Faster Dynamic Compact Tries with Applications to Sparse Suffix Tree Construction and Other String Problems

Takuya Takagi[1], Takashi Uemura[2], Shunsuke Inenaga[3], Kunihiko Sadakane[4], and Hiroki Arimura[1]

[1] IST & School of Engineering, Hokkaido University, Japan
tkg@ist.hokudai.ac.jp, arim@ist.hokudai.ac.jp
[2] Chowa Giken Corporation, Japan
[3] Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp
[4] National Institute of Informatics, Japan
sada@nii.ac.jp

Abstract. The dynamic compact trie is a fundamental data structure for a wide range of string processing problems. Jansson, Sadakane, and Sung (LNCS 4855, pp.424-435, FSTTCS 2007) presented the dynamic uncompacted trie data structure of n nodes in $O(n \log \sigma)$ space supporting pattern matching in $O((|P|/\alpha) f(n))$ time and insert/delete operations in $O(f(n))$ time, where $f(n) = ((\log\log n)^2/\log\log\log n)$ is the present best time bound for dynamic predecessor dictionary. Besides its advantage, it is not applicable to linear time suffix tree construction because it has quadratic space complexity there. By extending their work, we present the dynamic *compact* trie data structure that can store a set of k strings of a single reference string of length n in $O(n \log \sigma + k \log n)$ bits of space even if the total length of edge labels is quadratic in n, still supporting pattern matching and insert/delete operations in the same time complexity as Jansson *et al.*'s dynamic trie. As application, we show that our data structure can be used to solve online construction of the sparse suffix tree (SST) in $O((n/\alpha)(f(n) + \log \sigma))$ time and $O(n \log \sigma)$ bits for evenly taken space $r = \Theta(\log_\sigma n)$ with more general bounds.

1 Introduction

The *dynamic compact trie* [6, 16] for a set of k strings over alphabet Σ is a fundamental data structure in string processing, where its binary version is called a *Patricia trie* [14]. One of its most important properties seems that if all label strings are take from a single refenrence string of length n, then the compact trie occupies at most $O(n \log \sigma + k \log n)$ bits of space [6]. For this virture, it plays an essential role in a number of string problems such as dynamic dictionary matching [7], suffix tree [16], sparse suffix tree [12], succinct index [13], and external string index [5].

1.1 Main results

In this paper, we show how to accelerate operations of a compact trie on Word RAM using bit-parallelism and dynamic predecessor data structures still keeping the linear space. Although Jansson, Sadakane, and Sung [9] proposed the dynamic trie data structure using $O(n \log \sigma)$ bits of space allowing sub-linear time pattern matching and insert/delete operations as well as navigational operations, it is still not sufficient for applications such as online suffix tree construction.

Problems that we encounter: A basic idea is to use the *packed string matching* approach [3]. Since each letter is encoded with $\log \sigma$ bits and we can read w bits in constant time on Word RAM, we expect to process $\alpha = \log_\sigma n = \lceil w/\log \sigma \rceil$ consecutive letters in one time step. However, we encounter a difficulty caused by branching during traversal of a compact trie. Actually, in the worst case, prefix search with a pattern of length P requires $O(\lceil P/\alpha \rceil + d_P \log \sigma)$ time, where $d_p = O(K)$ is the number of branching nodes to visit. Using this modification, for example, we only obtain a construction algorithm for sparse suffix trees [12, 8, 15] of a text of K index points and N letters that runs in quadratic time in $K = O(N)$.

Our proposal: To solve this problem, we devise a novel speed up technique by using bit-parallelism and fast dictionary lookup, respectively, for processing long non-branching paths and dense branching subtrees. Based on this, we propose the *fast compact trie* on $\mathcal{D}$, that is a compact trie augmented with any dynamic predecessor dictionary $\mathcal{D}$ for w-bit integers. Suppose that $\mathcal{D}$ stores N w-bit integers in $s(N)$ bits by supporting predecessor and insert operations in $f(w, N)$ time, where s satisfies the inequality $s(N_1) + s(N_2) \leq s(N_1 + N_2)$ (*). Then, we have:

Theorem 1 (main result). *The proposed dynamic compact trie data structure stores a set S of K strings of total size N letters over Σ in $O(N \log \sigma + K \log N + s(K))$ bits, with supporting general prefix search in $O(\lceil (|P|/\alpha) \rceil f(N))$ time, and insert in $O(\lceil |P|/\alpha \rceil f(N) + \log \sigma)$ time, as well as traversal operations in $O(\log \sigma)$ time all in the worst case, where P is a pattern string, and $\alpha = \log_\sigma n$.*

The above result accelerates prefix search without slowing down other trie operations. If we employ the dynamic data structure for $O(\log n)$-bit integers by Beame and Fich [1], which support $f(n) = O(((\log \log n)^2/\log \log \log n))$ predecessor, successor, insertion, and deletion operations as auxiliary structure, then we have the following results.

Theorem 2. *The proposed dynamic compact trie data structure stores a set S of mutually distinct variable-length strings with total size N letters in $O(N \log \sigma)$ bits supporting pattern matching operation in $O(\lceil (|P|/\alpha) \rceil ((\log \log N)^2/\log \log \log N))$ time and in insert/delete operation in $O(\lceil (|P|/\alpha) \rceil ((\log \log N)^2/\log \log \log N) + \log \sigma)$ time both in the worst case, where $\alpha = \log_\sigma N$.*

Applications: As an application of our speed up technique for compact tries, we have the following results. A sparse suffix tree (SST) [12] is a compact trie for a subset of K suffixes of an input text of length N. It has been open that whether general SSTs can be constructed in online $O(N \log \sigma)$ time using $O(N \log \sigma + K \log N)$ bits of space [12]. Inenaga and Takeda [8] showed that it is the case for the SSTs over word alphabets. Recently, Uemura and Arimura [15] extended their results for SSTs over any regular[5] prefix-code $\Delta \subseteq \Sigma^+$ of total size $\delta = ||\Delta||$ letters.

Theorem 3 (faster sparse suffix tree construction). *For any finite prefix code Δ of total size δ, the SST for an encoded text of K codewords and N letters can be constructed online in $O((\lceil \frac{N}{\alpha} \rceil + K)((\log\log N)^2 / \log\log\log N))$ time using $O(N \log \sigma + (K + \delta) \log N)$ bits, where $\alpha = \log_\sigma N$.*

2 Preliminaries

We give definitions and notations necessary to the later sections. For concepts not found here, please consult appropriate textbooks (e.g., [4, 6]).

Basic definitions:word Let Σ be an alphabet of size σ, and n be a natural input size. In this paper, the base of the logarithm $\log n$ is two otherwise stated. For any integer $i \leq j$, the notation $[i, j]$ denotes the interval $\{i, i+1, \dots, j\}$. We denote the empty string by ε. Let $X = X[1] \cdots X[n]$ be a string with length $n = |X|$, where $X[i] \in \Sigma$. For any $0 \leq i \leq j \leq |X| - 1$, we denote by $X[i, j]$ the substring $X[i]X[i+1] \cdots X[j]$. For strings $X, Y \in \Sigma^*$, we denote by $lcp(X, Y)$ the length of the longest common prefix of X and Y counted in letters. We assume the Word RAM with $w = \Theta(\log n)$-bits words supporting standard operations, and that each letter is encoded with $\log \sigma$ bits, and w is a multiple of $\log \sigma$. Let $\alpha = w / \log \sigma$ be the constant called *speed up factor*. Since we can read w bits in constant time, we can read or process α consecutive letters in one time step.

Dynamic compact tries: Let Σ be an alphabet with size σ, and let $S = \{S_1, \dots, S_K\} \subseteq \Sigma^*$ be a set of K mutually distinct strings with total size $n = ||S|| = \sum_i |S_i|$ letters on Σ. The *dynamic compact trie* $\mathcal{T}_S$ for S consists of a path compressed trie $\mathcal{T}_S$ for all strings of S, and the reference string $T = S_1 \cdots S_K$ with total length $n \log \sigma$ bits. In more general setting, we can use as the reference string any single string T that contains each string of S as substring, which is convenient in some applications (e.g. [7, 16]). Precisely, the path compressed trie $\mathcal{T}_S$ is obtained from an ordinary trie for S by compressing non-branching paths so that every internal node has at least two children and each edge has a substring of S as its label, reducing the tree structure in at most $2K - 1$ nodes in $O(K \log n)$ bits. Further saving of the space is obtained by replacing the edge labels with pairs of pointers to the reference string T.

We introduce some data types for the compact trie. Any substring s of T is represented by a pair of pointers $\langle k, j \rangle$, $k \leq j \leq |T|$, such that $T[k, j] = s$. A *locus* v in a compact trie, which corresponds to a node in the associated uncompacted trie, is designated by the unique string $s = str(v) \in \Sigma^*$ that the path from the root to v, $path(v)$, spells out. We say that v is the *locus* of s and denoted by

[5] A prefix code is *regular* if it is accepted by a finite automaton.

$v = [s]$. A node v is said to be *real* if it is a branching node, and *virtual* otherwise. Each real node p has the fields $Parent(v)$ and $Lab(v)$, the pointer to its parent and the string label of its incoming edge of v. For any letter a in Σ, $child(v, a)$ denotes the a-child of node v. The *node depth* $nd(v)$ and *letter depth* $d(v)$ of a node v are the numbers of real nodes and letters on $path(v)$, respectively.

A *reference* of a node v, or a *pointer* to v, is any triple $\phi = (p, \langle k, j\rangle)$ of a real node p, called the *origin*, and a pair of integers $\langle k, j\rangle$, $k \le j + 1$, representing the substring $s = T[k, j]$, called the *offset string*, such that v is reachable from p downward by following s in the standard way [16]. A reference $\phi = (p, \langle k, j\rangle)$ is of *canonical* if the $s = T[k, j]$ is shortest among all references to ϕ. For a canonical pointer ϕ of a real node, p is real and $T[k, j] = \varepsilon$. In what follows, we often identify a node to its reference. For a pattern string P, the *LCP node* for P is the deepest descendant q of a given node v that the label string of the path from v to q is a prefix of P.

After above compression, the compact trie can store all strings of S in $s = O(n \log\sigma + K \log n)$ bits of space supporting pattern matching, insert, and delete operations, traversal operations, as well as their time complexities:

- $LCP(u, P)$: Returns a reference ϕ to the *longest common prefix node* (*LCP node*, for short), that is, the locus of the longest prefix of a pattern string P contained in S below the node u in $O(|P| \log\sigma)$ time. This query is also known as *prefix search query*.
- $Insert(u, P)$: Insert a new leaf v below the node u such that the edge between u and the leaf is labeled by the string P. in $O(|P| \log\sigma)$ time if it is not a member of the collection.
- $Delete(u)$: Delete the leaf u and the edge between u and its parent in $O(|P| \log\sigma)$ time, where P is the label of the edge.
- $Root() = Root(\mathcal{T})$: Return the root of the trie.
- $Parent(u)$: Return a reference ϕ to the parent of a given node u in $O(\log\sigma)$ time.
- $Child(u, a)$: Return a reference to the a-child of the node u in $O(\log\sigma)$ time.

In the above definition, we assume that $\mathcal{T}$ is understood, u, v, and ϕ are references to a node in the trie, P is a variable-length string of length $O(2^w)$, called a pattern string, and a is a letter in Σ. We also use the balanced binary tree encoding of branching.

Dynamic predecessor data structure: A *dynamic predecessor data structure* (e.g., [1, 2, 17]) is a data structure $\mathcal{D}$ for storing a set S of $O(\log n)$-bit integers in $s(K)$ space supporting predecessor, successor, insert, and delete operations in $f(n)$ time. In the worst case, the fastest structure seems the data structure of [1] with $s(K) = K \log n$ and $f(n) = O(((\log\log n)^2/\log\log\log n))$. On average, the fastest one will be the dynamic z-fast trie [2] with $s(K) = K \log n$ and $f(n) = O(1)$.

3 The Proposed Data Structure

3.1 Speeding up prefix search: The small trie case

In this subsection, we present the data structure in the *small trie case*, that is, the small dynamic compacted trie with height exactly $w = O(\log n)$ bits, or α letters, which stores a set $S = \{\, X_i \mid i = 1, \dots, K \,\}$ of K fixed-length strings of length exactly α letters. Assume that S is prefix-free, i.e., no member is a proper prefix of other. We refer to such a small trie of letter height at most α as a *micro trie*. The dynamic data structure in the small trie case, denoted by $\mathcal{S}$, consists of the following components.

- A small dynamic compacted trie $\mathcal{T}$ of height exactly w bits (i.e., α letters) for storing the set S, where $\mathcal{N}$ is the set of internal vertices, $root(\mathcal{T}) \in \mathcal{N}$ is the root, and $\mathcal{L} = \{\ell_1, \dots, \ell_K\}$ is the set of K leaves which correspond to elements $X_1, \dots, X_K$ of S. The children of each node v are sorted by the first letter of their incoming edge labels. Every real node of $\mathcal{T}$ has a pointer to the component $\mathcal{D}$ below. All leaves are assumed to be already materialized as real nodes if they are not.
- A dynamic predecessor data structure $\mathcal{D}$ for storing K keys in the lexicographic order $X_1, \le \cdots \le X_K$ in $s(K)$ space supporting predecessor (*Pred*), successor (*Succ*), insert (*Insert*), and delete (*Delete*) operations in $f(n)$ time. As returned value, the structure returns the index i instead of the member X_i itself, but sometimes we identify X_i with i. For simplicity, we assume the imaginary leaf indices 0 and $K + 1$ such that $Pred(1) = 0$ and $Succ(K) = K + 1$.
- An array $\mathsf{LCA} : [1, K] \to \mathcal{N}$ for associating nodes in $\mathcal{N}$ to leaf indices $[1, K - 1]$, where the entry $\mathsf{LCA}(i)$ $(i \in [1, K-1])$ stores the pointer to $\mathsf{LCA}(\ell_i, \ell_{i+1})$, the lowest common ancestor of ℓ_i and ℓ_{i+1} in the micro trie $\mathcal{T}$. For imaginary leaf indices 0 and $K + 1$, we assume that $LCA(0) = LCA(K) = root(\mathcal{T})$.

Furthermore, we may identify each key X_i with the corresponding leaf ℓ_i for $i \in [1, K]$. Throughout this section, we assume that a pattern string $X \in \Sigma^\alpha$ in the small trie case is given as the substring of the reference string T by $X = T[k, j]$ for some $k \in [1, N]$. Moreover, we use the relative letter depth $d_*(u) = d(u) - d(root(\mathcal{T}))$ instead of the absolute letter depth $d(u)$ for any node u in $\mathcal{T}$. Recall that the operation $LCP(v, X)$ returns the reference ϕ to the locus of the longest common prefix of pattern string X from a node v. Then, we have the following lemma.

Theorem 4 (LCP operation in the small trie case). *In the dynamic compact trie data structure for strings with length α letters (or $w = O(\log n)$-bits), we can implement $LCP(v, X)$ operation in $O(f(n))$ time, where n is the current length of the reference string T, $\alpha = \log_\sigma n$, v is a reference to a node in the micro trie, and X is a pattern string of length at most α letters, packed in a w-bit word.*

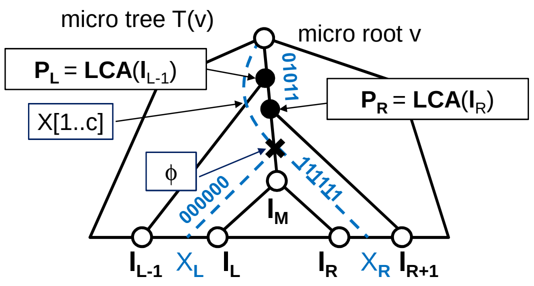

Fig. 1. Small trie case: Fast LCP queries using the dynamic predecessor structure for strings of $O(\log_\sigma n)$ bits

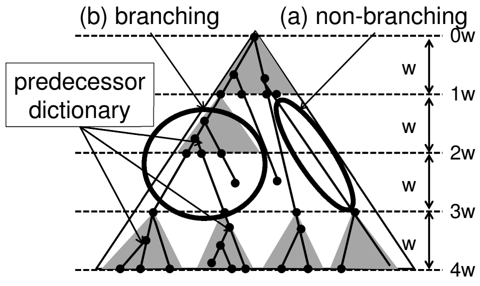

Fig. 2. Large trie case: Micro tree decomposition of the dynamic compact trie structure for strings of $2^{O(w)}$ bits.

To handle keys X_i in S, as well as pattern X, having length less than α, we make padding to X_i so that it has length exactly α, and explicitly store its original length $|X_i|$ at leaf ℓ_i, though the detail is omitted.

We next consider the the *semi-static* compact trie data structure, in which the reference string $T = T[1, n] \in \Sigma^*$ is is fixed first, or the right-end letter $c \in \Sigma$ is only appended to or deleted from the end of T in FILO manner. In this case, the space for strings is bounded by the length of T rather than the total length of the edge labels as in the case of online suffix tree construction.

Lemma 1 (semi-dynamic case). *In the semi-dynamic data structure in the small trie case, we can implement* $Insert(v, X)$ *and* $Delete(v, X)$ *operations in* $O(f(n) + \log\sigma)$ *time maintaining the data structure in* $O(n \log\sigma + K \log n)$ *bits of space, where* n *is the current length of the reference string* T*,* v *is a reference to a node in the micro trie, and* X *is a pattern string of length at most* α *letters, packed in a* w*-bit word. We assume that* $d_*(v) + |X| \le \alpha$ *so that the height of the micro trie is kept within* α *letters.*

Finally, we consider the full dynamic case in which we can arbitrary make insert and delete operation for a pattern string at leaf position, where the additional factor $O(K \log\sigma)$ is added to the space complexity for storing K strings.

Lemma 2 (full-dynamic case). *In the full-dynamic data structure in the small trie case, we can implement* $Insert(v, X)$ *and* $Delete(v, X)$ *operations in* $O(f(n) + \log\sigma)$ *time in* $s = n \log\sigma + O(K \log\sigma + K \log n)$ *bits of space maintaining the set* S *of* K *strings on* Σ *with total length* n *letters, where* v *is a reference to a node in the micro trie, and* X *is a pattern string of length at most* α *letters, packed in a* w*-bit word.*

The maintenance of a set of edge labels in the full-dynamic data structure does not depend on the assumption of the small trie case, and thus, it can be directly applied to the large trie case in the next subsection as well.

3.2 Speeding up prefix search: The large trie case

We present the dynamic compact trie with no height restriction for storing variable-length strings of length $O(2^w)$. The data structure stores all strings in $O(n \log \sigma)$ bits supporting LCP operation in $O(\lceil (n/\alpha) \rceil f(n))$ time and insert,/delete operation in $O(\lceil (n/\alpha) \rceil f(n) + \log \sigma)$ time.

Micro trie decomposition: In our data structure, we partition a general trie $\mathcal{T}$ of height $h \geq 0$ into small tries of height at most $\alpha = O(\log_\sigma n)$ letters. We split the nodes of $\mathcal{T}$ into $\lceil d/\alpha \rceil$ levels as follows. A node $u \in \mathcal{T}$, either virtual or real, is a *boundary node* if the depth $d(u)$ of u from $root(\mathcal{T})$ is of multiple of α, i.e., $0\alpha, 1\alpha, 2\alpha, \ldots, \lceil h/\alpha \rceil$. Let BN be the set of all boundary nodes of $\mathcal{T}$. We say that a node v has level i if $d(u) \in [i\alpha, (i+1)\alpha - 1]$ holds for $i \in [0, h]$. For each $u \in BN$, We define the subtree $\mathcal{T}_u$ of $\mathcal{T}$ with height α letters as the subtree rooted at u that consists of all descendants v of u whose depth satisfies $d(v) \in [i\alpha, (i+1)\alpha]$. Clearly, all internal nodes have level i, and the leaves have level $i+1$.

We consider two cases for each $u \in BN$. If the subtree $\mathcal{T}_u$ consists of a single path, we are done. If it contains at least one branching node, say v, we insert real nodes at the positions of the root and leaves of $\mathcal{T}_u$, that is, the boundary of level i and $i+1$ if not. For each boundary node $u \in BN$, we have a dynamic compact trie data structure $\mathcal{S}_u$ in the small case (Sec. 3.1), which is associated to the subtree $\mathcal{T}_u$ at u as the underlying micro trie of $\mathcal{S}_u$, and each edge e outgoing from any real node $v \in \mathcal{T}_u$ must have a pointer to $\mathcal{D}$. The above transformation adds $O(K)$ extra real nodes and $O(K \log n)$ space for dynamic predecessor data structures to the dynamic trie.

The proof of the next lemma uses a somewhat surprising fact that if S is a prefix-free set of strings with total length of n letters, then it can contain at most $K \leq O((n/\log_\sigma n))$ mutually distinct strings asymptotically. For the completeness, we include the proof of this claim below.

Lemma 3 (the space complexity in the large case). *In the large trie case, the dynamic data structure can store a set S of K strings on Σ with total length n letters in $O(n \log \sigma)$ bits of space when the strings of S are mutually distinct, and prefix-free.*

Proof. Below, we show the upperbound $K \leq O((n \log \sigma / \log n))$ of the number K of the mutually distinct strings in S. Let $\beta = \frac{1}{2} \log_\sigma n = (\log n / 2 \log \sigma)$ be a number. Classify the strings of S to two classes as follows. If the first set S_1 consists of all strings shorter than β, then there can be at most $K_1 = |S_1|$ must be $K_1 \leq \sigma^\beta = \sqrt{n}$ mutually distinct short strings in combination (Consider the complete σ-ary tree with height β, and locate all short strings in it). If the second set S_2 consists of all strings longer or equal to β, then the number $K_2 = |S_1|$ must be $K_2 \leq (n/\beta) = (2n \log \sigma / \log n)$ letters since their length must sum up to n. Overall, the total number $K = K_1 + K_2 = |S|$ of strings satisfies the inequality $K \leq (2n \log \sigma / \log n) + \sqrt{n}$. This approaches $K = O((n \log \sigma / \log n))$ as n goes to ∞. Since the reference string occupies $O(n \log \sigma)$ bits and each node requires $O(\log n)$ bits, we have $O(n \log \sigma)$ bits of space overall. $\square$

LCP operation: Given a pattern X of length $P = O(2^w)$, we implement general prefix search from any node v in $\mathcal{T}$ to run in $O(\lceil P/\alpha \rceil f(w, N))$ time as follows. Then, we have the following lemma.

Lemma 4 (the dynamic compact trie in the large trie case). *Given the dynamic data structure for a set of k strings with total length n, we can implement LCP operation in $O(\lceil (|P|/\alpha) \rceil f(n))$ time, where P is a pattern string of arbitrary length $O(2^w)$.*

Proof. Starting from v, if some dynamic data structure $\mathcal{D}$ is associated, then $\mathcal{T}_v$ has a branching node, and thus, we make branching by making LCP operation in $O(f(n))$ time by Lemma 4. Otherwise, v has only one path whose length is larger than α. Then, we repeat the LCP operation on the path to meet a disagreement in $O(\log \log n)$ time at every α letters on the path using XOR and MSB as in the proof of Theorem 4. This completes the proof. □

The insert operation: Initially, the trie $\mathcal{T}_S$ consists of the root only. Then, we can insert the initial string to $\mathcal{T}_S$ as usual.

Lemma 5 (the full-dynamic compact trie in the large case). *We can dynamically maintain the dynamic data structure for storing a set S of k strings with total length n letters using space $s = n \log \sigma + O(K \log \sigma + K \log n)$ bits of space supporting Insert and Delete operations in $O(\lceil (|P|/\alpha) \rceil f(n) + \log \sigma)$, where P is a pattern string of length $O(2^w)$.*

Proof. From Lemma 4, we make LCP operation at v in the claimed time complexity, and if it is not contained yet, we add a new leaf with the edge label P, and makes appropriate maintenance. Details are omitted. □

Analysis: The time complexity is clear from the discussion in the previous subsections. On space complexity, if each micro trie $\mathcal{S}_v$ contains at most K_v nodes including its leaves, the dictionary $\mathcal{D}_v$ uses $O(s(K_i))$ bits. Moreover, every real, branching node belongs to at most two micro trie regions. If the space $s(N)$ satisfies the inequality (*) in Sec. 1.1, the total space is $S = \sum_i O(s(K_i)) = O(s(n))$, this proves Theorem 1. Since it is the case for the space $s(N) = O(N \log N)$ of the predecessor data structure of [1], Corollary 1 follows.

4 Application to Sparse Suffix Tree Construction and Other String Problems

We apply our dynamic compact trie structure for variable-length strings of $O(2^w)$ bits to solve the following problems efficiently in space and time on RAM. Let n be a natural input size, $\alpha = \log_\sigma n$, and $f(n) = ((\log \log n)^2 / \log \log \log n)$.

Corollary 1. *We can solve the following problems by the dynamic compact trie data structure. Let Σ be an alphabet of size σ, and T be a text of length n.*

(1) Online sparse suffix tree (SST) construction [8, 12, 15] for a text of length n in $t = O((n/\alpha)f(n) + kf(n) + k\log\sigma)$ time using $s = O(n\log\sigma + k\log n + \delta\log n)$ bits of space, where k is the number of indices, T is a string over a prefix code $\Delta \subseteq \Sigma^+$, and δ is the total length of the codes.
(2) Succinct substring indexes using sparse suffix trees [13] in $t = O(|P|f(n)\log\sigma + n\log\sigma)$ time and $s = O(n\log\sigma)$ bits of space and $p = O((n/\alpha)f(n) + k\log\sigma)$ preprocessing, where P is a pattern string.
(3) Succinct dynamic dictionary matching [7]. $t = O(n(\log\sigma)f(n))$ time and $s = O(m\log\sigma)$ bits of space and $O((m/\alpha)(f(n) + \log\sigma))$ preprocessing, where m is the total length of strings in a set $S \subseteq \Sigma^*$.

In the above corollary, (1) this is $O(\alpha)/f(n)$ times speed up of [15]. In (2), the preprocessing is $O(\alpha/f(n))$ times speed up of the original by Kolpakov *et al.* [13], while the search is not. In (3), the time of the first algorithm of Hon et al. [7] is $O(n\log n + occ)$, while our algorithm runs in $t = O(n(\log\sigma)f(n))$ time. The space is same $O(n\log\sigma)$. Since $(\log\sigma)f(n) = O(\log n)$, our solution gives a slight improvement. Also, we can apply the succinct dynamic trie data structure of Jansson, Sadakane, and Sung [9] directly to (1) the sparse suffix tree problem on an input string of length n. However, it may result quadratic space complexity.

We give a brief sketch of the algorithms for the above problems. Below, we assume that the reader is familiar with Ukkonen's online suffix tree construction algorithm [16]. Given a compact trie $\mathcal{T}$ for a set S of strings and a text $T = T[1, n]$ of length n, the *longest common prefix* (LCP) of P at $T[1, i]$ is the longest suffix $s = T[j, i]$ of $T[1, i]$ that is contained by some pattern $P \in S$. We introduce the following convention. Assume that the special substring $T[j, i]$, where $j \geq i + 1$, represents the *negative string* and it matches any string of length $\ell = j - i$.

For every *stage* $1 \leq i \leq n$, the interval $A = [j, i]$ is called an *active interval*, and the locus $\phi \in \mathcal{T}$ of A in $\mathcal{T}$ is called the *active point* at stage i. Then, the *online LCP computation* is the problem of successively maintaining the active interval $A = [j, i]$ at $T[1, i]$ for every $i = 1, \ldots, n$. Let $r \geq 1$ be any positive integer, and $\mathcal{T}_r$ be a *even-spaced sparse suffix tree* [12], called the r-sparse suffix tree, for the set S, i.e., the compact trie for all r-suffixes of strings of S. Each string of length r delimited by index points is called an *r-segment*, or a segment, of T. Generally, if all segments are prefix-free, we can build the sparse suffix tree online in linear time [15].

Lemma 6. *The online LCP computation problem for the r-sparse suffix tree for a set S of strings can be solved in $O((n/\alpha + k)f(n) + k\log\sigma)$ time using $O(n\log\sigma + k\log n)$ bits of space, where for every $i = 1, \ldots, n$, the algorithm correctly maintains the active point ϕ for $A = [j, i]$.*

Proof. (1) The algorithm $\mathcal{A}$ is a modification of Ukkonen's online suffix tree construction algorithm [16], also resembles one in [15]. $\mathcal{A}$ traverses $\mathcal{T}$ by scanning the text $T = T[1, n]$, while it keeps track of the substring $T[j, i]$ on T, called the *active interval* during the computation. Let $A = [j, i]$ be the active interval and ϕ be the active point. $\mathcal{A}$ starts from the root of $\mathcal{T}$.

(2) At every stage $i = 1, \ldots, n$, $\mathcal{A}$ enters the phase for exntension of type III, and tries to extend suffix ended at the presentϕ for $T[j, i-1]$ by letter $c = T[i]$ to include the suffix $T[j, i] = T[j, i-1]c$ in $\mathcal{T}$. $\mathcal{A}$ succeeded to extend the branch by c, then it proceeds from $T[j, i-1]$ to the locus for $T[j, i]$ (called extension of type III in [16]), and repeat this process until it encountered the disagreement at some node, and then enters phase for extension of type II. (3) At the disagreement node ϕ, $\mathcal{A}$ inserts a new leaf as a child of ϕ with label c. Then, it makes the suffix transition from ϕ with path string $y = cx$, where c is a segment on T and x any string, to the unique node with path string x obtained from y by removing the starting segment c (extension of type II). This process is repeated until extension of type III occurs at some node ϕ.

In the case of r-sparse suffix tree, we must take care when we go out side of $\mathcal{T}$. This is possible; If the active string $A = [j, i]$ is a proper substring of a r-segment, then removing r-length prefix yields $A' = [j', i] = [i, j']$ with $j' = j - r > i$. In this case, the active interval $T[j', i]$ represents a *negative* string. Therefore, the correct rule is to skip all input letters by incrementing i one by one until $j \leq i$ happends. (In [15] and [8] used a special finite automaton for the set of possible segments.). The correctness of the above procedure is not hard to see. Since any negative $T[j, i]$ (with $j \geq i + 1$) matches any string of the same length, we can obtain the LCP for $T[1, i]$ from the LCP $T[j, i]$ for $T[1, i-1]$ by increment i by one and continue to decrement j to find the LCP. (1) For the time complexity, the proof proceeds as [16] and [15]. The point is that Sicne extension of type II implies $j \leftarrow j + r$, the number of suffix link transition is bounded by $O(k) = O(n/r)$. Therefore, the total time for type III is proven to be $O(k \log \sigma)$. (2) Since one type III extension implies $i \leftarrow i + 1$, the number of extension of type III is $O(n)$. Thus, the total time for type III extensions is proven to be $O((n/\alpha + k)f(n))$ time. By applying the argument similar to [16], we have the time complexity. □

Proof for Corollary 1 1. We first consider the online sparse suffix tree (SST) construction. The algorithm of [15] is the modification of the algorithm $\mathcal{A}$ of Lemma 6 for online LCP computation except that during extension of type II, the algorithm extend the suffix $T[j, i]$ by inserting the current letter $c = T[i]$ at the active point, and the rest is same. The correctness easily follows from [15], and the proof for the time complexity of $\mathcal{A}$ appplies to the modified one. This show the claim.

2. Succinct substring indexes using sparse suffix trees. Let $r = \log_\sigma n = \alpha$ be the segment length. Construction is almost same as 4 and from [13] except that we use the ordinary r-sparse suffix tree. Thus, we have to separately make $r = \alpha$ left-searches. In preprocessing, we construct the ordinary r-sparse suffix tree for text T as well as the reverse suffix and the range query structure. In this case, the space and the preprocessing immediately follows from (1) of the lemma. For search, we make $r = \alpha$ *LCP* operations for $P[i, |P|]$ for every $i \in [1, \alpha]$. Thus, the time follows by applying Lemma 4 and from [13].

3. Succinct dynamic dictionary matching. Our algorithm is similar to [7], but slightly defferent from it since we use the online LCP computation algorithm of

Lemma 6 to avoid the use of complicated marked ancestor computation. Given a set S of patterns, we build the r-SST $\mathcal{T}$ for S for $r = \log_\sigma n = \alpha$. Similarly to the above (2) [13], we make LCP operations r times for $P_h = P[h, |P|]$ for every $i \in [1, \alpha]$. By the definition of the online LCP and active interval $T[j, i]$ for $\mathcal{T}$, a pattern $P \in S$ appears at end position i iff a subpattern P_h appears at end position i which is an r-index point and there is a right match at i also. We see that the P_h's occurrence iff the active point ϕ for $T[j, i]$ is a proper active interval in the sense of r-suffixes, and visits the leaf that is the locus of P_h. From this argument, we can directly use the traversal by the algorithm $\mathcal{A}$ of 6. This completes the proof. □

5 Experimental Results

Experimental settings: As datasets, we used Japanese text files in UTF-8 encoding, whose statistics are shown in Table 1. *Japanese Wikititles* dataset consisted of titles of Japanese Wikipedia pages from the Wikipedia site[6], which we prepared the sorted and unsorted versions. *Japanese Newspapers* dataset consisted of Japanese newspaper articles from KyotoCorpus4.0 [7].

We implemented the accelerated version of the sparse suffix tree construction algorithm [15] using the proposed dynamic compact trie data structure in C/C++ as described in Sec. 3.2 and Sec. 4. Specifically, we implemented the following versions of sparse suffix tree (SST) construction algorithms:

- ST: Ukkonen's suffix tree (ST) construction algorithm [16].
- SST: The SST construction algorithm by Uemura *et al* [15].
- SST_L and SST_LH: Faster SST construction algorithms based on dynamic compacted trie in Sec. 4 that uses bit-parallel LCP computation (L) only, and both LCP and the hash table [8] (LH), where the table is attached to the root and stores labels of length three.

In the above implementations, we use the speed up factor $\alpha = 4$ since our machine is 32-bit PC. We compiled the programs with g++ using -O2 option. ran the expriments on a 1.7 GHz Intel Core i5 processor with 4 GB of memory, running Mac OS X 10.7.5. We measured the total construction time of the suffix trees by each algorithm on a given input string, and also that of the generalized suffix trees.

Experimental results: In Table. 2, we show our experimental results, which showed the total number of trees for ST (suffix trees) and sparse suffix trees, and the construction time of the ordinary and sparse suffix trees by the above algorithms. From the result, we observed that the algorithm SST_L with LCP was 36% faster than SST, and moreover 231% faster than ST, while SST_LH

[6] `http://dumps.wikimedia.org/jawiki/`.

[7] `http://nlp.ist.i.kyoto-u.ac.jp/nl-resource/corpus/`

[8] In the experiment, we actually used the string container `map<string>` in C++/STL library, which is implemented with the balanced binary tree [4].

Table 1. The data sets are used this experiments.

Data set	Code	Total size (Byte)	Number of strings	Ave. string length (Byte)	Ave. code length (Byte)
Japanese Wikititles	UTF-8	30,414,297	1,372,988	22.1	2.44
Japanese Newspaper	UTF-8	30,400,000	102,674	296.1	2.97

Table 2. The results our experiments.

Data set	Tree size (nodes)		Construction time(Sec.)			
	ST	SST	ST	SST	SST_L	SST_LH
Japanese Wikititles (unsorted)	45,526,142	18,409,345	72.52	29.76	**21.91**	23.35
Japanese Wikititles (sorted)	46,663,980	18,809,217	72.62	24.93	**13.41**	14.79
Japanese Newspaper	48,484,390	16,273,137	55.83	20.17	**13.61**	15.22

augmented by both of LCP and Hash table was even 6% slower than SST_L. This result shows that the proposal of speeding-ups by LCP-computation is useful for long non-branching paths, while it remains still open to efficiently handle the complex branching computation of compacted tries preserving the dynamic nature of the compact trie.

6 Conclusion

We presented a faster dynamic compact trie with linear space that supports pattern matching, and insert/delete operations in $O(\lceil(|P|/\alpha)\rceil f(n))$ worst-case time in pattern size P on the Word RAM with $w = O(\log n)$-bit words, where σ is the alphabet size, $\alpha = \lceil w/\log\sigma\rceil$, and $f(n) = O(((\log\log n)^2/\log\log\log n))$.

As most closely related work, we note that the dynamic z-fast trie [2] can directly support pattern matching (from the root), insert, and delete operations in $O(\lceil P/\alpha\rceil + \log L)$ time on average as shown in [2], where L is the maximum size of strings in S. However, we do not know how to support general prefix search with the above time, and also in the worst case.

Acknowledgments

References

1. P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38 – 72, 2002.
2. D. Belazzougui, P. Boldi, and S. Vigna. Dynamic z-fast tries. In *Proc. SPIRE 2010*, Vol. 6393, *LNCS*, 159–172, 2010.

3. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann. Optimal packed string matching. In *Proc. FSTTCS 2011*, Vol. 13, 423–432, 2011.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2 edition, 2001.
5. P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.
6. D. Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer science and computational biology.* Cambridge, 1997.
7. W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. Vitter. Succinct index for dynamic dictionary matching. In *Proc. ISAAC'09, LNCS 5878*, 1034–1043, 2009.
8. S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *Proc. CPM'06*, Vol. 4009, *LNCS*, 60–71, 2006.
9. J. Jansson, K. Sadakane, and W.-K. Sung. Compressed dynamic tries with applications to lz-compression in sublinear time and space. In *Proc. 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007), LNCS 4855*, 424–435, 2007.
10. J. Jansson, K. Sadakane, and W.-K. Sung. CRAM: Compressed random access memory. In *Proc. ICALP'12, LNCS 7391*, 510–521. 2012.
11. Y. Kaneta, H. Arimura, and R. Raman. Faster bit-parallel algorithms for unordered pseudo-tree matching and tree homeomorphism. *JDA*, 14:119–135, 2012.
12. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON'96*, Vol. 1090, *LNCS*, 219–230, 1996.
13. R. Kolpakov, G. Kucherov, and T. Starikovskaya. Pattern matching on sparse suffix trees. In *Proc. CCP'11*, 92–97, 2011.
14. D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
15. T. Uemura and H. Arimura. Sparse and truncated suffix trees on variable-length codes. In *Proc. CPM'11*, Vol. 6661, *LNCS*, 246–260, 2011.
16. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 13(3):249–260, 1995.
17. D. E. Willard. New trie data sturucture which support very fast search operations. *Journal of Computer and System Sciences*, 28:379–394, 1984.

A Appendix

Thoerem 4. In the dynamic compact trie data structure for strings with length α letters (or $w = O(\log n)$-bits), we can implement $LCP(v, X)$ operation in $O(f(n))$ time, where n is the current length of the reference string T, $\alpha = \log_\sigma n$, v is a reference to a node in the micro trie, and X is a pattern string of length at most α letters, packed in a w-bit word.

Proof. We assume without loss of generality that $v = \mathit{root}(\mathcal{T})$, $X = T[k, j]$, and that all keys in S have the same length α. In the case that v is not the root, appropriate preprocessing of v and X is sufficient, and the detail is omitted. To compute the reference $\phi = LCP(v, X)$, it suffice to find the letter depth $c = d_*(\phi) \in [0, \alpha]$, and the real node $p \in \mathcal{N}$ that is the immediate ancestor of ϕ because $\phi = (p', \langle k', j' \rangle)$ can be obtained from these values by $p' = p$, $k' = k + d_*(p)$, and $j' = k + c$.

First, we compute $c = d_*(\phi)$ as follows. We first compute the predecessor and successor of X, $\mathit{Pred}(X)$ and $\mathit{Succ}(X)$, by data structure $\mathcal{D}$. Clearly, we have $\mathit{Pred}(X) \leq X \leq \mathit{Succ}(X)$. If the pattern X completely matches some key X_i with $\mathit{Pred}(X) = \mathit{Succ}(X) = X_i$, then return the reference to the leaf $\phi = \ell_i$, and we are done. Otherwise, from Fig. 1, we can compute $d_*(\phi)$ by the code

$$d_*(\phi) = \max\{\mathit{lcp}(\mathit{Pred}(X), X), \mathit{lcp}(X, \mathit{Succ}(X))\}. \tag{1}$$

The above computation is done in $O(\max\{f(n), \log\log n\})$ since the predecessor and successor are computable in $O(f(n))$ time on $\mathcal{D}$, and the length lcp is computable by XOR and MSB operations in $O(\log\log n)$ time on Word RAM using integer addition for letter alignment (e.g. [11]).

Next, we will find the real node p as the immediate real ancestor of ϕ as follows. For this purpose, we use the next claim on LCA.

Claim 1: Any internal node u of $\mathcal{T}$ can be represented by a pair of some consecutive leaves, that is, $LCA(\ell_{i_*}, \ell_{i_*+1}) = u$ for some $i_* \in [0, K]$.

Proof for Claim 1: Since u is branching, it must have at least two immediate subtrees. Among them, take any consecutive substrees $\mathcal{T}_L, \mathcal{T}_R$ as left and right subtrees and select the rightmost leaf ℓ_L of $\mathcal{T}_L$ and the leftmost leaf ℓ_R of $\mathcal{T}_R$. Then, we can easily show that $LCA(\ell_L, \ell_R) = u$. (*End of Proof for Claim 1*)

By the above lemma, we will find such an index i_*. We define $Z = X[1, c] \in \Sigma^*$ to be the prefix of X with length c, where $c = d_*(\phi)$. Consider all keys of S that correspond to all leaves in the subtree $\mathcal{T}_\phi$ of $\mathcal{T}$ rooted at ϕ. Since these keys, as well as X, have a common prefix Z, they occupy a consecutive interval $\mathcal{I}$ between some indices $1 \leq L \leq R \leq K$ such as

$$X_L \leq \cdots \leq X_h \leq X \leq X_{h+1} \leq \cdots \leq X_R, \tag{2}$$

where $h \in [L, R]$.

To find X_L and X_R, we use the fact that such a maximal interval in $[1, K]$ can be represented by $S \cap [LB, UB]$ for the lexicographically least and greatest α-strings $LB = Zc_0 \cdots c_0$ and $UB = Zc_\sigma \cdots c_\sigma \in \Sigma^\alpha$ with prefix Z, where c_0

and c_σ are the least and greatest letter of Σ (See Fig. 1). From this, we can find $X_L = \mathit{Succ}(LB)$ and $X_R = \mathit{Pred}(UB)$ in $O(f(n))$ time by predecessor data structure $\mathcal{D}$. From this observation, the letter depth $d_*(p)$ of p is given by

$$\begin{aligned} d_*(p) &= \max\{\mathit{lcp}(X_{L-1}, X), \mathit{lcp}(X, X_{R+1})\} \\ &= \max\{\mathit{lcp}(X_{L-1}, X_L), \mathit{lcp}(X_R, X_{R+1})\} \end{aligned} \tag{3}$$

where $X_{L-1} = \mathit{Pred}(X_L)$ and $X_{R+1} = \mathit{Succ}(X_R)$ since p is the lowest branching node p above ϕ.

Claim 2: The above equation correctly gives $d_*(p)$.

Proof for Claim 2: Let X_i be any leaf for some $i < L$. From Eq. 2, we see that X_i is outside and X_L is inside of the subtree $\mathcal{T}_\phi$ rooted at ϕ. Thus, their LCA $u_L = \mathit{lcp}(X_i, X_L)$ must be above the root ϕ. Moreover, if $i \le i' < L$ for two leaves X_i and $X_{i'}$ outside $\mathcal{T}_\phi$, then it follows that $\mathit{lcp}(X_i, X_L)$ locates above $\mathit{lcp}(X_{i'}, X_L)$. By this monotonicity, we conclude that $u_L = \mathit{lcp}(X_{L-1}, X_L)$ is the lowest branching node above ϕ defined by the LCA of X_i and X_L for all $i < L$. By similar argument on $u_R = \mathit{lcp}(X_R, X_{R+1})$, we have the claim. (*End of Proof for Claim 2*)

By Claim 2, we can find the real node p in constant time by accessing array LCA as the lower one of $\mathsf{LCA}(L-1) = \mathit{LCA}(X_{L-1}, X_L)$ and $\mathsf{LCA}(R) = \mathit{LCA}(X_R, X_{R+1})$. Now, we have the depth $c = d_*(\phi)$ and the real node p above ϕ as well as its depth $e = d_*(p)$. Hence, we have the reference $\phi = (p, \langle k', j' \rangle)$ as the answer, where $k' = k + d_*(p) = k + e$ and $j' = k + d_*(\phi) = k + c$. $\square$

Lemma 1. In the semi-dynamic data structure in the small trie case, we can implement $\mathit{Insert}(v, X)$ and $\mathit{Delete}(v, X)$ operations in $O(f(n) + \log \sigma)$ time maintaining the data structure in $O(n \log \sigma + K \log n)$ bits of space, where n is the current length of the reference string T, v is a reference to a node in the micro trie, and X is a pattern string of length at most α letters, packed in a w-bit word. We assume that $d_*(v) + |X| \le \alpha$ so that the height of the micro trie is kept within α letters.

Proof. We make similar assumption to the proof of Theorem 4. Let T be the reference string of the current length n. For simplicity, we consider only the case that only α-letter substring X of T can be freely added or deleted without changing T. In the online case, we modify the algorithm just by appending a new string X to the end of T before insert it.

Initially, the micro trie $\mathcal{T}$ consists only of $\mathit{root}(\mathcal{T})$, and no dictionary is associated to it. If we insert the first string X, we create a leaf below the root that has the label X. We put the pointer to $\mathcal{D}$ at $\mathit{root}(\mathcal{T})$ and the new leaf. Suppose that the data structure contains a string set S with $|S| \ge 2$ and an associated $\mathcal{D}$.

To insert pattern string $X = T[k', j'] \in \Sigma^\alpha$ at the locus $\phi = (p, \langle k, j \rangle)$, we first find the edge e containing ϕ by its reference origin p. Then, we split e at ϕ, create a new real node v there, attach the new leaf ℓ with edge label X in $O(\log \sigma)$ time in a similar way as [16], while we make $\mathit{Insert}(X)$ on $\mathcal{D}$ in

$O(f(n))$ time. We have to correctly maintain the pointers from v and ℓ to $\mathcal{D}$, and also the pointer $\mathsf{LCA}(X') = LCA(X', X) = \phi$ associated to the entry for $X' = Pred(X)$ in $\mathcal{D}$. Hence, the insert operation takes $O(f(n) + \log\sigma)$ time. The delete operation can be implemented similarly in the reverse order with the same time complexity, where we remove the leaf ℓ at the end of e, and remove X from $\mathcal{D}$ only if no pointer on it and if it is necessary. $\square$

Lemma 2. In the full-dynamic data structure in the small trie case, we can implement $insert(v, X)$ and $Delete(v, X)$ operations in $O(f(n) + \log\sigma)$ time in $s = n\log\sigma + O(K\log\sigma + K\log n)$ bits of space maintaining the set S of K strings on Σ with total length n letters, where v is a reference to a node in the micro trie, and X is a pattern string of length at most α letters, packed in a w-bit word.

Proof. We make similar assumption to the proof of Theorem 4. To make the data structure dynamic, we implement the dynamic version of the reference string T using a technique similar to [10] as follows. For a constant $\beta \geq \alpha$, we use a collection of blocks of length β letters, where each block $B = (t, head, tail, prev, next)$ consists of the following fields: the string $t \in \Sigma^\beta$ of consecutive β letters, the relative start and end positions $head$ and $tail$ of the stored string in t in $O(\log\log n)$ bits, the pointers $prev$ and $next$ to the previous and next blocks in $O(\log n)$ bits. If the micro trie stores K real nodes, each edge label s_i, $i \in [1, K]$, is represented in a doubly linked list $B_1, \ldots, B_{\lceil |m_i|/\beta\rceil}$ of $\lceil(|m_i|/\beta)\rceil$ blocks, where each block contains successive β letters of the string s_i. As invariant, we assume that only the first and last blocks can have substrings whose lengths are shorter than β, and the rest of the blocks contains substrings with length exactly β letters.

We now modify the pointer pair $\langle k, j\rangle$ for a substring $T[k, j]$ to be the pair $\langle q, c\rangle$, where q is the pointer to the block containing the beginning of s and $c = |s|$ is the length of s. In insert operation, we split an existing edge label s to ensure that each real node has its own label string separately. This maintenance requires at most one β-block to be split into two β-blocks in $O(\log n)$ time on Word RAM by copying the prefix of suffix of s and maintaining associated pointers. In delete operation, it suffices to remove the specified leaf ℓ and the blocks associated to its edge label string s in the reverse order of insert reclaiming the memory. This is valid because it is ensured as invariant that each node ℓ has its label s as a separate doubly-linked list of blocks. Overall, the total space for the reference string T consisting of K strings with total length n letters is given by $s = \sum_{i=1}^{K} \lceil(|m_i|/\beta)\rceil(\beta\log\sigma + O(\log n + \log\log n)) = n\log\sigma + O(K\log\sigma + K\log n)$ for constant β since $\sum_i |s_i| = n$. This completes the proof. $\square$