# Compressed Dynamic Tries with Applications to LZ-Compression in Sublinear Time and Space

Jesper Jansson[1,⋆], Kunihiko Sadakane[1], and Wing-Kin Sung[2]

[1] Department of Computer Science and Communication Engineering, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
jj@tcslab.csce.kyushu-u.ac.jp, sada@csce.kyushu-u.ac.jp
[2] Department of Computer Science, National University of Singapore, 3 Science Drive 2, 117543 Singapore
Genome Institute of Singapore, 60 Biopolis Street, Genome 138672, Singapore
ksung@comp.nus.edu.sg

**Abstract.** The *dynamic trie* is a fundamental data structure which finds applications in many areas. This paper proposes a compressed version of the dynamic trie data structure. Our data-structure is not only space efficient, it also allows pattern searching in $o(|P|)$ time and leaf insertion/deletion in $o(\log n)$ time, where $|P|$ is the length of the pattern and $n$ is the size of the trie. To demonstrate the usefulness of the new data structure, we apply it to the LZ-compression problem. For a string $S$ of length $s$ over an alphabet $\mathcal{A}$ of size $\sigma$, the previously best known algorithms for computing the Ziv-Lempel encoding (LZ78) of $S$ either run in: (1) $O(s)$ time and $O(s \log s)$ bits working space; or (2) $O(s\sigma)$ time and $O(sH_k + s \log \sigma / \log_\sigma s)$ bits working space, where $H_k$ is the $k$-order entropy of the text. No previous algorithm runs in sublinear time. Our new data structure implies a LZ-compression algorithm which runs in sublinear time and uses optimal working space. More precisely, the LZ-compression algorithm uses $O(s(\log \sigma + \log\log_\sigma s)/\log_\sigma s)$ bits working space and runs in $O(s(\log\log s)^2/(\log_\sigma s \log\log\log s))$ worst-case time, which is sublinear when $\sigma = 2^{o(\log s \frac{\log\log\log s}{(\log\log s)^2})}$.

## 1 Introduction

A *trie* [7] is a rooted tree in which every edge is labeled by a symbol from an alphabet $\mathcal{A}$ in such a way that for every node $u$ and every $a \in \mathcal{A}$, there is at most one edge from $u$ to a child of $u$ that is labeled by $a$. (From here on, we assume $\mathcal{A}$ is fixed and define $\sigma = |\mathcal{A}|$.) Each leaf $\ell$ in the trie represents a string obtained by concatenating the symbols on the unique path from the root to $\ell$; thus, a trie can be used to store a set of strings over $\mathcal{A}$. A *dynamic* trie is a fundamental data structure allowing operations to modify it dynamically, i.e., allowing strings to be inserted or deleted from the trie. It find applications in many areas including information retrieval, natural language processing, database systems, compilers, data compression, and computer networks. As an example, in

⋆ Supported by Japan Society for the Promotion of Science (JSPS).

V. Arvind and S. Prasad (Eds.): FSTTCS 2007, LNCS 4855, pp. 424–435, 2007.

computer networks, dynamic tries are used in IP routing to efficiently maintain the hierarchical organization of routing information to enable fast lookup of IP addresses [14]. In data compression, dynamic tries are used to represent the so-called LZ-trie and the Huffman coding trie which are the key data structures in the Ziv-Lempel encoding (LZ78) [20] (or its variant LZW encoding [17]) and the Huffman encoding, respectively. Furthermore, many data structures such as the suffix trie/suffix tree, the Patricia trie [11], and the associative array (hashing table) can be maintained as dynamic tries.

Without loss of generality, assume $\sigma \leq n$. A dynamic trie $T$ of size $n$ can be implemented using a standard tree data-structure in $O(n \log n)$ bits space such that: (1) insertion or deletion of a leaf into or from $T$ takes $O(1)$ time; and (2) finding the longest prefix of a query pattern $P$ in $T$ takes $O(|P|)$ time. A number of solutions have been proposed to improve the average time and space complexities of tries [1,2,11]. However, in the worst case, those solutions still use $O(n \log n)$ bits space and pattern searching still requires $O(|P|)$ time. Employing the latest advances on compressed trees, a trie can now be maintained in $O(n \log \sigma)$ bits space under the unit-cost RAM model such that: (1) insertion or deletion of a leaf takes $O(\log n)$ time; and (2) the longest common pattern query takes $O(|P|)$ time. Note that none of the existing data structures can answer the longest common pattern query in $o(|P|)$ time.

This paper assumes a unit-cost RAM model with word size logarithmic in $n$, in which standard arithmetic and bitwise boolean operations on word-sized operands can be performed in constant time [9]. Also, we assume the pattern $P$ is packed in $O(|P| \log \sigma / \log n)$ words. Under such a model, we propose a data structure which uses $O(n \log \sigma)$ bits such that: (1) insertion or deletion of a leaf takes $O((\log \log n)^2 / \log \log \log n)$ time; and (2) the longest common pattern query takes $O(\frac{|P|}{\log_\sigma n} \frac{(\log \log n)^2}{\log \log \log n})$ time. Note that when $\sigma = 2^{o(\log n \frac{\log \log \log n}{(\log \log n)^2})}$, our $O(n \log \sigma)$-bits dynamic trie data-structure can be maintained such that the longest common pattern query can be performed in $o(|P|)$ time while insertion and deletion takes $o(\log n)$ time.

In this paper we define "sublinear" as follows. We assume that the alphabet size $\sigma$ is a function of $n$ (or a constant). We say the space is sublinear if it is $o(n \log \sigma)$ because $n \log \sigma$ is the input size. We say the time is sublinear if it is $o(n \log \sigma)$. Note that no algorithm can achieve sublinear time for large alphabets such as $\log \sigma = \Omega(\log n)$ because it takes $\Omega(\frac{n \log \sigma}{\log n})$ time to read the input. We give sublinear time algorithms when $\sigma = 2^{o(\log n \frac{\log \log \log n}{(\log \log n)^2})}$.

Our improvement stems from the observation that small tries (that is, tries of size $O(\log_\sigma n)$) can be maintained very efficiently. Hence, our data structures partition the trie into many small tries and maintain them individually. With this approach, we not only store the trie using $O(n \log \sigma)$ bits, but also allow fast queries and efficient insertions and deletions.

To demonstrate the usefulness of our dynamic trie data structure, we applied it to generate the LZ78 encoding of a text. The Ziv-Lempel encoding (LZ78) [20] (or its variant LZW encoding[17]) of a text is a popular compression scheme.

Ziv and Lempel [20] showed that the LZ78 encoding scheme gives an asymptotically optimal compression ratio.

The current solutions for constructing the LZ78 encoding of a text first construct the LZ-trie and then generate the LZ78 encoding. These solutions either run in: (1) $O(s)$ time and $O(s \log s)$ bits working space [5,15]; or (2) $O(s\sigma)$ time and $O(s \log \sigma)$ bits working space [3]. None of the solutions in the literature runs in sublinear time and $O(s \log \sigma)$-bit working space. By maintaining the LZ-trie using our dynamic trie data structure, we obtain the first LZ compression algorithm which uses optimal working space and runs in sublinear time when $\sigma = 2^{o(\log s \frac{\log \log \log s}{(\log \log s)^2})}$. More precisely, we propose an algorithm which uses $O(s(\log \sigma + \log \log_\sigma s)/\log_\sigma s)$ bits working space and runs in in $O(s(\log \log s)^2/(\log_\sigma s \log \log \log s))$ worst-case time. Note that the working space is asymptotically smaller than the outputted compressed text.

The paper is organized as follows. Section 2 reviews some previously known facts about tries and LZ78 encoding. Section 3 defines the LZ78 encoding and gives some simple data structures that are useful for maintaining a LZ-trie. Sections 4 and 5 detail our dynamic trie data structure. Finally, Section 6 presents our LZ compression algorithms.

## 2 Previous Work

A dynamic trie data structure can be implemented naively using $O(n \log n)$ bits such that: (1) insertion and deletion of a leaf takes $O(1)$ time; and (2) the longest prefix of any query pattern $P$ in $T$ can be found in $O(|P|)$ time. Many practical improvements have been proposed which yield good performance (on average) for searching a pattern. Morrison [11] proposed the Patricia trie which compresses a path by merging the nodes of degree 2. This idea reduces the size of the trie. Later, Andersson and Nilsson [1] proposed the LC-trie, which reduces the depth of the trie by increasing the branching factor (level compression). This idea reduces the average running time [6].

Willard [18,19] proposed two data structures for maintaining a trie of depth $O(\log M)$ for some positive integer $M$: (1) the Q-fast trie [19], which uses $O(n \log M)$ bits space and searches for the pattern $P$ in $T$ in $O(\sqrt{\log M})$ time while inserting or deleting a leaf in $O(\sqrt{\log M})$ time; and (2) the Y-fast trie [18], which is a static trie that uses $O(n \log M)$ bits space and can report the longest prefix of any pattern $P$ in $T$ in $O(\log \log M)$ time.

*Ziv-Lempel encoding* (LZ78) is a widely used encoding scheme for compressing a text [17,20]. LZ78 also has applications in compressed indexing; Navarro [13] presented a compressed full-text self-index called *LZ-index* based on the LZ-trie whose size is proportional to the compressed text size. The LZ-index allows efficient pattern queries.

A straightforward implementation of LZ78 based on Lempel and Ziv's original definition takes $O(n^2)$ worst-case time to process a string of length $n$. Rodeh, Pratt, and Even [15] improved the running time to $O(n)$ using suffix trees, and

Brent [5] gave another linear time compression algorithm based on hashing. However, both algorithms use $O(n \log n)$-bits working space. This is larger than the size of the Ziv-Lempel encoding, which is $O(nH_k)$ where $H_k$ is the $k$-order entropy of the text. People have recently realized the importance of space-efficient data compression algorithms [3,10]. Given a long text, we may have enough memory to store the compressed text (that is, the Ziv-Lempel encoding). However, we may be unable to construct it if the working space requirement is too large. For example, we are able to store the Ziv-Lempel encoding of the human genome in a 2GB RAM computer, but we may fail to construct the encoding due to the size of the memory. Hence, a space-efficient construction algorithm is necessary. Utilizing the solution of Arroyuelo and Navarro [3], the Ziv-Lempel encoding of a text can be constructed using $O(\sigma n)$ time and $O(nH_k + n \log \sigma / \log_\sigma n)$ bits working space.

## 3 Preliminaries

We first reviews simple data structures used for dynamically maintaining a set of length-$(\log_\sigma n)$ strings and a tree, respectively, in Sections 3.1 and 3.2. These data structures are the building blocks of our dynamic trie data structure, which is used to dynamically maintain a LZ-trie. Section 3.3 reviews the definitions of the LZ78 encoding and the LZ-trie.

### 3.1 A Data Structure for Maintaining a Set of Length-$(\log_\sigma n)$ Strings

This subsection describes a dynamic data structure for maintaining a set of $k$ strings, each of length at most $\log_\sigma n$, over an alphabet of size $\sigma$. It needs to support three operations: (1) insertion of a length-$(\log_\sigma n)$ string, (2) deletion of a length-$(\log_\sigma n)$ string, and (3) predecessor of a string $P$ (that is, reporting the string currently in the set which is lexicographically just smaller than $P$).

We make use of the *dynamic predecessor data structure* of Beame and Fich [4], whose properties are summarized in the next lemma:

**Lemma 1 ([4]).** *The dynamic predecessor data structure of Beame and Fich [4] can maintain a set of $\ell$ $O(\log n)$-bit integers using $O(\ell \log n)$ bits under insertions and deletions while supporting predecessor queries so that each insert/delete/predecessor operation takes $O((\log \log n)^2/(\log \log \log n))$ time.*

We immediately obtain:

**Lemma 2.** *Consider $k$ strings of length at most $\log_\sigma n$ over an alphabet of size $\sigma$. We can store all strings in $O(k \log n)$ bits such that insert/delete/predecessor can be found in $O((\log \log n)^2 / \log \log \log n)$ time.*

*Proof.* Treat the strings as integers in the range $\{0, 1, \ldots, n-1\}$ and apply Lemma 1. □

### 3.2 Data Structures for Maintaining an Edge-Labeled Tree

This section discusses how to dynamically maintain an edge-labeled tree $T$. We assume the size of the tree and all labels are integers smaller than $n$. We support the following operations:

- $Insert(u, \kappa, v)$: Insert a leaf $v$ as a child of $u$ and label the edge $(u, v)$ by $\kappa$.
- $Delete(v)$: Delete the leaf $v$ and the edge between $v$ and its parent (if any).
- $Child(u, \kappa)$: Return the child $v$ of $u$ such that the edge $(u, v)$ is labeled by $\kappa$.

**Lemma 3.** *A tree $T$ can be maintained dynamically in $O(|T| \log n)$ bits space such that $Child/Insert/Delete$ can be answered in $O((\log\log n)^2/(\log\log\log n))$ time.*

*Proof.* We represent $T$ using two dynamic predecessor data structures $\mathcal{D}_1$ and $\mathcal{D}_2$, as in Lemma 1. For each edge $(u, v)$ labeled by $\kappa$, we maintain $n^2 \cdot u + n \cdot \kappa + v$ in $\mathcal{D}_1$ and $n^2 \cdot v + n \cdot u + \kappa$ in $\mathcal{D}_2$. $\mathcal{D}_1$ and $\mathcal{D}_2$ take $O(|T| \log n)$-bit space. Since $u, v, \kappa \leq n$, there is a one-to-one mapping between $(u, v, \kappa)$ and the number $w = n^2 \cdot u + n \cdot \kappa + v$ in $\mathcal{D}_1$. To be precise, $v = w \bmod n$, $u = \lfloor w/n^2 \rfloor$, $\kappa = \lfloor (w - u \cdot n^2)/n \rfloor$. Similarly for $\mathcal{D}_2$.

To insert a leaf node $v$, which is a child of $u$ with edge label $\kappa$, it can be done by inserting $n^2 \cdot u + n \cdot \kappa + v$ in $\mathcal{D}_1$ and $n^2 \cdot v + n \cdot u + \kappa$ in $\mathcal{D}_2$.

To delete a leaf node $v$, we first query $\mathcal{D}_2$ to retrieve the integer $w$ which is just bigger than $n^2 \cdot v$. Note that $w = n^2 \cdot v + n \cdot u + \kappa$ where $u$ is the parent of $v$ and $\kappa$ is the label of $(u, v)$. Then, the leaf node $v$ can be removed by deleting $n^2 \cdot u + n \cdot \kappa + v$ from $\mathcal{D}_1$ and $n^2 \cdot v + n \cdot u + \kappa$ from $\mathcal{D}_2$.

To compute $Child(u, \kappa)$, we first retrieve the integer $w$ which is just bigger than $n^2 \cdot u + n \cdot \kappa$ in $\mathcal{D}_1$. Then, $Child(u, \kappa)$ equals the remainder when we divide $w$ by $n$.

The running time for each of the three operations is $O((\log\log n)^2/(\log\log\log n))$ time by Lemma 1. □

### 3.3 LZ78 Encoding and LZ-Trie

Ziv-Lempel encoding [20], or LZ78, is a data compression scheme for strings. For a given string $S = S[1..n]$, it constructs a *phrase list* and a LZ-*trie* procedurely using the following method: First, initialize a trie $T$ as empty, the current position $p = 1$, and the number of phrases $c = 0$. Then, parse $S$ into phrases from left to right until $p > n$ as follows. Find the longest string, $t \in T$, that appears as a prefix of $S[p..n]$. Set $c = c + 1$. Obtain the phrase $s_c = S[p..p + |t|] = t \cdot S[p + |t|]$ and insert it into $T$. Then, set $p = p + |t| + 1$ and repeat the parsing for the next phrase.

The trie $T$ generated during the above process is called the LZ-*trie* while the list of phrases $s_1, s_2, \ldots, s_c$ is called the *phrase list*. The Ziv-Lempel encoding of the given string $S$ consists of the LZ-trie together with the phrase list for $S$. By [20], it holds that $\sqrt{n} \leq c \leq n/\log_\sigma n$. Also, the LZ-trie and the phrase list can be stored in $c \log c + O(c \log \sigma) = nH_k + O(n \log \sigma / \log_\sigma n)$ bits.

## 4 Dynamically Maintaining a Trie of Height $\log_\sigma n$

In this and the next section, we show how to maintain a trie while efficiently supporting the following operations:

- $Insert(T, u, a)$: Insert a leaf $v$ as a child of $u$ such that the label of $(u, v)$ is $a$, where $a \in \mathcal{A}$.
- $Delete(T, u)$: Delete the leaf $u$ and the edge between $u$ and its parent (if any).
- $Lcp(T, P)$: Report the length $\ell$ such that $P[1..\ell]$ is the longest prefix which exists in $T$.

Here, we discuss the dynamic trie data structure for small tries. First, we consider how to maintain a trie of size $O(\log_\sigma n)$. Then, we study how to maintain a trie of height at most $\log_\sigma n$. (In the next section, we discuss how to maintain a general trie.)

### 4.1 Maintaining a Trie of Size $O(\log_\sigma n)$

This subsection describes how to dynamically maintain a trie $T$ of size $O(\log_\sigma n)$.

**Lemma 4.** *Given a precomputed table of size $O(n^{5\epsilon})$ bits for any constant $0 < \epsilon < 0.2$, we can maintain a trie $T$ of size $\epsilon \log_\sigma n$ using at most $3\epsilon \log n$ bits. All operations Lcp, Insert, and Delete take $O(1)$ worst case time. Also, preorder of any node can be computed in $O(1)$ time.*

*Proof.* The data structure has two parts. First, the topology of $T$ is stored in $2|T| = 2\epsilon \log_\sigma n$ bits using parenthesis encoding [12,8]. Second, the edge labels of all edges are stored in preorder using $|T| \log \sigma = \epsilon \log n$ bits. Therefore the total space is at most $3\epsilon \log n$ bits.

In addition, the data structure also requires four pre-computed tables. The first table stores the value of $Lcp(R, Q)$ for any trie $R$ of size at most $\epsilon \log_\sigma n$ and any string $Q$ of length at most $\epsilon \log_\sigma n$. The second table stores the value of $preorder(R, Q)$, which is the preorder of any string $Q$ in the trie $R$ for any trie $R$ of size at most $\epsilon \log_\sigma n$ and any string $Q$ of length at most $\epsilon \log_\sigma n$. Since there are $O(2^{2\cdot\epsilon \log_\sigma n} \cdot \sigma^{\epsilon \log_\sigma n} \cdot \sigma^{\epsilon \log_\sigma n}) = O(n^{4\epsilon})$ different combinations of $R$ and $Q$, both tables can be stored in $O(n^{4\epsilon} \log\log_\sigma n) = O(n^{5\epsilon})$ bits space. The size of the tables for insert/delete is $O(2^{2\cdot\epsilon \log_\sigma n} \cdot \sigma^{\epsilon \log_\sigma n} \cdot \epsilon \log_\sigma n \cdot \sigma \cdot \epsilon \log n) = O(n^{5\epsilon})$.

The four operations can be supported in $O(1)$ time as follows using a precomputed table for each operation.

- To insert/delete a node $x$, we update the topology and the edge label.
- $Lcp(T, P)$ can be computed by asking $O(1)$ queries. in the precomputed table.
- Preorder of any string in $T$ can also be computed in $O(1)$ time. □

**Lemma 5.** *The tables for Lcp() and preorder() can be constructed incrementally using $O(\log_\sigma n)$ time per entry. When the size of the tables is $n$, $Lcp(R, Q)$ and $preorder(R, Q)$ queries can be answered in $O(1)$ time for any $R$ of size at most $0.2 \log_\sigma n$ and $Q$ of length at most $0.2 \log_\sigma n$.*

### 4.2 Maintaining a Trie of Height $O(\log_\sigma n)$

This section describes how to dynamically maintain a trie of height $O(\log_\sigma n)$.

**Lemma 6.** *Given a precomputed table of size $O(n^{5\epsilon})$ bits for any constant $0 < \epsilon < 0.2$, we can dynamically maintain a trie $T$ of height $\frac{\epsilon}{2}\log_\sigma n$ using $O(|T|\log\sigma)$ bits space such that all operations Lcp, Insert, and Delete take $O((\log\log n)^2/\log\log\log n)$ time.*

*Proof.* Let $u_i$ be the node in $T$ whose preorder is $i$. Let $S = \{s_1, s_2, \ldots, s_{|T|}\}$ be the set of strings where $s_i$ is the string representing the path label of $u_i$. Note that the $s_i$'s are sorted in alphabetical order.

A block is defined to be a series of strings $s_i, s_{i+1}, \ldots, s_j$ where $i \leq j \leq |T|$. Note that all strings in a block can be represented as a subtrie of $T$. The nodes $u_i, u_{i+1}, \ldots, u_j$ are connected if we add the nodes on the path from the root to $u_i$. Therefore the size of the subtrie is at most $j - i + 1 + \frac{\epsilon}{2}\log_\sigma n$.

The set $S$ can be partitioned into a set $\mathcal{B} = \{B_1, B_2, \ldots B_{|\mathcal{B}|}\}$ of non-overlapping blocks such that $B_1 \cup B_2 \cup \ldots \cup B_{|\mathcal{B}|} = S$. We also maintain the invariant that (1) every block contains at most $\frac{\epsilon}{2}\log_\sigma n$ strings and (2) at most one block has less than $\frac{\epsilon}{4}\log_\sigma n/2$ strings. Besides, for each $B_i \in \mathcal{B}$, let $s_{b(i)}$ be the smallest string in $B_i$.

Our dynamic data structure represents the trie $T$ using a two-level data structure.

- **(1) Top-level:** Using the data structure in Lemma 2, we store $\{s_{b(1)}, \ldots, s_{b(|\mathcal{B}|)}\}$.
- **(2) Block-level:** For each block $B_i \in \mathcal{B}$, we can represent the strings in $B_i$ as a trie of size $\epsilon\log_\sigma n$ and store the trie using Lemma 4.

We first show that the space required is $O(|T|\log\sigma)$ bits. Note that $|\mathcal{B}| = O(\frac{|T|}{\epsilon\log_\sigma n})$ blocks. The space required for the top-level structure is $O(\epsilon^{-1}|\mathcal{B}|\log n) = O(\epsilon^{-1}|T|\log\sigma)$ bits. Each block requires $O(\log n)$ bit space by Lemma 4. The space for the block-level structure is $O(|\mathcal{B}|\log n) = O(|T|\log\sigma)$.

The time complexity of the three operations is as follows.

- $Lcp(T, P)$: Let $P'$ be the first $\frac{\epsilon}{2}\log_\sigma n$ characters of $P$. To compute the longest common prefix of $P$ in $T$, we first find $s_i$ and $s_{i+1}$ such that $P'$ is alphabetically in between $s_i$ and $s_{i+1}$; let $lcp_1$ be the longest common prefix of $P'$ and $s_i$ and $lcp_2$ be the longest common prefix of $P'$ and $s_{i+1}$; then, $Lcp(T, P)$ equals the maximum of $lcp_1$ and $lcp_2$. To locate $s_i$, our strategy is to first locate the $s_{b(j)}$ which is alphabetically just smaller than or equal to $P'$. By Lemma 2, $s_{b(j)}$ can be found in $O((\log\log n)^2/\log\log\log n)$ time. Then, within $B_j$, we locate the $s_i$ just smaller than or equal to $P'$. By Lemma 4, this step takes $O(1)$ time.
- $Insert(T, u, a)$: Suppose $u$ represents a string $s \in S$. This operation is equivalent to insert a new string $s \cdot a$ after $s$. Let $B_j$ be the block containing $s$. We first insert $s \cdot a$ into $B_j$ using $O(1)$ time by Lemma 4. If $B_j$ contains less than $\frac{\epsilon}{2}\log_\sigma n$ strings, then the insert operation is done. Otherwise, we need

to split $B_j$ into two blocks each containing at least $\frac{\epsilon}{4}\log_\sigma n$ strings. The split takes $O(1)$ time since $B_j$ is packed in $O(\log n)$ bits. Lastly, we update the top-level structure to indicate the existence of the new block, which takes $O((\log\log n)^2/\log\log\log n)$ time.

– *Delete*$(T, u)$: The analysis is similar to the Insert operation. □

## 5 Maintaining a Trie with No Height Restrictions

This section gives a data structure to dynamically maintain a general trie $T$. We also show how to build an auxiliary data structure for $T$ using $O(|T|)$ time such that the preorder of any node can be reported in $O(\log\log n)$ time.

We describe a dynamic data structure for a trie $T$ such that insertion/deletion of a leaf takes $O((\log\log n)^2/\log\log\log n)$ time and longest common prefix of $P$ can be computed in $O(\frac{|P|}{\log_\sigma n}\frac{(\log\log n)^2}{\log\log\log n})$ time.

Our data structure represents a general trie $T$ by partitioning it into tries of height at most $h = \frac{\epsilon}{2}\log_\sigma n$ for some constant $0 < \epsilon < 0.2$. To formally describe the representation, we need some definitions.

Let $\delta = h/3$. For any node $u \in T$, $u$ is denoted as a *linking node* if (1) the height of $u$ is of multiple of $\delta$ and (2) the subtrie rooted at $u$ has more than $\delta$ nodes.

Let $LN$ be the set of linking nodes of $T$. For any $u \in LN$, let $\tau_u$ be the subtrie of $T$ rooted at $u$ including all descendents $v$ of $u$ such that there is no linking node in the path between $u$ and $v$. For any non-root node $v \in T$, we denote by $p(v)$ the linking node such that $p(v)$ is the lowest ancestor of $u$ in $T$.

Let $T'$ be a tree whose vertex set is $LN$ and whose edge set is $\{(p(u), u) \mid u \in LN$ and $u$ is not the root$\}$. The label of every edge $(p(u), u)$ in $T'$ is the length-$\delta$ string represented by the path from $p(u)$ to $u$ in $T$.

Based on the above discussion, $T$ can be represented by storing (1) $T'$ and (2) $\tau_u$ for all $u \in LN$. The next lemma bounds the size of $LN$.

**Lemma 7.** *$|LN| \le |T|/\delta$. Also, for any $u \in LN$, $\tau_u$ is of height smaller than $2\delta$.*

*Proof.* Each $u \in LN$ has at least $\delta$ unique nodes associated to it. Hence $|T| = \sum_{u\in LN}|\tau_u| \ge |LN|\delta$. Thus, $|LN| \le |T|/\delta$. By construction, $\tau_u$ is of height smaller than $2\delta$. □

The theorem below is our main result. It states how to maintain $T'$ and $\tau_u$ for all $u \in LN$.

**Theorem 1.** *We can dynamically maintain a trie $T$ using $O(|T|\log\sigma)$ bits space such that $Lcp(T, P)$ takes $O(\frac{|P|}{\log_\sigma n}\frac{(\log\log n)^2}{\log\log\log n})$ time while insertion/deletion of a leaf takes $O((\log\log n)^2/\log\log\log n)$ time.*

*Proof.* We represent $T'$ by Lemma 3 using $O(|T'|\log n) = O(\frac{|T|}{\log_\sigma n}\log n) = O(|T|\log\sigma)$ bits. For every $u \in LN$, the height of $\tau_u$ is bounded according to Lemma 7, so we can represent $\tau_u$ as in Lemma 6 using $O(|\tau_u|\log\sigma)$ bits. Since

$\sum_{u \in LN} |\tau_u| = |T|$, all $\tau_u$'s can be represented in $O(|T| \log \sigma)$ bits. Also, we maintain the lookup tables for answering queries $Lcp(R, Q)$ and $preorder(R, Q)$ for any tree $R$ of size at most $\epsilon \log_\sigma |T|$ and any query $Q$ of length at most $\epsilon \log_\sigma |T|$ where $0 < \epsilon < 1$.

For $Lcp(T, P)$, the longest prefix of $P$ which exists in $T$ can be found in two steps. First, we find the longest prefix of $P$ in $T'$. It is done in $O(\frac{|P|}{\log_\sigma n} \frac{(\log \log n)^2}{\log \log \log n})$ time using the predecessor data structure in Lemma 3. Suppose $u$ is the node in $T'$ corresponding to the longest prefix $P[1..x]$ of $P$. Second, we find the longest prefix of $P[x+1..|P|]$ in $\tau_u$. By Lemma 6, it takes another $O(\frac{(\log \log n)^2}{\log \log \log n})$ time.

For insertion/deletion of a leaf node $u$, suppose we need to insert/delete the leaf node $u$ in the subtrie $\tau_v$ where $v \in LN$. By Lemma 6, it takes $O(\frac{(\log \log n)^2}{\log \log \log n})$ time. Moreover, if the insertion/deletion creates/destroys a new linking node $v'$ in $\tau_v$, we need to do the following additional steps. (1) Insert/delete a new leaf in $T'$ corresponding to $v'$ (This step can be done in $O(\frac{(\log \log n)^2}{\log \log \log n})$ time by Lemma 3); (2) Create/delete a new subtrie $\tau_{v'}$ (Since $\tau_{v'}$ is of size smaller than $\log_\sigma n$, we can create/delete it in $O(1)$ time); and (3) Insert/delete $\tau_{v'}$ from $\tau_v$ (Since $\tau_{v'}$ is stored in $O(1)$ blocks in $\tau_v$, we can modify those blocks in $O(\frac{(\log \log n)^2}{\log \log \log n})$ time). (4) For every insertion, if the size of the lookup tables $Lcp()$ and $preorder()$ is smaller than $n^\epsilon$, we incrementally increase the size of the tables by one using Lemma 5. For every deletion, if the size of the tables is bigger than $2n^\epsilon$, we reduce the size of the tables by one using Lemma 5. □

The following lemma states how to build an auxiliary data structure for $T$ to answer preorder queries.

**Lemma 8.** *Given a trie $T$ represented by the dynamic data structure in Theorem 1, we can generate an auxiliary data structure of size $O(|T| \log \sigma)$ bits in $O(|T|)$ time such that the preorder of a node can be computed in $O(\log \log n)$ time.*

*Proof.* The auxiliary data structure stores information for every linking node $u$ (that is, $u \in T'$). First, we store the preorder of $u$. Then, for the corresponding subtrie $\tau_u$, define $\mathcal{B}$ and the set $\{s_{b(1)}, s_{b(2)}, \ldots s_{b(|\mathcal{B}|)}\}$ as in Lemma 6. We store three information below.

1. By Lemma 2, using $O(|\mathcal{B}|(\log \log n)^2 / \log \log \log n)$ time, we extract all strings in $\{s_{b(1)}, s_{b(2)}, \ldots s_{b(|\mathcal{B}|)}\}$. The set $\{s_{b(1)}, s_{b(2)}, \ldots s_{b(|\mathcal{B}|)}\}$ is stored in $O(|\mathcal{B}| \log n)$ bits space using $O(|\mathcal{B}| \log \log n)$ time by the y-fast trie data structure [18]. Then, given any string $P$, we can report the largest $i$ such that $s_{b(i)}$ is alphabetically smaller than or equal to $P$ using $O(\log \log n)$ time.
2. It stores an array $V[1..|\mathcal{B}|]$ where $V[j]$ equals the preorder values of the $s_{b(i)}$. Since each preorder value can be stored in $\log n$ bits, the array $V$ can be stored in $|\mathcal{B}| \log n = O(|T|)$ bits.
3. For each $B_i \in \mathcal{B}$, all strings in $B_i$ are represented as a trie of size $O(\log n)$ bits using Lemma 4.

For any node $v \in T$, let $u$ be the linking node that is the lowest ancestor of $u$ in $T$. Let $B$ be the block in $\tau_u$ which contains $v$ and $w$ be the node in $\tau_u$ corresponds to the smallest string in $B$. Note that the preorder of $v$ equals the sum of (1) the preorder of $u$ in $T$, (2) the preorder of $w$ in $\tau_u$, and (3) the preorder of $v$ in $B$.

For (1), the preorder of $u$ in $T$ is stored in the auxiliary data structure. For (2), by y-fast trie, using $O(\log\log n)$ time, we can find the preorder of $w$ in $\tau_u$. For (3), by Lemma 4, the preorder $v$ in $B$ can be determined in $O(1)$ time. The lemma follows. □

## 6 LZ-Compression

This section gives a two-phase algorithm to construct the LZ-compression of the input text $S[1..s]$. The first phase constructs the LZ-trie based on the trie data structure in Theorem 1. Then, it enhances the LZ-trie with an auxiliary data structure so that preorder of any node can be computed efficiently using Lemma 8. The second phase generates the phrase list. It scans the text $S$ to output the list of preorders of the phrases. Fig. 1 describes the details of the algorithm. The lemma below states the running time of our algorithm. We assume a unit-cost RAM model with word size $\log s$, and $\sigma \leq s$.

**Lemma 9.** *Suppose we use the trie data structure in Theorem 1. The algorithm in Fig. 1 builds the* LZ*-trie $T$ and the phrase list using $O(\frac{s}{\log_\sigma s}\frac{(\log\log s)^2}{\log\log\log s})$ time and $O(\frac{s(\log\sigma+\log\log_\sigma s)}{\log_\sigma s})$ bits working space.*

*Proof.* Phase 1 builds the trie $T$ through the while-loop in Step 4 of Fig. 1. Since there are $c$ phrases, the while-loop will execute $c$ times and generate $c$ phrases $s_1, s_2, \ldots, s_c$. For the $i$-th iteration, by Theorem 1, Step 4.1 can find $s_i$ in $O(\frac{|s_i|}{\log_\sigma s}\frac{(\log\log s)^2}{\log\log\log s})$ time. Step 4.2 stores the length of $s_i$ by delta-code in $1+\lceil\log s_i\rceil+2\lceil\log(1+\lceil\log s_i\rceil)\rceil$ bits. Then, Step 4.3 inserts the phrase $s_i$ into the trie $T$ using $O((\log\log s)^2/\log\log\log s)$ time. Finally, the LZ-trie $T$ is enhanced with an auxiliary data structure for preorder by Lemma 8.

Since $\sum_{i=1}^{c}|s_i| = s$, the $c$ iterations take $O(\sum_{i=1}^{c}\frac{|s_i|}{\log_\sigma s}\frac{(\log\log s)^2}{\log\log\log s})$ time, which equals $O(\frac{s}{\log_\sigma s}\frac{(\log\log s)^2}{\log\log\log s})$ time. The auxiliary data structure is constructed using $O(c) = O(\frac{s}{\log_\sigma s})$ time.

Given the trie $T$ and the string $S$, Phase 2 first enhances the data structure so that preorder of any node in $T$ can be computed in $O(\log\log s)$ time by Lemma 8. For each phrase $s_i$, we first obtain its length $\ell$ stored by delta-code. Then we search the trie for the node representing the phrase $s_i = S[p..p+\ell-1]$. It takes $O(\frac{|s_i|}{\log_\sigma s}\frac{(\log\log s)^2}{\log\log\log s})$ time by Theorem 1. The preorder of the phrase $s_i$ can be computed in $O(\log\log s)$ time. In total, Phase 2 takes $O(\sum_{i=1}^{c}\frac{|s_i|}{\log_\sigma s}\frac{(\log\log s)^2}{\log\log\log s})$ time, which equals $O(\frac{s}{\log_\sigma s}\frac{(\log\log s)^2}{\log\log\log s})$ time.

**Algorithm** *LZcompress*
**Input:** A sequence $S[1..s]$.
**Output:** The compressed text of $S$.

**1** Initialize $T$ as an empty trie. /* Phase 1: Construct the trie tree $T$ */
**2** Denote empty phrase as phrase 0.
**3** $p = 1$;
**4** **while** $p \leq n$ **do**
**4.1** Find the longest phrase $t \in T$ that appears as a prefix of $S[p..s]$.
**4.2** Store the length of $t$ by delta-code.
**4.3** Insert the phrase $t \cdot S[p + |t|]$ into $T$.
**4.4** $p = p + |t| + 1$;
**endwhile**
**5** Enrich the trie $T$ so that we can compute the preorder of any node in $T$ by Lemma 8.
**6** $p = 1$; $j = 1$ /* Phase 2: Construct the phrase list $s_1 s_2 \ldots s_c$ */
**7** **while** $p \leq n$ **do**
**7.1** Obtain the length $\ell$ of the next phrase stored by delta-code.
**7.2** Find the phrase $t = S[p..p + \ell - 1] \in T$.
**7.3** $s_j$ = preorder index of $t$ in $T$
**7.4** Output $s_j$.
**7.5** $p = p + |t| + 1$; $j = j + 1$;
**endwhile**
**End** *LZcompress*

**Fig. 1.** Algorithm for LZ-compression

In total, the running time is $O(\frac{s}{\log_\sigma s} \frac{(\log\log s)^2}{\log\log\log s})$ time. The working space required to store the LZ-trie is $O(c \log \sigma) = O(\frac{s \log \sigma}{\log_\sigma s})$ bits, and the space for storing lengths of the phrases is $\sum_{i=1}^{c} O(1 + \log s_i) = O(c \log \frac{s}{c}) = O(\frac{s \log\log_\sigma s}{\log_\sigma s})$. □

As a final remark, the working space of the algorithm is precisely $O(c \log \sigma + c \log\log_\sigma s)$ where $c$ is the number of phases output. Since $c \geq \sqrt{s}$, the working space must be asymptotically smaller than the output size, which is $O(c \log c + c \log \sigma)$. Note that the output size is larger than $c \log c \geq \frac{1}{2}\sqrt{s} \log s$, while the tables used in the algorithm have size $O(s^\epsilon)$ for arbitrarily small $\epsilon > 0$.

Secondly, the output codes of the algorithm in Fig. 1 are different from the original LZ78. The algorithm outputs the same codes as [16][1]. Then we can decode any substring of $S$ of length $O(\log_\sigma s)$ in constant time. The output size of [16] is asymptotically the same as the original LZ78.

[1] More precisely, the output codes represents preorders of the trie. To convert it into the original LZ78, we need one more scan of $S$ using the trie.

## References


1. Andersson, A., Nilsson, S.: Improved behaviour of tries by adaptive branching. Information Processing Letters 46, 295–300 (1993)
2. Aoe, J.: An efficient digital search algorithm by using a double array structure. IEEE Transactions on Software Engineering 15(9), 1066–1077 (1989)
3. Arroyuelo, D., Navarro, G.: Space-efficient construction of LZ-index. In: Deng, X., Du, D.-Z. (eds.) ISAAC 2005. LNCS, vol. 3827, Springer, Heidelberg (2005)
4. Beame, P., Fich, F.E.: Optimal bounds for the predecessor problem. In: Proc. of the $31^{st}$ Annual ACM Symposium on the Theory of Computing (STOC 1999), pp. 295–304 (1999)
5. Brent, R.P.: A linear algorithm for data compression. Australian Computer Journal 19(2), 64–68 (1987)
6. Devroye, L., Szpankowski, W.: Probabilistic behavior of asymmetric level compressed tries. Random Structures and Algorithms 27, 185–200 (2005)
7. Fredkin, E.: Trie memory. Communications of the ACM 3, 490–500 (1960)
8. Geary, R.F., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 159–172. Springer, Heidelberg (2004)
9. Hagerup, T.: Sorting and searching on the word ram. In: Proceedings of Symposium on Theory Aspects of Computer Science, pp. 366–398 (1998)
10. Hon, W.-K., Lam, T.-W., Sadakane, K., Sung, W.-K.: Constructing compressed suffix arrays with large alphabets. In: Ibaraki, T., Katoh, N., Ono, H. (eds.) ISAAC 2003. LNCS, vol. 2906, Springer, Heidelberg (2003)
11. Morrison, D.R.: PATRICIA - Practical Algorithm To Retrieve Information Coded In Alphanumeric. Journal of the ACM 15(4), 514–534 (1968)
12. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing 31(3), 762–776 (2001)
13. Navarro, G.: Indexing text using the Ziv-Lempel trie. Journal of Discrete Algorithmcs (JDA) 2(1), 87–114 (2004)
14. Nilsson, S., Karlsson, G.: IP-address lookup using lc-tries. Journal on Selected Areas in Communications IEEE 17(6), 1083–1092 (1999)
15. Rodeh, M., Pratt, V.R., Even, S.: Linear algorithm for data compression via string matching. Journal of ACM 28(1), 16–24 (1981)
16. Sadakane, K., Grossi, R.: Squeezing Succinct Data Structures into Entropy Bounds. In: Proc. ACM-SIAM SODA, pp. 1230–1239. ACM Press, New York (2006)
17. Welch, T.A.: A technique for high-performance data compression. IEEE Computer, 8–19 (1984)
18. Willard, D.E.: Log-logarithmic worst case range queries are possible in space $\theta(n)$. Information Processing Letters 17, 81–84 (1983)
19. Willard, D.E.: New trie data structures which support very fast search operations. Journal of Computer and System Sciences 28, 379–394 (1984)
20. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory IT-24(5), 530–536 (1978)