized as being difficult, so that the efficiency of the overall approach boils down to the effective implementation of the merging between the two trees T_o and T_e. This was stated as an open problem for the PRAM model [Vis94a] and in fact PRAM suffix tree construction algorithms that attempted this divide-and-conquer approach had to deviate substantially from the odd/even merging step [Vis94b, Har94, SV94, Har97].

In this paper we elucidate novel structural properties of odd and even trees, and design new algorithms for merging T_o and T_e that are optimal in many computational models such as the RAM [Far97], the PRAM [FM96], and various external memory models [FFM98]. As a consequence, our odd/even approach yields optimal suffix-tree construction algorithms in all these models.

In the rest of the paper we will only present results for the RAM model and the external memory models, since the PRAM merging algorithm has many PRAM-specific complications that obscure the combinatorial structure of the approach. The two algorithms we present use almost all of the combinatorial structure available for merging, and the problems that these merging algorithms solve are of more topical interest.

1.2 Our Results

Before investigating the complexity of suffix tree construction in the two models above, we discuss the desired format of our output. Suffix trees are often used as indices. Therefore, the operation they should support is that of tracing from the root with some query pattern. Thus, one typically assumes a format for the suffix tree in which the edges leaving a node are lexicographically sorted according to the first character of the string labeling them. We call any trie so sorted a *sorted trie*. In this representation, sorting is an obvious lower-bound for building suffix trees since we can use the suffix tree to retrieve the sorted order of the input characters in linear time.

In all models considered, we will design algorithms in which sorting is the bottleneck, thus establishing that the complexity of sorting and suffix tree construction match.

RAM. The time needed to build a suffix tree depends on the size of the alphabet from which the string is drawn. In addition to the case of constant-size alphabet, there are two other interesting cases to consider: the case of unbounded alphabet, in which string characters can only be manipulated by comparison in some total order on $\Sigma \cup \{¥\}$; and the case of integer alphabet, in which characters are drawn from a range of integers.

In the former case there is a lower bound of $\Omega(n \log n)$ for suffix tree construction[1]. Any linear-time algorithm for suffix tree construction on constant-size alphabet directly gives an optimal-time algorithm for this case too.

The case of integer alphabet is more interesting. If we insist that the edges at each node be sorted, then integer sorting is still a lower bound. Therefore we can assume that the input characters have been sorted, in whatever time it takes to sort those integers [Tho98]. We can then replace each character by its rank in the sorted list so that the resulting string consists of integers in the range $\{1, \ldots, n\}$, which we abbreviate

[1] In this case, it is possible to give a "stronger" lower bound. The $\Omega(n \log n)$ bound can be obtained from element distinctness, even without assuming the sorted output form.

as $[n]$. Observe that the suffix tree of the new string is the same as that of the old string, since this alphabet mapping preserves the lexicographical ordering on substrings.

There is no superlinear lower bound for building the suffix tree of a string from alphabet $[n]$, while the best upper bound known is the straightforward $O(n \log n)$. Note that in [DK95], Delcher and Kosaraju claimed a linear time algorithm for this problem, but this algorithm turned out to have a bug in it, as they noted in [DK96], where they also posed closing the *log-gap* as an important open problem.

In this paper, we solve this open problem and close the gap by giving a linear time algorithm for the $[n]$-alphabet case, thus actually providing a linear time reduction of suffix tree construction to alphabet sorting.

DAM. Known suffix tree construction algorithms exhibit a marked absence of *locality of references,* and therefore they elicit many disk accesses (I/Os) when the size of the indexed string is too large to be fit into the internal memory of the computer. This is a serious problem because I/Os are expensive and thus the performance of these algorithms can be dramatically affected by their pattern of disk accesses. For some algorithmic problems, the issue of carefully structuring external-memory access patterns to overcome this bottleneck has been addressed and is well-understood [Vit], but for suffix tree construction and string processing in general, many issues have not been settled. In this paper we study the suffix-tree construction problem in external memory by assuming that the string is drawn from an unbounded alphabet and by considering a *Disk Access Machine* (DAM) model to evaluate the performance of external-memory algorithms.

In particular, we refer to the DAM model introduced by Vitter and Shriver [VS94][2] where a computer consist of a processing unit, an internal memory of size M and an arbitrarily large external memory partitioned into transfer blocks, called *disk pages.* Each page contains B items, e.g., integers, characters, pointers; and each disk access transfers into the internal memory a *single* disk page, called an *I/O.* Note that $M < n$ and $1 \leq B \leq M/2$. Since I/Os are expensive, a simple way to evaluate the efficiency of external-memory algorithms in the DAM is to count the total number of I/Os performed by the various operations [VS94]. This accounting scheme has gained popularity [Vit] but it does not accurately predict the running time of algorithms on real machines. More accurate disk models have been proposed [RW94, VS94, Vit, Shr97] but they are complex because it is virtually impossible to exploit all the fine points of disk characteristics systematically when designing algorithms or writing code.

Since we do not want to complicate the following discussion by introducing specific trade-offs or other technicalities that would simply obscure the combinatorial structure of the proposed method, we choose to use a notation that is simple, intuitive and independent of the accounting scheme. Since sorting is an obvious I/O-lower bound to the complexity of suffix tree construction in the DAM model, we define $Sort(n)$ as the I/O-complexity of sorting n items on a DAM and describe the complexity of the proposed external-memory algorithms in terms of $Sort(n)$.

In particular, we show how to reduce suffix tree construction to sorting and to a few *low-I/O* primitives, such as scanning a set of items stored contiguously on disk. The resulting external-memory algorithm will be the first I/O-optimal algorithm for suffix tree construction in the DAM. Its complexity will be described in terms of $Sort(n)$, which will be optimal in different ways of counting the memory accesses and modeling the memory bottleneck.

1.3 Organization

The paper is organized as follows. In Section 2, we present some preliminary observations on suffix trees that we will use for our algorithms. In Section 3, we present the algorithm for integer alphabets. In Section 4, we present the I/O-optimal algorithm. Finally, in Section 5, we comment on optimal algorithms for suffix tree construction on other computational models; these can be derived from suitably modifying the divide-and-conquer structure of the algorithms presented in the following sections.

[2]This model has also been called the *two-level memory model* [VS94], and some variants include the *hierarchical memory model* [ACS87, ACF90], the *block transfer model* [ACS87], etc.

2 Preliminaries

2.1 Properties of Suffix Trees

Let $S \in \Sigma^n$ be a string formed by n characters drawn from Σ. The suffix tree T_S of S is the compacted trie[3] of all the suffixes of $S¥$, where $¥ \notin \Sigma$. Throughout the paper, we will assume that suffix trees are represented as follows. Leaf l_i represents suffix $S[i, n]$, and array entry $l[i]$ points to l_i. Each internal node v has a length $L(v)$ that is the sum of the edge lengths on the path from the root to v. Then the string at v, denoted $\sigma(v)$, is $S[i, i + L(v) - 1]$ where l_i is any leaf below v. The children of node v are stored in a list sorted by the first character on the edge from v. Neither these first characters, nor any other part of the string represented by any edge is stored, since it can be retrieved in linear time. See Figure 2.1 for an example. In this representation, the size of the suffix tree is $O(n)$.

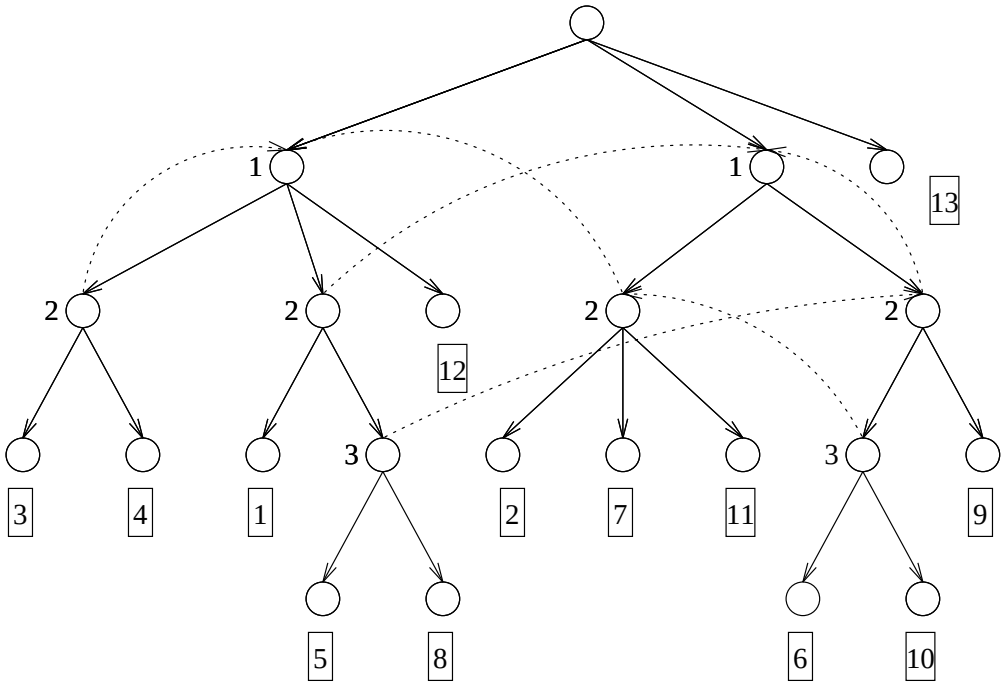

Figure 2.1: Suffix tree of string $S = 121112212221¥$. Numbers in boxes represent which suffix is denoted by that leaf. Numbers next to internal nodes represent the string length $L(\cdot)$ of that node. Dotted lines are suffix links (See Lemma 2.1).

The following well-known lemma gives suffix trees a nice structure.

Lemma 2.1 ([Wei73]) *Let $a \in \Sigma$ and $\alpha \in \Sigma^*$. If there is a node v in T_S such that $\sigma(v) = a\alpha$, then there is a node w in T_S such that $\sigma(w) = \alpha$.*

Given this lemma we can define, for every node v in T_S, the *suffix link* $sl(v) = w$, where v and w are defined as in Lemma 2.1. Notice that $sl(\cdot)$ links form a tree rooted at the root of T_S. The depth of any node v in this $sl(\cdot)$ tree is then just $L(v)$.

Given a string $\alpha \in \Sigma^m$, let $|\alpha|$ denote the string length m. We use $\texttt{lcp}(\alpha, \beta)$ to indicate the *length of the longest common prefix* of any two strings α and β, and $\texttt{lca}(v, w)$ to denote the *least common ancestor* of any two nodes v and w in a tree. The property of suffix trees most often exploited algorithmically is the following relationship between `lcp` in S and `lca` in T_S.

[3] A compacted trie differs from a trie in that maximal branch-free paths are replaced by edges labeled by the appropriate substring.

$$\forall v, w \in T_S, \qquad \texttt{lcp}(\sigma(v), \sigma(w)) = |\sigma(\texttt{lca}(v, w))|. \tag{1}$$

In all computational models, we will use the fact that it is easy to find the least common ancestors of two nodes [HT84, BFC00, CGG+95].

2.2 Suffix Arrays and Euler Tours

Let $\Delta = \{S_i \mid S_i \in \Sigma^{n_i}\}$ be a set of strings of arbitrary lengths and assume, for the sake of exposition, that no string in Δ is a prefix of another. We denote by T the sorted compacted trie of the strings in Δ, and adopt the same notation introduced for suffix trees in Section 2.1 to refer to T's structure. In particular, leaf l_i will now represent the string S_i, and array entry $l[i]$ will point to l_i. The size of T is $O(|\Delta|)$ in addition to the overall length of the strings in Δ. Since T is sorted, the children of any node v are stored in a list sorted by the first character on the edges from v. Observe that the inorder traversal of the leaves of T gives the lexicographically ordered sequence of the strings in Δ. We denote by $s_1, \ldots, s_{|\Delta|}$ the permutation of $1, \ldots, |\Delta|$ such that S_{s_i} is lexicographically smaller than S_{s_j} if and only if $s_i < s_j$.

The suffix array of T consists of two arrays, the *sort array* $\texttt{A}_T$ and the *longest common prefix array* $\texttt{LCP}_T$. $\texttt{A}_T$ is the lexicographically ordered sequence of the strings in Δ, that is $\texttt{A}_T[i] = s_i$. The array $\texttt{LCP}_T$ stores the lengths of the common prefix of adjacent strings in $\texttt{A}_T$, that is $\texttt{LCP}[i] = \texttt{lcp}(S_{s_i}, S_{s_{i+1}})$.

Suppose we are given the sorted compacted trie T and we wish to compute $\texttt{A}_T$ and $\texttt{LCP}_T$. $\texttt{A}_T$ is simply the inorder listing of the leaves of T, whereas $\texttt{LCP}_T$ can be obtained by means of Equality (1). Hence, both of them can be computed simultaneously during an inorder traversal of T. Conversely, given $\texttt{A}_T$ and $\texttt{LCP}_T$ it is a straightforward exercise to reconstruct the trie T in linear time on a RAM [FM96]. We give other model-specific complexities below.

Given a rooted tree $T' = (V, E)$, the *Euler Tour* $\texttt{ET}(T')$ of T' is a sequence of nodes. $\texttt{ET}(T')$ has length $2|E| + 1$ and is obtained by performing a depth-first search (DFS) on T' and outputting each node every time it is visited. In the RAM model, $\texttt{ET}(T')$ is computed via an explicit DFS of the tree T', whereas in the DAM model $\texttt{ET}(T')$ is efficiently computed by simulating known PRAM-algorithms [CGG+95].

3 RAM Algorithm

We start by developing a few tools that will simplify our presentation. Recall that $\texttt{lca}_T(u, v)$ denotes the least common ancestor of a pair of nodes u, v in T. We will drop the T-subscript whenever there is no ambiguity.

Theorem 3.1 ([HT84, BFC00]) *A tree T with m nodes can be preprocessed in $O(m)$ time so that, for any pair of its nodes u, v, $\texttt{lca}(u, v)$ can be computed in constant time.*

Given a sorted compacted trie T built on a set Δ of k strings, we can compute the arrays $\texttt{A}_T$ and $\texttt{LCP}_T$ in $O(k)$ time via a traversal of T (see Section 2.2). Conversely, given $\texttt{A}_T$ and $\texttt{LCP}_T$ we can reconstruct the sorted compacted trie T in linear time by exploiting Theorem 3.1 and Equation 1. Notice that both transformations are independent on the length of the strings stored in the two data structures. Therefore, we can state:

Theorem 3.2 *Given a sorted compacted trie T built on k strings, $\texttt{A}_T$ and $\texttt{LCP}_T$ can be computed in $O(k)$ time. The inverse transformation also takes $O(k)$ time.*

We have now all the ingredients to detail the implementation of the three steps of our divide-and-conquer approach for the case of the RAM model.

3.1 Building the odd tree

We show how to compute T_o, the compacted trie of all suffixes of S beginning in odd positions. Note that we can extend S with any character so that it has length a power of two. At the end of the computation, we simply prune the tree in linear time to remove the extra suffixes. The padding simplifies some boundary

cases below, by making sure that the length of the string is always divisible by two. The padding at most doubles the length of S and so does not affect the time complexity. While the string can be padded with any character, padding with ¥'s makes it easier to strip away unwanted suffixes at the end of the computation.

Step 1. Map pairs of characters into single characters as follows. For $i = 1$ to $n/2$, form pairs $\langle S[2i-1], S[2i]\rangle$. Lexicographically sort them by radix sort in linear time, remove duplicates, and compute the new string S' such that $S'[i]$ is the rank of $\langle S[2i-1], S[2i]\rangle$ in the sorted list. S' is a string of length $n/2$ over the integer alphabet $[n/2]$. For our running example in Figure 2.1, we have that $S' = 212343¥$.

Step 2. Recursively compute $T_{S'}$, the suffix tree of S', and derive $\mathtt{A}_{T_{S'}}$ and $\mathtt{LCP}_{T_{S'}}$. See Figure 3.2 for $T_{S'}$ for our example string. $\mathtt{A}_{T_{S'}} = [2,1,3,4,6,5,7]$ and $\mathtt{LCP}_{T_{S'}}[0,1,0,1,0,0]$.

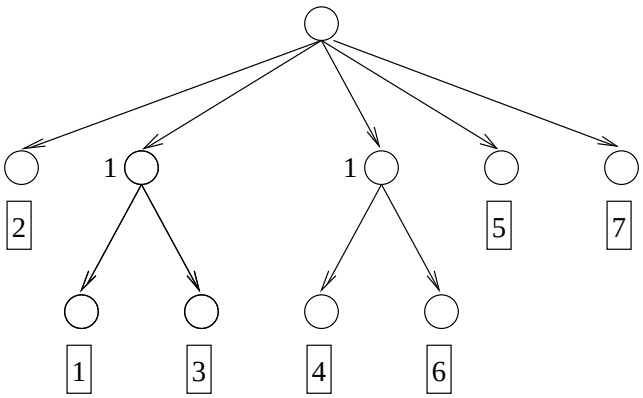

Figure 3.2: Suffix tree of string $S' = 212343¥$

Step 3. Notice that any odd suffix $S[2i-1]\ldots S[n]¥$ is equivalent to the suffix $S'[i]\ldots S'[n/2]¥$. Thus, by the lexicographic ordering of the characters of S', we have that $\mathtt{A}_{T_o}[i] = 2*\mathtt{A}_{T_{S'}}[i]-1$, and so this computation takes linear time. For our example, $\mathtt{A}_{T_o} = [3,1,5,7,11,9,13]$.

Step 4. Since each character in S' represents two characters in S, in order to compute $\mathtt{LCP}_{T_o}$ we need only to check whether the `lcp` between two adjacent suffixes in $\mathtt{A}_{T_o}$ must be extended by one unit. Thus,

$$\mathtt{LCP}_{T_o}[i] = 2 * \mathtt{LCP}_{T_{S'}}[i] + \begin{cases} 1 & \text{if } S[\mathtt{A}_{T_o}[i] + 2 * \mathtt{LCP}_{T_{S'}}[i]] = S[\mathtt{A}_{T_o}[i+1] + 2 * \mathtt{LCP}_{T_{S'}}[i]]; \\ 0 & \text{otherwise.} \end{cases}$$

This computation takes linear time. For our example, $\mathtt{LCP}_{T_o}[i] = [1,2,0,2,1,0]$.

Step 5. Construct T_o from $\mathtt{A}_{T_o}$ and $\mathtt{LCP}_{T_o}$ in linear time (Theorem 3.2). Figure 3.3 gives the odd tree for our example string.

Lemma 3.3 *If $T(n)$ is the time it takes our algorithm to build the suffix tree of a string $S \in [n]^n$, then T_o can be built in $T(n/2) + O(n)$ time.*

3.2 Building the even tree

We construct T_e by first computing the two arrays $\mathtt{A}_{T_e}$ and $\mathtt{LCP}_{T_e}$, and then deriving the even tree in linear time, according to Theorem 3.2.

Step 1. Preprocess the tree T_o in order to answer `lca`-queries in constant time (Theorem 3.1).

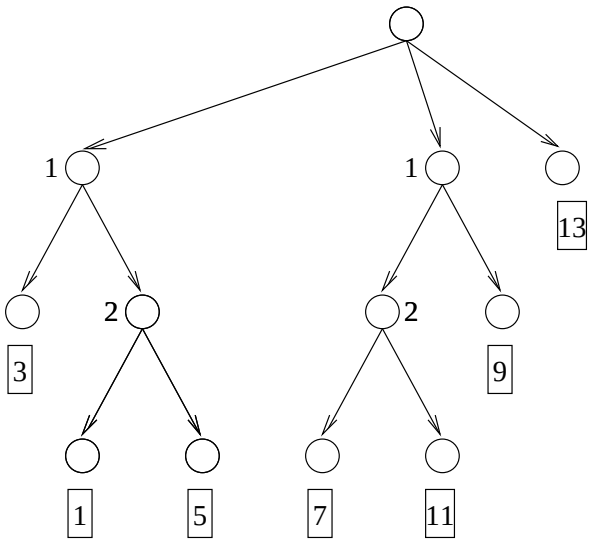

Figure 3.3: Odd tree of string $S = 121112212221¥$

Step 2. Construct $\mathtt{A}_{T_e}$ by observing that an even suffix of S is a single character followed by an odd suffix of S. Thus, the construction of this array boils down to a radix sort problem. We need only to stably sort $\mathtt{A}_{T_o}$, using the key $S[2i]$ for any odd suffix $S[2i+1, n]$. By the correctness of radix sort [AHU74], the sorted sequence so obtained correctly gives $\mathtt{A}_{T_e}$, and this sorting process takes linear time.

In our example string, $\mathtt{A}_{T_o} = [3, 1, 5, 7, 11, 9, 13]$ and thus we stably sort the following tuples: $[\langle S[2], 3\rangle, \langle S[4], 5\rangle, \langle S[6], 7\rangle, \langle S[10], 11\rangle, \langle S[8], 9\rangle, \langle S[12], 13\rangle] = [\langle 2, 3\rangle, \langle 1, 5\rangle, \langle 2, 7\rangle, \langle 2, 11\rangle, \langle 1, 9\rangle, \langle 1, 13\rangle]$. Stably sorting by the first number yields the ordering $[\langle 1, 5\rangle, \langle 1, 9\rangle, \langle 1, 13\rangle, \langle 2, 3\rangle, \langle 2, 7\rangle, \langle 2, 11\rangle]$. Subtracting one from the second index completes the computation and yields $\mathtt{A}_{T_e} = [4, 8, 12, 2, 6, 10]$.

Step 3. The construction of $\mathtt{LCP}_{T_e}$ exploits again the observation that an even suffix is a single character followed by an odd suffix. Hence, given any two even suffixes $S[2i, n], S[2j, n]$ adjacent in $\mathtt{A}_{T_e}$, it is:

$$\mathtt{lcp}(S[2i, n], S[2j, n]) = \begin{cases} \mathtt{lcp}(S[2i+1, n], S[2j+1, n]) + 1 & \text{if } S[2i] = S[2j]; \\ 0 & \text{otherwise.} \end{cases}$$

¿From the preprocessing of T_o (Step 1), the `lcp`-query on the odd suffixes can be answered in constant time, so that the construction of $\mathtt{LCP}_{T_e}$ takes linear time. For our example string, $\mathtt{LCP}_{T_e} = [1, 1, 0, 1, 3]$.

Step 4. Tree T_e is derived from the two arrays $\mathtt{A}_{T_e}$ and $\mathtt{LCP}_{T_e}$ in linear time (Theorem 3.2). These operations applied on our example string yield the tree in Figure 3.4.

Lemma 3.4 *Given $S \in [n]^n$ and its odd tree T_o, the even tree T_e can be constructed in $O(n)$ time.*

3.3 Merging the odd and even trees

In this section, we show how to merge T_e and T_o into T_S via their suffix arrays. In particular, we illustrate how to compute the two arrays $\mathtt{A}_{T_S}$ and $\mathtt{LCP}_{T_S}$ from $\mathtt{A}_{T_o}, \mathtt{A}_{T_e}$ and $\mathtt{LCP}_{T_o}, \mathtt{LCP}_{T_e}$. The final suffix tree T_S is then derived from these two arrays in linear time by Theorem 3.2. This routine runs in linear time, thus completing the suffix tree construction algorithm.

The obvious procedure to merge $\mathtt{A}_{T_e}$ and $\mathtt{A}_{T_o}$, trivially, is to merge the two lists by lexicographic order. The lexicographic order of an odd and even string can be determined quickly if we know the longest common prefix of the two suffixes. Thus, the efficiency of the merging process boils down to the efficiency of odd/even suffix `lcp` computation. Since each suffix may represent a string of up to $\Omega(n)$ characters, we cannot compare

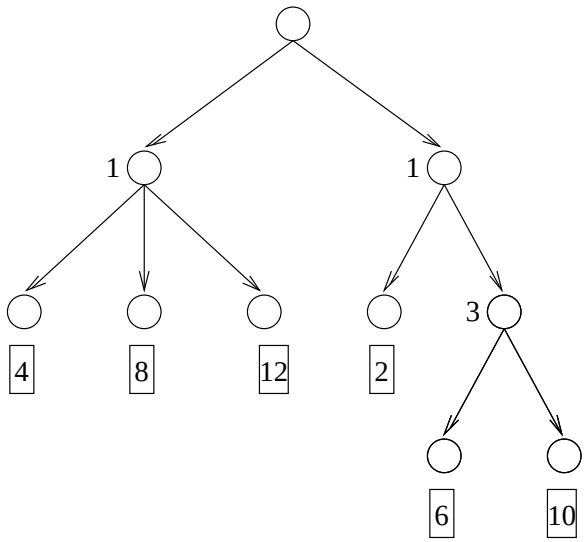

Figure 3.4: Even tree of string $S = 121112212221¥$

character-by-character each pair of suffixes. This would give an $O(n^2)$ time algorithm for merging. Linear time merging of $\mathtt{A}_{T_e}$ and $\mathtt{A}_{T_o}$ could be possible if we could compute the needed `lcp` in constant time. In that case, the `LCP` array would be trivial to compute.

Lemma 3.5 *Given $S \in [n]^n$, its suffix tree T_S can be constructed by merging the odd tree T_o with the even tree T_e in $O(n)$ linear time, provided that a constant-time oracle for computing the longest common prefix between odd and even suffixes is available.*

We now address the problem of building such an `lcp` oracle.

Step 1. In this step, we perform what we refer to as an *over-merge* of T_o and T_e. An over-merge will be very similar to an actual merging of T_o and T_e. Of course, if we could actually merge T_o and T_e directly, we would be done, so instead, we show how to perform a faulty merge, which will be good enough to construct our `lcp` oracle.

Suppose we wanted to merge two *uncompacted* tries. Then we would start at the roots, and find the correspondence between the edges descending from one root and those descending from the other root. For example, we would find the edge labeled 1 descending from one root and identify it with the edge labeled 1 descending from the other root. We could then recursively merge the subtrees below the newly merged 1-nodes. When this is done, we would continue with the descendant edges labeled 2, and so forth. If at some point, one root has a descendant labeled i but the other does not, we would simply include the entire subtree below the i-edge without modification. We call this straightforward trie merge a *Coupled-DFS* and note that if the tries are sorted, the procedure runs in linear time.

How should this be modified for over-merging uncompacted tries? First, we can label each edge by the first character on the string labeling that edge. We know that if an edge incident on the root of one tree is labeled with a character and an edge incident on the root of the other tree is labeled with the same character, these edges must correspond to each other in that in the merged tree, these two edges must be at least partially merged. Since we only consider sorted compacted tries, this correspondence can be found in time linear in the number of children to be merged.

Now consider two edges of the compacted tries that we have determined to correspond to each other. Suppose that the strings labeling these edges have different lengths, say k and l, where $k < l$. Then we break the longer edge into two edges so that the top edge has length k and the other has length $l - k$. Now, if we only knew how far to merge the two top edges, we would be done, in that this is all that is required to finish

merging the two trees. But this would require an `lcp` oracle, which is exactly what we are after. However, as noted above, we only hope to over-merge the trees, so we can safely merge the entire corresponding pair of edges, simply based on their first character. We then recurse with the pendant subtrees. Thus, we perform a coupled-DFS, with the two modifications that we break edges so that their lengths equal that of their corresponding edges, and we identify edges with just their first character.

Note that these two modifications do not change the time complexity of the procedure, which is therefore linear overall. Let us call the new tree T_M, and preprocess it for `lca`-queries (Theorem 3.1), also in linear time. See Figure 3.5 for T_M on our example string.

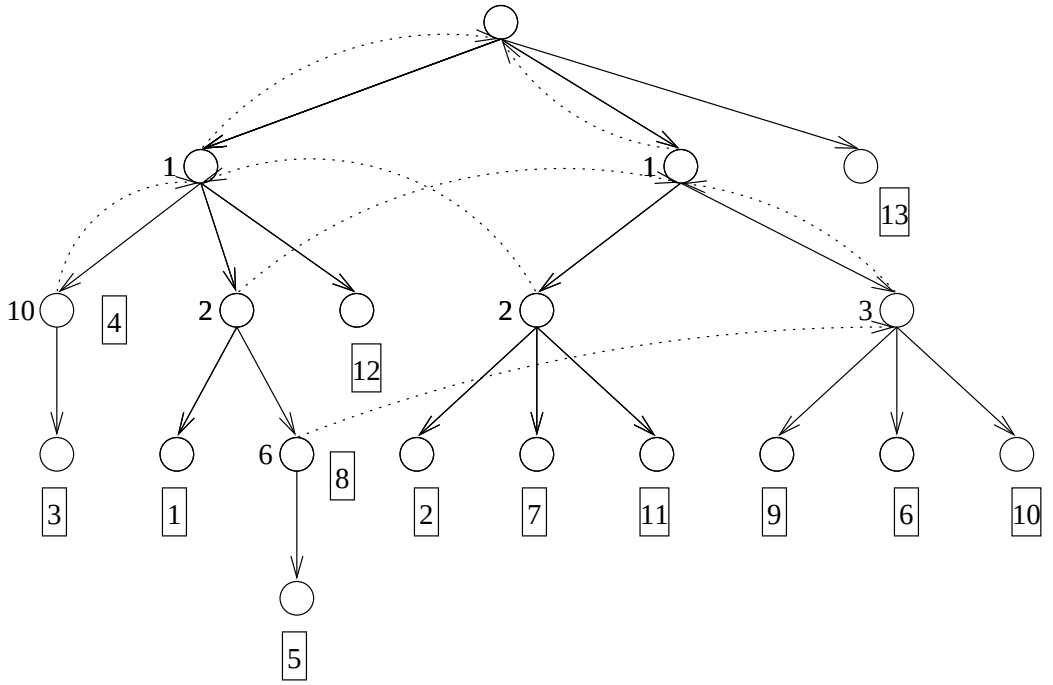

Figure 3.5: Overmerging even and odd trees of string $S = 121112212221\$$ yields T_M. The dotted lines are the $d(\cdot)$ tree of Step 2.

Step 2. Any node of T_M is called *odd* if it occurred in T_o, and *even* if it occurred in T_e. Notice that a node, for example the root, can be both odd and even. Additionally, let us call any node of T_M with both odd and even descendants, an *odd/even* node.

Let u be an odd/even node, other than the root, and let l_{2i} and l_{2j-1} be descendants such that $u = \texttt{lca}(l_{2i}, l_{2j-1})$. In linear time we can find such a pair for every node u as follows. Perform a DFS on the tree, computing for each node an odd and even descendant, if it has both. Now, we need only pick an even descendant from one child of u and an odd descendant from another child of u to achieve an odd/even pair whose `lca` is u. Let $d(u) = \texttt{lca}(l_{2i+1}, l_{2j})$. We can compute all such d's in linear time given the preprocessing of T_M (Step 1).

Notice that if T_M were a suffix tree, then $sl(u) = d(u)$. In this case, we would have that d-pointers form a tree. We will show that, even though d-pointers are not suffix links, they do form a tree (see Lemma 3.6).

Step 3. Let $\hat{L}(u)$ equal the depth of u in the tree defined by d-pointers. Compute $\hat{L}$ in linear time by a DFS traversal of the tree defined by the d-pointers. Notice, for example, that in Figure 3.5, $\hat{L}(\texttt{lca}(l_5, l_8)) = 3$, $\hat{L}(\texttt{lca}(l_6, l_9)) = 2$ and $\hat{L}(\texttt{lca}(l_9, l_{10})) = 2$. Function $\hat{L}(\cdot)$ gives the constant-time oracle we are searching for.

Lemma 3.6 *The function d defines a tree on the odd/even nodes of T_M, and for any l_{2i} and l_{2j-1} we have* $\hat{L}(\texttt{lca}(l_{2i}, l_{2j-1})) = \texttt{lcp}(S[2i, n], S[2j-1, n])$.

Proof: Consider any odd/even node u in T_M. Since u is either odd or even, $L(u)$ is defined and takes its value from the original tree.

Let u be an even node, and l_{2i} and l_{2j} be any pair of even leaves below u. Trivially $\texttt{lcp}(S[2i,n], S[2j,n]) \geq L(u)$. Let $l_{2i'-1}$ and $l_{2j'-1}$ be any pair of odd leaves below u. Note that i' need not differ from j' because u need not be an odd node. However, whether u is odd or not, $\texttt{lcp}(S[2i'-1,n], S[2j'-1,n]) \geq L(u)$.

Now, let $l_{2i''}$ and $l_{2j''-1}$ be a pair of leaves such that $u = \texttt{lca}(l_{2i''}, l_{2j''-1})$. We claim that $\texttt{lcp}(S[2i'',n], S[2j''-1,n]) \leq L(u)$. To see this, notice that in the DFS-merging procedure we merge the two paths leading to $l_{2i''}$ and $l_{2j''-1}$ for at least $\texttt{lcp}(S[2i'',n], S[2j''-1,n])$ characters. Symmetrically, we conclude the same things if u is odd. Thus, we have shown that

Claim 3.6A *The* `lcp` *value of* any *odd and even pair of leaves whose* `lca` *is u must be the same.*

Proof: We remark here that if $\texttt{lca}(l_{2i'}, l_{2j'-1}) = \texttt{lca}(l_{2i''}, l_{2j''-1}) = u$, then $\texttt{lcp}(S[2i',n], S[2j'-1,n]) = k \leq L(u)$, for some k. Since $\texttt{lcp}(S[2i',n], S[2i'',n]) \geq L(u) \geq k$, $\texttt{lcp}(S[2i'',n], S[2j'-1,n]) = k$ as well. A similar argument shows that $\texttt{lcp}(S[2i'',n], S[2j''-1,n]) = k$, which establishes the claim. □

We now proceed to prove the Lemma by induction on the length of the `lcp`. If a pair of odd and even suffixes disagree on their first character, then the `lca` of their leaves will be the root since we always merge based on the first character of an edge label. Consequently, the Lemma holds in the base case.

Suppose the Lemma holds for any odd and even pair of suffixes whose `lcp` is less than k. Let l_{2i} and l_{2j-1} be a pair of even and odd leaves such that $\texttt{lcp}(S[2i,n], S[2j-1,n]) = k > 0$. Let $u = \texttt{lca}(l_{2i}, l_{2j-1})$. u is not the root, since $k > 0$. Let $l_{2i'}$ and $l_{2j'-1}$ be the leaves that were used to define $d(u)$. Thus $d(u) = \texttt{lca}(l_{2i'+1}, l_{2j'})$.

Our induction has two parts: we know that the d-pointers form a tree on all nodes where the inductive hypothesis holds, and thus, we know that $\hat{L}$ is defined for such nodes and that it gives the correct `lcp` values. Therefore, $\hat{L}(u) = 1 + \hat{L}(d(u)) = 1 + \texttt{lcp}(S[2i'+1,n], S[2j',n]) = \texttt{lcp}(S[2i',n], S[2j'-1,n]) = \texttt{lcp}(S[2i,n], S[2j-1,n])$; the first equality is by the first part of the induction, the second by the second part of the induction, the third since $k > 0$, and the last by Claim 3.6A. It follows that the d-pointers form a tree, which settles the Lemma. ■

Combining Lemmas 3.3, 3.4, 3.5 and 3.6, we are ready to state the main result of this section:

Theorem 3.7 *Given a string $S \in [n]^n$, the suffix tree T_S of S can be deterministically constructed in $O(n)$ time and space on the RAM model.*

4 DAM Algorithm

We begin by describing some general external-memory tools that will be used subsequently as basic blocks for implementing our DAM algorithms. We then detail the odd and even tree construction algorithms, and finally, show how to merge T_o and T_e. In the rest of this section we assume that the input data to the external-memory algorithms, and their outputs, are stored in a contiguous portion of the disk. Furthermore, as in the RAM case (see Section 3.1), we simplify the presentation by assuming that S is padded with ¥'s so that it has length a power of two, and at the end of the computation, we prune the tree to remove the extra suffixes.

4.1 Tools

From [AV88], we have $Sort(n) = \Theta((n/B)\log_{M/B}(n/B))$ I/Os for the one disk case. In what follows, many steps (such as scanning an n item array) will involve $O(n/B)$ I/Os so that their complexities will be subsumed by $Sort(n)$.

Theorem 4.1 ([CGG+95]) *Let T be a tree of size m, represented by adjacency lists. Then we can solve the following problems in $O(Sort(m))$ I/Os:*

- *Answer a batch of $O(m)$ lca-queries, or answer a batch of $O(m)$ range minima queries on an array A of integers, where $\texttt{rmq}_A(i,j) = \min_{i \le k \le j}\{A[k]\}$ is the* range minimum query *from i to j in A.*
- *Construct the Euler Tour $\texttt{ET}(T)$ or compute the depth of each node.*
- *Reconstruct the adjacency lists of T's nodes from the knowledge of their parent pointers. If T is a (compacted) trie, we can sort its adjacency lists according to the first labeling character of each edge in the same I/O-bound.*
- *If T is a list, compute the* rank *of each element in the list.*

¿From the result above it is simple to derive the following corollary:

Corollary 4.2 *In $O(Sort(m))$ I/Os we can solve the following problems:*

- *Retrieve $k = O(m)$ arbitrary characters from a string $X[1,m]$ stored contiguously on the disk, that is, given a list of indices $i_1, i_2, \ldots, i_k$ possibly with duplicates, the output is the list $X[i_1], X[i_2], \ldots, X[i_k]$.*
- *Given a tree T of size m, in which some of its nodes are marked, determine for every node one of its marked descendants (if any).*
- *Given a tree T of size m in which some of its nodes are marked in such a way that: if u, v are marked then $\texttt{lca}(u,v)$ is marked too, determine for every node its (unique) closest marked ancestor and descendant.*

Proof: For the first problem, sort the m queries according to the starting positions of the characters to be retrieved, and then scan simultaneously this ordered list and the string X in order to retrieve the queried characters. The cost of the sorting step dominates the overall I/O cost.

The solution to the second problem is slightly more involved. Compute the Euler Tour $\texttt{ET}(T)$ (Theorem 4.1), and identify the first and the last occurrence of each node in $\texttt{ET}(T)$ by means of a sorting step. Then, scan $\texttt{ET}(T)$ and keep some nodes in an external-memory stack. When visiting a node v, all the nodes in the stack will be ancestors of v. If v is marked, all nodes in the stack are popped and each of them, say w, forms a pair $\langle\, w, v \,\rangle$ that is stored on disk (v is the marked descendant we have found for w), and we are done with v. Otherwise, if v is a first occurrence, push it on the stack; if v is the last occurrence, pop it from the stack (if it is on the top). This can only happen if v has no marked descendant. We execute a total of $|\texttt{ET}(T)| = O(m)$ pop and push operations. This takes $O(m/B)$ I/Os by keeping the two top pages of the stack in internal memory. Hence the initial sorting step dominates the overall I/O cost.

The previous algorithm can be easily modified to provide a simple solution to the third problem. The way the nodes are marked ensures that the closest marked ancestor and descendant of each node in T are unique. This indeed implies that tree T can be decomposed into paths that possibly touch each other but only on the marked nodes. Hence, the algorithm used to solve the second problem, can be applied to compute for every node its closest marked descendant. For the closest marked ancestor, we can slightly modify this algorithm by now pushing/popping from the external-memory stack *only* the marked nodes encountered during the scanning of $\texttt{ET}(T)$. This way the node on the top of the stack actually identifies the closest marked ancestor of the currently visited node. We execute a total of $|\texttt{ET}(T)| = O(m)$ pop and push operations, thus taking $O(m/B)$ I/Os. The construction of $\texttt{ET}(T)$ takes $O(Sort(m))$ I/Os by Theorem 4.1, thus implying the final I/O-bound. ■

The suffix array data structure is also useful in the external-memory setting to simplify our computations. We will show how to convert between suffix arrays and sorted tries in an I/O-efficient manner and we will exploit this equivalence throughout our algorithm.

Theorem 4.3 *Given a sorted compacted trie T of size m, $\texttt{A}_T$ and $\texttt{LCP}_T$ can be computed in $O(Sort(m))$ I/Os. Given $\texttt{A}_T$ and $\texttt{LCP}_T$, the sorted compacted trie T can be constructed in $O(Sort(m))$ I/Os.*

Proof: Both $\mathtt{A}_T$ and $\mathtt{LCP}_T$ can be computed from T in $O(Sort(m))$ I/Os by exploiting the Euler Tour $\mathtt{ET}(T)$ (Theorem 4.1). Conversely, T can be constructed from $\mathtt{A}_T$ and $\mathtt{LCP}_T$ in phases as follows. After phase i, assume, inductively, that the sorted trie containing the strings $\mathtt{A}_T[1], \ldots \mathtt{A}_T[i]$ and an external-memory stack containing the nodes on the path leading to $\mathtt{A}_T[i]$ have been built. In phase $(i+1)$, create a leaf l_{i+1} representing string $\mathtt{A}_T[i+1]$ and pop all nodes in the stack whose string length is strictly greater than $\mathtt{LCP}_T[i]$. Call u the last popped node and v the node on the top of the stack. If $\mathtt{LCP}_T[i] = L(v)$ then set the parent of l_{i+1} to be v; otherwise create a new node w, set $L(w) = \mathtt{LCP}_T[i]$, set the parent of u and l_{i+1} to be w, and set the parent of w to be v. Finally push w and l_{i+1} onto the stack, thus completing phase $i+1$.

We execute a total of $O(m)$ pop and push operations in $O(m/B)$ I/Os (again keeping the two top pages of the stack in internal memory). The final sorted trie T can be obtained from the parent pointers in $O(Sort(m))$ I/Os (Theorem 4.1). ■

In the rest of this section, we put together all the above tools and design few subroutines that will constitute the basic blocks upon which our final external-memory algorithm will be based. We start with a simple definition:

Definition 4.4 *Let T be an m node compacted trie built on a string set Δ and let Δ' be a subset of Δ. The compacted trie $T_{\Delta'}$ built on the strings of Δ' is called the* skeleton tree *of T with respect to Δ'.*

Theorem 4.5 *Given a set Δ of k strings and its sorted compacted trie T, the skeleton tree $T_{\Delta'}$ can be constructed on any subset of strings $\Delta' \subset \Delta$ in $O(Sort(k))$ I/Os.*

Proof: Construct $\mathtt{A}_T$ and $\mathtt{LCP}_T$ from T in $O(Sort(k))$ I/Os (Theorem 4.3 with $m = O(k)$). Then construct $\mathtt{A}_{T_{\Delta'}}$ by scanning $\mathtt{A}_T$ and selecting the strings of Δ', in $O(k/B)$ I/Os. During this scanning step, also compute the array $\mathtt{LCP}_{T_{\Delta'}}$ by observing that, given two consecutive strings S_{s_i} and S_{s_j} in Δ' (i.e., $S_{s_{i+1}}, \ldots, S_{s_{j-1}}$ do not belong to Δ'), we have $\mathtt{lcp}(S_{s_i}, S_{s_j}) = \min_{i \le h < j}\{LCP_T[h]\}$. Finally, the skeleton tree $T_{\Delta'}$ is derived from the two arrays $\mathtt{A}_{T_{\Delta'}}$ and $\mathtt{LCP}_{T_{\Delta'}}$ in $O(Sort(k))$ I/Os (Theorem 4.3). ■

Theorem 4.6 *Two sorted uncompacted tries with a total of m nodes can be merged in $O(Sort(m))$ I/Os.*

Proof: This sorting process proceeds by simulating a merge driven by a depth-first visit in an I/O-efficient way. DFS-merging consists of identifying the roots of both tries, merge their adjacency lists, and then recursively merge the subtrees descending from edges labeled with the same characters. Such an approach may induce a lot of random accesses to the two trie structures, and thus turns out to be not useful in the external-memory setting. Our solution follows this basic idea but takes advantage of the Euler Tours of the two uncompacted tries as follows.

We associate with each edge its corresponding labeling character in $O(Sort(m))$ I/Os (Theorem 4.1). Then we advance in each Euler-Tour list as long as the characters labeling the two examined edges match; otherwise, output to the disk the Euler tour of the subtree with the path from the root labeled by the lexicographically smaller string. This visit takes $O(m/B)$ I/Os and yields the "merged" Euler Tour. The merged and sorted compacted trie can be obtained from this Euler Tour via another sorting step since we actually know the parent pointer of each node (Theorem 4.1). ■

The final tool we will need in the design of our algorithm, is the one that allows an efficient computation of the suffix-link pointers in a suffix tree (see Section 2.1).

Theorem 4.7 *Given a suffix tree T with a total of m nodes, the suffix links $sl(v)$ and the string lengths $L(v)$ of all internal nodes v in T can be computed in $O(Sort(m))$ I/Os.*

Proof: Mark all suffix-tree leaves and pick, for each internal node v, a pair of (marked) leaves l_i and l_j such that $\mathtt{lca}(l_i, l_j) = v$. Notice that l_i and l_j are leaves descending from any two distinct children of v (Corollary 4.2). We know that $sl(v) = \mathtt{lca}(l_{i+1}, l_{j+1})$, so we can set simultaneously all the $sl()$-links by answering a batch of $O(m)$ $\mathtt{lca}$-queries in $O(Sort(m))$ I/Os (Theorem 4.1). Since the depth in the $sl()$-tree of each node v equals the value $L()$, we can perform the computation of the Ls in $O(Sort(m))$ I/Os by using Theorem 4.1. ■

4.2 Building the odd tree

We present a straightforward modification of the RAM algorithm.

Step 1. Form the pairs $\langle S[2i-1], S[2i]\rangle$, sort them, remove duplicates, and finally compute the new string $S'[i]$ = rank of $\langle S[2i-1], S[2i]\rangle$ in the sorted list. This takes $O(Sort(n))$ I/Os.

Step 2. Recursively compute the suffix tree $T_{S'}$ of S', and then derive the two arrays $\mathtt{A}_{T_{S'}}$ and $\mathtt{LCP}_{T_{S'}}$ in $O(Sort(n))$ I/Os (Theorem 4.3).

Step 3. Compute $\mathtt{LCP}_{T_o}$ by using the relation $\mathtt{A}_{T_o}[i] = 2 * \mathtt{A}_{T_{S'}}[i] - 1$. This step involves the scanning of the array $\mathtt{A}_{T_{S'}}$ thus taking $O(n/B)$ I/Os.

Step 4. As in the RAM, compute $\mathtt{LCP}_{T_o}$ by using the relation:

$$\mathtt{LCP}_{T_o}[i] = 2 * \mathtt{LCP}_{T_{S'}}[i] + \begin{cases} 1 & \text{if } S[\mathtt{A}_{T_o}[i] + 2 * \mathtt{LCP}_{T_{S'}}[i]] = S[\mathtt{A}_{T_o}[i+1] + 2 * \mathtt{LCP}_{T_{S'}}[i]]; \\ 0 & \text{otherwise.} \end{cases}$$

This step involves batched lookups in the arrays $\mathtt{LCP}_{T_{S'}}, \mathtt{A}_{T_o}$ and S, thus taking $O(Sort(n))$ I/Os (Corollary 4.2).

Step 5. Derive T_o from $\mathtt{A}_{T_o}$ and $\mathtt{LCP}_{T_o}$ in $O(Sort(n))$ I/Os (Theorem 4.3).

Lemma 4.8 *If $C(n)$ is the I/O-complexity taken by our algorithm to build the suffix tree of a string $S \in \Sigma^n$, then the odd tree T_o can be built in $C(n/2) + O(Sort(n))$ I/Os.*

4.3 Building the even tree

Every step in Section 3.2 for building the even suffix array from the odd suffix array is simply a sorting step or a batch of $O(n)$ `lca`-queries on the leaves of T_o. These `lca`-queries can be implemented via `rmq`-queries in $\mathtt{LCP}_{T_o}$. Consequently, every step in Section 3.2 can be implemented in $O(Sort(n))$ I/Os (Corollary 4.2).

Lemma 4.9 *Given a string $S \in \Sigma^n$ and its odd tree T_o, the even tree T_e can be built in $O(Sort(n))$ I/Os.*

4.4 Merging the odd and even trees

As in the RAM case, merging is the crux of the DAM algorithm. A first approach to designing a DAM algorithm would be to take the RAM algorithm and simply replace each RAM step with its I/O-efficient analog. There are two steps that do not immediately have I/O-efficient versions: merging the leaf lists, and the coupled-DFS to build the `lcp` oracle. We will show below that we can replace the coupled-DFS with an I/O-efficient routine, based on exploiting the structure of the merged tree. This tree will be central to our discussion, and will refer to it as the *over-merged* tree T_M.

The leaf-list merging routine does not seem amenable to an I/O-solution. The problem is that each merge step requires an `lca` computation. If we use a parallel merge routine to batch the `lca` queries, we get an algorithm with $O(Sort(n) \log\log n)$ I/Os. We will, instead, show how to fix the over-merged tree to build the suffix tree.

We will first prove some structural properties of odd and even trees and show how they can be used to find a *superset* of the odd/even nodes. Then, we exploit this superset of odd/even nodes to construct I/O-efficiently the intermediate (over-merged) tree T_M. Finally, we explain how the suffix tree T_S can be obtained from T_M by properly *detecting* and *undoing* the (over)merged paths.

4.4.1 Finding a superset of the odd/even nodes

For the sake of exposition, let us introduce some definitions and notations that will allow us to characterize the structure of odd/even nodes in a simple way.

Definition 4.10 *An* anchor pair *is a pair of nodes $v_o \in T_o$ and $v_e \in T_e$ such that $\sigma(v_o) = \sigma(v_e)$. Each such node is called an* anchor.

All anchor nodes and their ancestors are odd/even nodes. Consequently, the part of T_o and T_e lying above the anchor nodes is formed by odd/even nodes *only*. Hence, the set of all anchor pairs (and their ancestors) identifies a subset of odd/even nodes. But, what we can say about the part of T_o and T_e that lies below these anchor pairs ? The next definition sheds some light on these odd/even nodes.

Definition 4.11 *A* side tree *is defined to be a maximal component such that no node in it is an anchor node or has an anchor descendant. A* side tree pair *is defined to be a pair of side trees such that the parents of their roots form an anchor pair, and the first character on the edge from the anchor pair to the roots is the same.*

The nice property about side tree pairs is that the rest of the odd/even nodes in them adhere to a simple pattern, which facilitates their retrieval.

Lemma 4.12 *Given a side tree with root u, if some odd/even nodes descend from u, then they form a simple path, one endpoint of which is u.*

Proof: Consider a pair of odd/even nodes u' and u'' in T_M that come from a side tree of either T_o or T_e, and assume that neither is an ancestor of the other. The point in fact is that T_M is formed only by nodes of T_e and T_o. By definition, odd and even leaves descend from both u' and u'' in T_M. Let $l_{2i'}, l_{2j'+1}$ be a pair of even and odd leaves descending from u', and let $l_{2i''}, l_{2j''+1}$ be a pair of even and odd leaves descending from u''. Notice that the node $v_e = \texttt{lca}(l_{2i'}, l_{2j''})$ is even and the node $v_o = \texttt{lca}(l_{2i'+1}, l_{2j''+1})$ is odd, both occur below u, and $\sigma(v_e) = \sigma(v_o)$ because of the assumption above. Thus, (v_e, v_o) is an anchor pair descending from the side-tree root u, contradicting the definition of a side tree. ■

By Definition 4.11, no anchor pair is contained in a side tree pair and therefore there cannot exist a pair of odd/even nodes that coincide during the merging of the side tree pair. Consequently, the two downward paths formed by odd/even nodes in a side tree pair will interdigitate on the final merged path like the teeth of a zipper (these nodes, however, need not alternate in lockstep like the teeth of a zipper). With this in mind, and at the risk of abusing the analogy somewhat, we call the deepest odd/even node of a side tree the *tooth* of the side tree. The following property easily follows.

Property 1 *A node in T_o (resp. T_e) is an odd/even node if, and only if, it is the ancestor of either an anchor node or a tooth node in T_o (resp. T_e).*

This is the key property underlying the I/O-efficient detection of a superset of the odd/even nodes in T_o and T_e. In what follows, we show how to detect the set of anchor nodes in an I/O-efficient manner. Next, rather than computing the set of tooth nodes (which seems I/O-expensive), we show how to efficiently detect a leaf descending from each of them, called a *pull leaf*. The ancestors of these pull leaves will give us a *superset* of the odd/even nodes lying in the side-tree pairs. This will complete the first step of our merging procedure.

Finding anchor nodes. Recall that suffix links form a tree that can be determined using $O(Sort(m))$ I/Os on a suffix tree of size m (Theorem 4.7). Unfortunately, neither T_o nor T_e is a suffix tree and as such the suffix links may not be well-defined for the nodes in each of these trees individually. However, if we consider the two trees simultaneously, suffix links are well-defined: the suffix links of nodes in the odd tree point to nodes in the even tree, and vice versa. These links form two trees, one rooted at the root of T_o and the other at the root of T_e. We call these trees L_o and L_e, respectively, and compute them via a batch of `lca`-queries (Theorem 4.1).

Each edge in L_o and L_e can be labeled with a single character as follows. If $sl(v) = w$ and $\sigma(v) = a\sigma(w)$, for $a \in \Sigma$, label the edge (v, w) with character a. From the definition of suffix links, it follows that all the children of a node in either L_o or L_e are labeled with different characters. Thus L_o and L_e are in fact uncompacted tries.

The following lemma relates L_o and L_e to anchor pairs and is the basis of our algorithm for determining anchor nodes.

Lemma 4.13 *If v_o, v_e is an anchor pair in T_o and T_e, then one of v_o and v_e occurs in L_o and the other occurs in L_e. Furthermore, the path from the appropriate root to each of v_o and v_e is labeled with the reversal of $\sigma(v_o)$ $(= \sigma(v_e))$.*

Proof: Every time a suffix link is traversed, the parity of the length of the string at that node changes. Since $\sigma(v_o) = \sigma(v_e)$, they cannot be in the same tree. Additionally, a simple inductive proof shows that the characters labeling the path from the root, of either L_o or L_e, to v_o are the reversal of $\sigma(v_o)$, and similarly for v_e. ■

At this point it is clear that if we merge L_o and L_e, then we identify the anchor node pairs as those nodes that are merged in the process. Since L_o and L_e are *uncompacted tries*, we can merge them via Theorem 4.6, thus concluding that

Lemma 4.14 *Given a string $S \in \Sigma^n$, its odd tree T_o and its even tree T_e, the set of anchor nodes and anchor-node pairs can be determined in $O(Sort(n))$ I/Os.*

Finding pull leaves in side tree pairs. Once all anchor pairs have been flagged using Lemma 4.14, all side tree pairs can be found in $O(Sort(n))$ I/Os as follows. By Definition 4.11, the side tree pairs are given by pairs of subtrees in T_o and T_e such that: (i) they hang off an anchor pair and (ii) the edges leading to their roots have the same first labeling character. Consequently, they can be determined by building $\texttt{ET}(T_o)$ and $\texttt{ET}(T_e)$ (Theorem 4.1) and then scanning them while consulting the first character labeling each edge in T_o and T_e. Using the basic tools in Corollary 4.2, all these steps take $O(Sort(n))$ I/Os.

Now recall that rather then finding the tooth nodes directly, we determine a leaf descending from each of them, called the *pull leaf*, and then use this leaf to identify a downward path in the side tree. This path identifies a superset of the odd/even nodes in that side tree.

Let us therefore concentrate on the problem of detecting the pull leaves in each side tree pair, say s_o and s_e, taking $O(Sort(n))$ I/Os. We will first present an I/O-inefficient algorithm, and then design an I/O-efficient algorithm that takes $O(Sort(n))$ I/Os, thus completing the presentation. The key fact underlying the algorithms below is the path-like structure of odd/even-nodes (Lemma 4.12).

An I/O-inefficient algorithm. The idea of the inefficient external-memory algorithm is to introduce unary nodes in both trees for all possible L-lengths that occur in either side tree s_o or s_e. Upon doing so, we have reduced the problem of merging compacted tries into one of merging uncompacted tries, which we can solve by Theorem 4.6. So the question is, how many unary nodes do we need to introduce, and therefore how many characters do we need to retrieve?

We need to break every edge in s_o and s_e at every possible relevant length. There are $O(|s_e|)$ L-lengths of nodes in s_e and $O(|s_o|)$ L-lengths of nodes in s_o, and so every edge in either s_e or s_o can be split by no more than $O(|s_e| + |s_o|)$ unary nodes. Hence, the size of the two side trees will be $O((|s_e| + |s_o|)^2)$ after the introduction of unary nodes. Consequently, if we apply Theorem 4.6 on these two uncompacted tries, we get an I/O-inefficient merging algorithm requiring $O(Sort((|s_e| + |s_o|)^2))$ I/Os per side-tree pair. Therefore $O(Sort(n^2))$ I/Os would be necessary to merge all side tree pairs in T_o and T_e.

An I/O-efficient algorithm. We will apply the quadratic algorithm above on smaller and smaller subtrees to find better and better approximation of the pull leaves. Each iteration will take $O(Sort(n))$ I/Os, and we will show that the loop terminates in $O(1)$ rounds, after which we will have detected the pull leaves.

We initially set $\hat{s}_e = s_e$ and $\hat{s}_o = s_o$. During each iteration of the loop (described below), the trees $\hat{s}_e$ and $\hat{s}_o$ will shrink but they keep the nice property that they will always contain the pull leaves we are searching for. Each iteration of the loop consists of the following steps:

1. Take every $\sqrt{|\hat{s}_o|}^{th}$ leaf in $\hat{s}_o$ and every $\sqrt{|\hat{s}_e|}^{th}$ leaf in $\hat{s}_e$ (assume that these square-roots give integers), and form the skeleton trees $\hat{s}'_o$ and $\hat{s}'_e$ from these leaves (Theorem 4.5).

2. Run the quadratic external-memory algorithm on these trees, and produce a pair of "pseudo-pull" leaves l_o and l_e. These need not to be true pull leaves of $\hat{s}_o$ and $\hat{s}_e$, respectively, since they were found by merging sampled trees, rather than the trees s_o and s_e themselves.

3. Perform a *retracing step*, in which the paths $\hat{p}_e$ and $\hat{p}_o$ leading to l_e and l_o are traced in the whole trees $\hat{s}_e$ and $\hat{s}_o$. Merge these two paths as much as possible by retrieving the needed splitting characters in a batch of size $O(|\hat{s}_e| + |\hat{s}_o|)$. Assume without loss of generality that $u_e \in \hat{p}_e$ is the last merged (even) node that splits an edge in $\hat{p}_o$.

4. Insert a unary node u_o at this edge, with $L(u_o) = L(u_e)$, and thus form the new tree $\hat{s}_o$.

5. Treat the unary node u_o and the node u_e as "anchors" by detecting (if any) their single descending side tree pair (Definition 4.11). If so, we set $\hat{s}_o$ and $\hat{s}_e$ equal to that pair, and we recurse on this pair. Otherwise, l_e and l_o is a pull leaf pair.

The efficiency of this loop is strictly related to the fact, which we will show in the next lemma, that after the first iteration the new size of either $\hat{s}_o$ or $\hat{s}_e$ is the square-root of its original value. Then, in the recursive step, we sample the leaves of the new $\hat{s}_o$ and $\hat{s}_e$ more densely, and thus get at most three recursive levels to complete the computation.

Lemma 4.15 *All the pull leaves in T_e and T_o can be determined in $O(Sort(n))$ I/Os.*

Proof: The correctness of the algorithm follows from the correctness of the retracing step that ensures that we will never depart from the odd/even node path. Hence, although $\hat{s}_e$ and $\hat{s}_o$ shrink, they will always contain the pull leaves we are searching for.

Specifically, the algorithm merges the two paths $\hat{p}_e$ and $\hat{p}_o$ leading to l_e and l_o until the node $u_e \in \hat{p}_e$ is reached such that: it splits an edge in $\hat{p}_o$ but the next character, say c_o, on this edge does not match the corresponding character, say c_e, which labels the edge in $\hat{p}_e$ outgoing from u_e, if any. There are three cases and in all of them either we find the pull leaves, or we shrink one of the side trees, either $\hat{s}_o$ or $\hat{s}_e$, to the square-root of their original sizes.

(1) Case: u_e is a leaf. Here c_e does not exist. Then u_e is actually a pull leaf. The other pull leaf in s_o can be taken as any of the leaves descending from the edge split by u_e.

(2) Case: u_e occurs also in the skeleton tree $\hat{s}'_e$. We can immediately conclude that we also surely stopped at u_e when merging the two skeleton trees $\hat{s}'_e$ and $\hat{s}'_o$. Consequently no edge in $\hat{s}'_e$ outgoing from u_e is labeled by c_o. Therefore, let us denote by e' and e'' the two adjacent edges in $\hat{s}'_e$ outgoing from u_e such that their first labeling characters c', c'' satisfy the relation $c' < c_o < c''$ (possibly one of them is missing). The algorithm selects as new side tree, the subtree descending from the edge of u_e labeled with c_o. If this edge does not exist, the algorithm correctly selects as a pull leaf any leaf descending from u_e (and easily derives the other pull leaf). Otherwise, if this edge does exist then the size of the selected subtree will be $O(\sqrt{|\hat{s}_e|})$. Indeed, its set of leaves will be contained in the set of leaves of $\hat{s}_e$ that lie between the rightmost leaf descending from e' and the leftmost leaf descending from e''. There are $O(\sqrt{|\hat{s}_e|})$ such leaves.

(3) Case: u_e does not occur in the skeleton tree $\hat{s}'_e$. Hence u_e lies on an edge, say ed, of the skeleton tree $\hat{s}'_e$ (which actually represent a sub-path of $\hat{s}_e$). Let us denote by $\mathcal{L}$ the set of $\sqrt{|\hat{s}_e|}$ leaves in $\hat{s}_e$ that lie to the right (resp. left) of the rightmost (resp. leftmost) leaf descending from the edge ed if $c_e < c_o$ (resp. if $c_e > c_o$). The algorithm will select as new side tree the subtree in $\hat{s}_e$ that descends from the edge of u_e labeled with c_o. This subtree will be either empty, and thus the selected pull leaves are correct, or will be formed by leaves contained in $\mathcal{L}$, and thus it will have size $O(\sqrt{|\hat{s}_e|})$.

As far as the I/O-complexity is concerned, we observe that at each iteration of the loop we sample the two side trees $\hat{s}_e$ and $\hat{s}_o$ in $O(Sort(|\hat{s}_e| + |\hat{s}_o|))$ I/Os (Theorem 4.5). Merging the sampled trees and retracing the

two paths leading to the two picked "pseudo-pull" leaves costs $O((Sort(|\hat{s}'_e|+|\hat{s}'_o|)^2)) = O(Sort(|\hat{s}_e|+|\hat{s}_o|)) = O(Sort(|s_e|+|s_o|))$ I/Os (Theorem 4.6). The same I/O-cost is needed to insert the new unary node u_o in $\hat{s}_o$ and identify the new side tree pair descending from u_e and u_o (via the Euler tours of $\hat{s}_e$ and $\hat{s}_o$, Theorem 4.1). Since the loop is executed at most three times, the I/O-cost for finding the true pull leaves in s_e and s_o is $O(Sort(|s_e| + |s_o|))$ I/Os. The global algorithm operates simultaneously on all the side tree pairs, thus its I/O-cost is the one stated in the lemma because the total size of all side trees is actually $O(n)$. ■

4.4.2 From the superset of odd/even nodes to the (over)merged tree T_M

Recall from Section 3.3 that the (over)merged tree T_M is obtained from T_o and T_e by performing a coupled-DFS traversal of the odd and even trees so that two edge-strings are declared to be equal if they *begin with* the same character. Such an on-line merging seems I/O-expensive, hence we exploit the knowledge of anchor-node pairs and pull leaves to determine T_M in an I/O-efficient manner.

Once we have computed for each side tree pair a pair of pull leaves (Lemma 4.15), one below the odd tooth (in T_o) and one below the even tooth (in T_e), then we merge the paths leading to them by declaring two edge-strings to be equal if they *begin with* the same character. This is similar to the retracing step above, but it is now executed on the whole side trees s_e and s_o (plus the ancestors of their roots), and thus it takes $O(Sort(|s_e| + |s_o|))$ I/Os. Next, we *change* the parent pointer of each merged node in order to point to its preceding node in the merged path. This step simulates the merging of the side-tree pairs based on the DFS-coupled traversal.

We are left with the DFS-coupled merging of the part of T_o and T_e that lies above anchor-node pairs. The idea here is to decompose T_o and T_e into paths that are delimited by anchor nodes (but do not contain any of them). In fact, for any pairs of anchor nodes, say u and v in T_o (resp. T_e), we have that $\texttt{lca}(u, v)$ is also an anchor node of T_o (resp. T_e). These paths may possibly touch each other, but only at the first or last (anchor) nodes. Hence we have that: (1) we can pair one path of T_o with one path of T_e according to the delimiting anchor-node pairs; and (2) each pair of paths has the nice property that they can be merged by just looking at the string lengths (i.e. L values) associated with their nodes. In fact, from Definition 4.10 we know that these two paths denote the same string.

¿From these observations it is simple to design a procedure that merges the part of T_o and T_e lying above anchor-node pairs. From Lemma 4.14 we know the anchor-node pairs, and thus we can *re-name uniquely* all of them via a sorting step. We mark all anchor nodes and compute for every node u, which is an ancestor of some anchor node, its closest marked ancestor and descendant, say $anc(u)$ and $desc(u)$ respectively (Corollary 4.2). Next, we form the quadruples $\langle anc(u), desc(u), L(u), u\rangle$ and sort them lexicographically. As a result, nodes that lie on a path to be merged, are adjacent in this sorted list and share the same first two components of the quadruples. Then, we *set* the new parent pointers of these nodes to point toward the node that precedes them in the sorted sequence of quadruples. This step simulates the DFS-merging of the ancestors of anchor nodes, and takes $O(Sort(n))$ I/Os in total.

Finally, we construct the (over)merged tree T_M from this new set of parent pointers in $O(Sort(n))$ I/Os (Theorem 4.1).

Lemma 4.16 *Given the pull leaves and the anchor node pairs in T_o and T_e, the (over)merged tree T_M can be computed in $O(Sort(n))$ I/Os.*

4.4.3 Unmerging T_M to obtain the suffix tree T_S

An odd/even node u in T_M can be either a correct odd/even node, or it can be an (over)merged node that does not occur in the final T_S and thus must be *unmerged*. We can check the status of u as follows. Let l_{2j-1} and l_{2i} be a pair of odd and even leaves descending from u in T_M, such that $u = \texttt{lca}(l_{2i}, l_{2j-1})$. Let us define the pointer $d(u) = \texttt{lca}(l_{2i+1}, l_{2j})$ and the function $\hat{L}(u)$ be equal to the depth of u in the d-tree (Lemma 3.6). We can conclude that:

Property 2 *Node u is properly merged if, and only if, $L(u) = \hat{L}(u)$.*

The alternative is that $\hat{L}(u) < L(u)$, in which case we have merged too far. Computing $\hat{L}(u)$ for all merged nodes u in T_M takes $O(Sort(n))$ I/Os, since we need to answer a batch of $O(n)$ `lca`-queries on T_M's leaves and determine the depth of the nodes in the d-tree (Theorem 4.1).

Subsequently, we use $\hat{L}(u)$ to unmerge the (over)merged nodes in T_M thus completing the construction of T_S. We can either *completely unmerge* a node u or *partially unmerge* it. Let $p(u)$ be the parent of u in the original tree (T_e or T_o); and let $p_M(u)$ be the parent of u in T_M. Completely unmerging a node u means setting $p_M(u) = p(u)$, that is, reseting the parent of u to be its original parent in the unmerged tree. If Property 2 is violated, so that $L(u) > \hat{L}(u)$, then a complete unmerge is not always the correct thing to do. Consider, for example, Figure 3.5. It would be incorrect to completely unmerge node $\texttt{lca}(l_5, l_8)$. Instead we need to introduce a node that is the parent of l_5 and l_8 and set its string length to 3, its depth in the d-tree. This is a partial unmerge. Both types of unmerges are handled by the following procedure.

Define a node u to be a *border* node if $\hat{L}(u) < L(u)$, but $\hat{L}(p_M(u)) = L(p_M(u))$. Notice that the set of border nodes form an anti-chain in T_M. We can easily find all the border nodes in $O(Sort(n))$ I/Os. Now, for each border node u, we perform the following operations. Let T_o^u be the odd tree below u and let T_e^u be the even tree below u in T_M. No node in T_o^u or T_e^u should be merged, so unmerge them all by reseting, for each such node v, its parent pointer $p_M(v) = p(v)$. The remaining question is what node to hang T_o^u and T_e^u off of. But the set of border nodes are exactly those nodes at which we need a partial unmerge. These are handled, for each border node u, as follows:

1. Create a new node u' and make it a child of $p_M(u)$.
2. Set $L(u') = \hat{L}(u)$.
3. Hang T_o^u and T_e^u off of u'. Order the children of u' lexicographically.

Once border nodes u have been flagged, all their descendants in T_o^u and T_e^u can be identified in $O(Sort(n))$ I/Os by means of the Euler Tours $\texttt{ET}(T_o)$ and $\texttt{ET}(T_e)$ (Theorem 4.1). Hence, the partial or complete unmerge step in T_M takes $O(Sort(n))$ I/Os. When this procedure is completed, the tree will be correct because Property 2 will be satisfied for every node.

Lemma 4.17 *Given the (over)merged tree T_M, the final suffix tree T_S can be obtained in $O(Sort(n))$ I/Os.*

4.4.4 Putting it all together

We are therefore ready to state the main result of this section:

Theorem 4.18 *Given an arbitrary string $S \in \Sigma^n$, its suffix tree T_S can be constructed in optimal $O(Sort(n))$ I/Os, $O(n/B)$ space, and $O(n \log n)$ time.*

Proof: Let $C(n)$ be the number of I/Os taken by our algorithm to find T_S. We review the algorithm: (1) Find the odd tree T_o and the even tree T_e, by Lemmas 4.8 and 4.9; (2) Find the anchor pairs, by Lemma 4.14; (3) For each side tree pair, find a pair of pull leaves, by Lemma 4.15; (4) From the anchor nodes and the pull leaves, compute the (over)merged tree T_M, by Lemma 4.16; (5) Unmerge T_M to obtain T_S, by Lemma 4.17.

The first step takes $C(n/2) + O(Sort(n))$ I/Os, and the rest take $O(Sort(n))$ I/Os, thus establishing the theorem. We remark that the time complexity easily follows by observing that: (i) at each I/O operation, our algorithm executed a linear, namely $O(B)$, number of internal operations; (ii) the basic sorting routine employed in many places of our approach takes $O(n \log n)$ time. ■

5 Concluding remarks

Throughout, we have focused on sequential models, be they RAM or DAM models. The recursive odd-even approach lends itself to parallelism in various models as described below.

It is just a matter of an exercise to extend our external-memory algorithm to work *optimally* in a model where D parallel disks are available [VS94], called *PDAM* model. Here, we can use the optimal sorting routine described in [NV95], and derive the optimality from the sorting lower bound $\Omega(\frac{n}{DB}\log_{M/B}\frac{n}{M})$ proved in [AV88].

Theorem 5.1 *Given an arbitrary string $S[1,n]$, its suffix tree T_S can be optimally constructed in the PDAM model requiring $O(\frac{n}{DB}\log_{M/B}\frac{n}{DB})$ I/Os, $O(n/DB)$ space and $O(n\log n)$ time.*

Proof: In the DAM algorithm we used two basic routines, other than the sorting procedure: scanning a set of m items stored contiguously on the disk, and implementing an external-memory stack to perform batches of $O(m)$ pop and push operations. Both of these routines took $O(m/B)$ I/Os. Both can be made to work on the PDAM requiring $O(m/DB)$ I/Os. Consequently, the sorting cost subsumes the I/O-complexity of these two routines. ■

Previous string processing results on the PDAM have all been suboptimal [AFGV97]. By using our result, we trivially get the first known I/O-optimal suffix array [MM93] and String B-tree [FG99] construction algorithms. Furthermore, many string problems have suffix tree construction as their I/O bottleneck [Gus98]. All these now have efficient implementations on the (P)DAM.

The approach here can be extended to get an $O(n)$ work, $O(\log n)$ time, Las Vegas type algorithm for CRCW PRAM [FM96]; previously best known optimal algorithms required $\Omega(\log^2 n)$ time. However, there are many PRAM-specific issues involved in obtaining our result, and they are omitted here.

Our results can be extended to other parallel models of computation as well: see [FFM98] for a statement of the results on the BSP and EREW PRAM models. The details in these cases do not provide new insights, and therefore they are omitted.

Another issue of relevance is the way accounting works in external memory algorithms. It is well known that accessing contiguous pages in the external memory is less expensive than accessing random pages, a phenomenon routinely exploited by programmers in practice. However, DAM models typically do not account for this difference. See [FFM98] for an extension of the standard accounting scheme to differentiate between random page accesses and contiguous page accesses. Our DAM suffix tree construction algorithm matches the sorting bound under this new accounting scheme as well.

Preliminary experimental investigations on the construction of large suffix arrays using some of the techniques presented in this paper have been pursued in [CF00].

References

[ACF90] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 600–609, 1990.

[ACS87] A. Aggarwal, A. Chandra, and M. Snir. Heirarchical memory with block transfer. *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 204–216, 1987.

[AFGV97] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *Proc. of the 29th Ann. ACM Symp. on Theory of Computing*, pages 540–548, 1997.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AV88] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[BFC00] M. Bender and M. Farach-Colton. Least common ancestors revisited. *Latin '00*, 2000.

[CF00] A. Crauser and P. Ferragina. On constructing suffix arrays in external memory. In *Proceedings of the 7th Annual European Symposium on Algorithms*, pages 224–235, 1999. To appear in *Algorithmica*, 2000.

[CGG+95] Y. Chiang, M.T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. *Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[CS85] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, chapter 12, pages 97–107. NATO ASI Series F: Computer and System Sciences, 1985.

[DK95] A. Delcher and S. Kosaraju. Large-scale assembly of dna strings and space-efficient construction of suffix trees. *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*, pages 169–177, 1995.

[DK96] A. Delcher and S. Kosaraju. Large-scale assembly of dna strings and space-efficient construction of suffix trees (corrections). *Proc. of the 28th Ann. ACM Symp. on Theory of Computing*, page 659, 1996.

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. *Proc. of the 38th IEEE Annual Symp. on Foundation of Computer Science*, pages 137–143, 1997.

[FFM98] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. of the 39th IEEE Annual Symp. on Foundation of Computer Science*, pages 174–183, 1998.

[FG99] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[FM96] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. *Proc. of 23rd International Colloquium on Automata Languages and Programming*, pages 550–561, 1996.

[Gal85] Z. Galil. Open problems in stringology. In Z. Galil A. Apostolico, editor, *Combinatorial Algorithms on Words*, volume 12, pages 1–8. NATO ASI Series F, 1985.

[Gus98] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Addison Wesley, 1998.

[Har94] R. Hariharan. On building suffix trees. Personal Communication, 1994.

[Har97] R. Hariharan. Optimal parallel suffix tree construction. *Journal of Computer and System Sciences*, 55(1):44–69, 1997.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[Kos94] S. R. Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees. *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, pages 310–316, 1994.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[NV95] M. H. Nodine and J. S. Vitter. Greed sort: optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, 1995.

[RW94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[Shr97] E. Shriver. *Performance modelling for realistic storage devices.* PhD thesis, Department of Computer Science, New York University, New York, NY, 1997.

[SV94] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, pages 300–309, 1994.

[Tho98] M. Thorup. Faster deterministic sorting and priority queues in linear space. In *Proc. of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.

[Vis94a] U. Vishkin. Consistent naming for suffix tree construction. DIMACS talk on joint work with S. C. Sahinalp, 1994.

[Vis94b] U. Vishkin. On building suffix trees. Personal Communication, 1994.

[Vit] Jeffrey S. Vitter. External memory algorithms. Invited Tutorial in the *17th ACM Symposium on Principles of Database Systems (PODS '98)*, 1998. Also invited paper in *Proceedings of the 6th Annual European Symposium on Algorithms (ESA '98)*, Lecture Notes in Computer Science 1461, Springer-Verlag, 1998.

[VS94] J. S. Vitter and E. Shriver. Algorithms for parallel memory: Two-level memories. *Algorithmica*, 12:110–147, 1994.

[Wei73] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.