Algorithmica (2015) 71:969–988
DOI 10.1007/s00453-013-9836-6

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

Jesper Jansson · Kunihiko Sadakane · Wing-Kin Sung

Received: 22 June 2009 / Accepted: 11 September 2013 / Published online: 26 September 2013

Abstract The *dynamic trie* is a fundamental data structure with applications in many areas of computer science. This paper proposes a new technique for maintaining a dynamic trie T of size at most 2^w nodes under the unit-cost RAM model with a fixed word size w. It is based on the idea of partitioning T into a set of linked small tries, each of which can be maintained efficiently. Our method is not only space-efficient, but also allows the longest common prefix between any query pattern P and the

Parts of the results presented in this paper appeared in preliminary form in *Proceedings of the 27th Annual International Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007)*, volume 4855 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 424–435, 2007.

J.J. was funded by the Hakubi Project at Kyoto University and the Japan Society for the Promotion of Science (JSPS).
K.S. was partially supported by the Japan Society for the Promotion of Science (JSPS) through the "Funding Program for World-Leading Innovative R&D on Science and Technology (FIRST Program)".
W.-K.S. was partially supported by the MOE AcRF Tier 2 funding R-252-000-444-112.

J. Jansson (✉)
Laboratory of Mathematical Bioinformatics (Akutsu Laboratory), Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan
e-mail: jj@kuicr.kyoto-u.ac.jp

K. Sadakane
National Institute of Informatics (NII), 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
e-mail: sada@nii.ac.jp

W.-K. Sung
School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417, Singapore
e-mail: ksung@comp.nus.edu.sg

W.-K. Sung
Genome Institute of Singapore, 60 Biopolis Street, Genome, Singapore 138672, Singapore

strings currently stored in T to be computed in $o(|P|)$ time for small alphabets, and allows any leaf to be inserted into or deleted from T in $o(\log|T|)$ time. To demonstrate the usefulness of our new data structure, we apply it to LZ-compression. Significantly, we obtain the first algorithm for generating the LZ78 encoding of a given string of length n over an alphabet of size σ in sublinear ($o(n)$) time and sublinear ($o(n\log\sigma)$ bits) working space for small alphabets ($\sigma = 2^{o(\log n \frac{\log\log\log n}{(\log\log n)^2})}$). Moreover, the working space for our new algorithm is asymptotically less than or equal to the space for storing the output compressed text, regardless of the alphabet size.

Keywords Data structures · Dynamic trie · Longest common prefix query · LZ-compression

1 Introduction

A *trie* [10] 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 labeled by a from u to a child of u. Each node u in a trie thus represents a string obtained by concatenating the symbols on the unique path from the root to u; consequently, a trie can be used to store a set of strings over $\mathcal{A}$. Without loss of generality, we assume that the alphabet $\mathcal{A}$ includes a special end-of-string symbol \$, that every string over $\mathcal{A}$ terminates with \$, and that \$ only appears at the end of a string. In this way, any set of strings over $\mathcal{A}$ corresponds to the leaves of a trie.

A *dynamic trie* is a trie which supports operations that modify it online. Typically, these operations allow a set of strings to be maintained dynamically so that strings may be inserted into or deleted from it. Dynamic tries find applications in information retrieval, natural language processing, database systems, compilers, data compression, and computer networks. For example, in computer networks, dynamic tries are used for IP routing to efficiently maintain the hierarchical organization of routing information to enable fast lookup of IP addresses [22]. (Since IP addresses are of 32 bits or 128 bits, it is not practical to use direct lookup tables for routing.) 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 of the Ziv-Lempel encoding scheme (LZ78) [29] (or its variant LZW encoding [26]) and the Huffman encoding scheme, respectively. In addition, other useful data structures such as the suffix trie/suffix tree, the Patricia trie [19], and the associative array (hashing table) can be maintained by dynamic tries.

From here on, we assume that $\mathcal{A}$ is an ordered alphabet and define $\sigma = |\mathcal{A}|$. This paper assumes the unit-cost RAM model with word size w in which standard arithmetic and bitwise boolean operations on word-sized operands take constant time (see, e.g., [13]). More precisely, each operation in our algorithms can be performed in constant time by using lookup tables of $O(2^{\epsilon w})$ bits for some fixed $0 < \epsilon < 1$. Note that we use arithmetic operations for only $O(\log n)$ bit numbers where n is the length of the input, and therefore the lookup table size can be reduced from $O(2^{\epsilon w})$ to $o(n)$. For the memory model, we assume that any memory region of $\Theta(w)$ bits can be allocated and freed in constant time as in [24]. Also, our algorithms require a query pattern P to be packed in $O(\frac{|P|\log\sigma}{w})$ words.[1] Under this model, we propose a new, compact method for representing a dynamic trie T containing at most 2^w nodes so that: (1) any leaf can be inserted into or deleted from T efficiently; and (2) the longest common prefix between any query pattern P and T, i.e., the longest prefix of P which is identical to a prefix of a string stored in T, can be obtained efficiently.

1.1 Previous Work

By using standard tree data structures, a dynamic trie T can be implemented in $O(|T|\cdot(\log|T|+\log\sigma))$ bits of space so that: (1) insertion or deletion of a leaf takes $O(\log\sigma)$ time; and (2) answering the longest common prefix query takes $O(|P|\cdot\log\sigma)$ time.

A number of solutions have been presented to improve the space and average time complexities. Morrison [19] proposed the *Patricia trie* which compresses paths by contracting nodes having only one child, thereby reducing the total size of the trie. Andersson and Nilsson [1] proposed the *LC-trie* which decreases the height of the trie by increasing the branching factor (so-called *level compression*); this idea reduces the average running time [7]. The *String B-tree* [8] reduces the search time to $O(|P|+\log|T|)$ time. However, in the worst case, the above solutions use $O(|T|)$ words space and still require $O(|P|)$ time to answer a longest common prefix query. Darragh *et al.* [6] proposed a method named *Bonsai* which is based on hashing and uses $O(|T|\log\sigma)$ space, but requires $O(|P|)$ time (or $O(|P|\log\sigma)$ time, depending on how child nodes are represented) to answer a longest common prefix query. Employing the *succinct dynamic binary tree* [24], a dynamic trie can be simulated in $O(|T|\log\sigma)$ bits of space, by considering each character in the alphabet as a binary string of $\log\sigma$ bits, under the unit-cost RAM model so that: (1) insertion or deletion of a leaf takes $O(\log\sigma(\log\log|T|)^{1+\epsilon})$ time where $\epsilon>0$; and (2) the longest common prefix query takes $O(|P|\log\sigma)$ time. Arroyuelo [2] gave a data structure for representing a tree T using $|T|\cdot(2+\log\sigma)+o(|T|\log\sigma)$ bits supporting the parent and child operations in $O(\log\sigma+\log\log|T|)$ time, and insertion and deletion of nodes in $O((\log\sigma+\log\log|T|)\cdot(1+\frac{\log\sigma}{\log(\log\sigma+\log\log|T|)}))$ amortized time. In summary, none of the existing data structures can answer the longest common prefix query in $o(|P|)$ time.

We also remark that Willard [27, 28] proposed two well-known data structures for manipulating a trie T of fixed height $O(\log M)$ for a positive integer M: the *q-fast trie* [28], which takes $O(|T|\log M)$ bits space and searches for any pattern P over the binary alphabet in T using $O(\sqrt{\log M})$ time while inserting or deleting a leaf in $O(\sqrt{\log M})$ time, and *the y-fast trie* [27], which is a static trie that uses $O(|T|\log M)$ bits of space and can report the longest prefix of any pattern P over the binary alphabet in T using $O(\log\log M)$ time.

[1] In this paper, log denotes the base-2 logarithm, ln is the natural logarithm, and $\log_\sigma$ is the base-σ logarithm. Furthermore, $\lceil$ and $\rceil$ symbols have been omitted to increase the readability.

1.2 New Results

We present a new data structure for maintaining a dynamic trie T of size[2] at most 2^w. It uses $O(|T|\log\sigma)$ bits while: (1) insertion or deletion of any leaf takes $O((\log\log|T|)^2/\log\log\log|T|)$ worst-case time; and (2) computing the longest common prefix between a query pattern P and T takes $O(\frac{|P|}{\log_\sigma|T|}\frac{(\log\log|T|)^2}{\log\log\log|T|})$ worst-case time [Theorem 1]. In other words, each insertion or deletion of a leaf takes $o(\log|T|)$ time, and furthermore, for $\sigma = 2^{o(\log|T|\frac{\log\log\log|T|}{(\log\log|T|)^2})}$, our dynamic trie data structure can perform the longest common prefix query in $o(|P|)$ time. When considering the expected amortized time complexity instead of the worst-case time complexity for updates, we further improve the dynamic trie data structure so that: (1) insertion or deletion of a leaf takes $O(\log\log|T|)$ expected amortized time; and (2) the longest common prefix query takes $O(\frac{|P|}{\log_\sigma|T|}+\log\log|T|)$ worst-case time [Theorem 2].

Our improvements to existing data structures for dynamic tries stem from the observation that small tries (that is, tries of size $O(\log_\sigma|T|)$) can be maintained very efficiently. Accordingly, our new data structure partitions the trie T into many small tries which are linked together, and maintains the small tries individually. With this approach, we not only store the trie compactly using $O(|T|\log\sigma)$ bits, but also enable fast queries and efficient insertions and deletions.

1.3 Applications to LZ-Compression

To demonstrate the new dynamic trie data structure, we apply it to generate the Ziv-Lempel encoding [29] of a string. *The Ziv-Lempel encoding scheme* (LZ78) and its variant *LZW encoding* [26] are popular text compression schemes. Current solutions for generating the LZ78 encoding of a text of length n over an alphabet of size σ construct a trie called the LZ-trie from which the LZ78 encoding is directly obtained; see Sect. 2.3 below for details. LZ78 also finds applications in compressed indexing; Navarro [21] presented a compressed full-text self-index based on the LZ-trie called *LZ-index* whose space usage is proportional to that of the compressed text and which allows efficient exact pattern matching. There also exist indexing data structures based on the LZ-trie [9, 15].

People have recently realized the importance of workspace-efficient data compression algorithms [3, 14, 17]. Given a long text, there may be enough memory to store the compressed text (for example, its LZ78 encoding), but in some cases it is difficult to actually construct the compressed text because the working space requirement becomes too large. For example, we may be able to store the LZ78 encoding of the human genome in a 1GB RAM computer, but we may fail to construct the encoding due to limitations in the size of the memory [3]. Hence, workspace-efficient encoding algorithms are necessary.

[2]We use the term *size of a trie* to refer to the number of nodes in the trie, not the number of bits used to represent the trie.

However, none of the solutions in the literature for constructing the LZ78 encoding run in $o(n)$ time and $o(n \log \sigma)$ bits of working space. A straightforward implementation of LZ78 based on Ziv and Lempel's original definition takes $O(n \log \sigma)$ worst-case time to process a string of length n using trie data structures (if hashing is used to store the children of a node, this may be reduced to $O(n)$ expected time). Moreover, such an algorithm uses $O(n)$ words of working space and this may be much larger than the space usage of the LZ78 encoding, which is $(1 + o(1))nH_k$ bits, where H_k denotes the k-th order empirical entropy [16] of the text. Utilizing the solution of Arroyuelo and Navarro [3], the LZ78 encoding of a text can be constructed using $O(n(\log \sigma + \log \log n))$ time and $(1 + \epsilon)nH_k + O(\frac{n(k \log \sigma + \log \log_\sigma n)}{\log_\sigma n})$ bits of working space for any non-negative integer k and $\epsilon > 0$. For $k = o(\log_\sigma n)$, the working space of the method of Arroyuelo and Navarro [3] is $(1 + \epsilon)nH_k + o(n) \log \sigma$ bits, but if $H_k = \Theta(\log \sigma)$ then the working space is $\Theta(n \log \sigma)$ bits.

By employing our new dynamic trie data structure to maintain the LZ-trie and using a two-phase technique, we obtain the first sublinear ($o(n)$) time and sublinear ($o(n \log \sigma)$ bits) working space algorithm for generating the LZ78 encoding of a string over a small alphabet. More precisely, our proposed algorithm runs in $o(n)$ time and $o(n \log \sigma)$ bits of working space whenever $\sigma = 2^{o(\log n \frac{\log \log \log n}{(\log \log n)^2})}$. To be even more precise, it uses $O(\frac{n(\log \sigma + \log \log_\sigma n)}{\log_\sigma n})$ bits of working space and runs in either $O(\frac{n}{\log_\sigma n} \frac{(\log \log n)^2}{\log \log \log n})$ worst-case time [Theorem 3] or $O(\frac{n \log \log n}{\log_\sigma n})$ expected time [Theorem 4] for any $\sigma \leq n$. Also, for any $\sigma \leq n$, the working space of our new algorithm is asymptotically less than or equal to that of the output compressed text, as discussed at the end of Sect. 5.2.

1.4 Organization of Paper

The rest of the paper is organized as follows. Section 2 describes a few data structures needed by our new dynamic trie data structure and defines the LZ78 encoding scheme. Sections 3 and 4 detail our new linked dynamic trie data structure with good worst-case update time and good expected amortized update time, respectively. Finally, Sect. 5 presents our LZ78 encoding algorithm and Sect. 6 contains some concluding remarks.

2 Preliminaries

Some data structures for dynamically maintaining a set of strings and an edge-labeled tree efficiently, which form the building blocks of our new dynamic trie data structure, are described in Sects. 2.1 and 2.2, respectively. Section 2.3 reviews the definitions of the LZ78 encoding scheme and the LZ-trie.

2.1 Data Structures for Maintaining a Set of Strings of Length at most $\log_\sigma N$

This subsection recalls two variants of the *dynamic predecessor data structure* [4, 18] that can be used to maintain a set of strings, each of length at most $\log_\sigma N$ where

N is a fixed non-negative integer, over an alphabet of size σ while supporting three operations:

- *Insertion* of a string of length at most $\log_\sigma N$.
- *Deletion* of a string of length at most $\log_\sigma N$.
- *Predecessor* (or *successor*) of a string P, i.e., reporting the string currently in the set which is lexicographically just smaller (or bigger) than P.

We represent any string of length at most $\log_\sigma N$ by an integer in $\{0, 1, \dots, N-1\}$. If a string is shorter than $\log_\sigma N$, we append \$-symbols to the end of the string to make its length equal to $\log_\sigma N$.

First, for obtaining a good worst-case running time, we use the dynamic predecessor data structure of Beame and Fich [4], whose properties are summarized in the next lemma:

Lemma 1 *The dynamic predecessor data structure of Beame and Fich* [4] *can maintain a set of ℓ $O(\log N)$-bit integers using $O(\ell \log N)$ bits of space under insertions and deletions while also supporting predecessor* (*or successor*) *queries in such a way that each operation takes $O((\log\log N)^2/(\log\log\log N))$ time.*

We immediately obtain:

Lemma 2 *Consider ℓ strings of length at most $\log_\sigma N$ over an alphabet of size σ. We can store the strings in $O(\ell \log N)$ bits so that each of the insert/delete/predecessor operations can be performed in $O((\log\log N)^2/\log\log\log N)$ time.*

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

As for the expected amortized running time, using a data structure of Mehlhorn and Näher [18] leads to an improvement in the time complexity:

Lemma 3 *The dynamic predecessor data structure of Mehlhorn and Näher* [18] *can maintain a set of ℓ $O(\log N)$-bit integers using $O(\ell \log N)$ bits of space in such a way that a predecessor query takes $O(\log\log N)$ time and the insertion or deletion of an integer takes $O(\log\log N)$ expected amortized time.*

Lemma 4 *Consider ℓ strings of length at most $\log_\sigma N$ over an alphabet of size σ. We can store the strings in $O(\ell \log N)$ bits so that insertion or deletion of a string takes $O(\log\log N)$ expected amortized time and a predecessor query can be answered in $O(\log\log N)$ time.*

Proof Treat the strings as integers in the range $\{0, 1, \dots, N-1\}$ and apply Lemma 3. □

2.2 Data Structures for Maintaining an Edge-Labeled Tree

Here, we show how to dynamically maintain an edge-labeled tree T. We assume that the size of the tree is at most N for some fixed non-negative integer N, that the nodes are numbered by integers in $[0, N-1]$, and that all edge labels are integers in $[0, N-1]$. In addition, we assume that for each node u in T, all child edges of u have distinct labels. We consider the following operations:

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

Note that a node u is a leaf if *Child*(u, κ) does not exist for any $\kappa \in \mathcal{A}$.

Lemma 5 *An edge-labeled tree T can be maintained dynamically in $O(|T| \log N)$ bits of space in such a way that each Child, Insert, and Delete operation can be supported in $O((\log\log N)^2/(\log\log\log N))$ time.*

Proof Use two dynamic predecessor data structures $\mathcal{D}_1$ and $\mathcal{D}_2$ from Lemma 1 to represent T as follows. For each edge (u, v) labeled by κ, store the integer $N^2 \cdot u + N \cdot \kappa + v$ in $\mathcal{D}_1$ and the integer $N^2 \cdot v + N \cdot u + \kappa$ in $\mathcal{D}_2$. $\mathcal{D}_1$ and $\mathcal{D}_2$ take $O(|T| \log N)$ bits of space. Since $0 \leq u, v, \kappa \leq N$, there is a one-to-one mapping between (u, v, κ) and the integer $x = N^2 \cdot u + N \cdot \kappa + v$ in $\mathcal{D}_1$. To be precise, $v = x \bmod N$, $u = \lfloor x/N^2 \rfloor$, and $\kappa = \lfloor (x - u \cdot N^2)/N \rfloor$. Similarly for $\mathcal{D}_2$.

To insert a leaf node v, which is a child of u with edge label κ, insert $N^2 \cdot u + N \cdot \kappa + v$ into $\mathcal{D}_1$ and insert $N^2 \cdot v + N \cdot u + \kappa$ into $\mathcal{D}_2$.

To delete a leaf node v, first retrieve the integer x which is just larger than $N^2 \cdot v - 1$ (or the successor of $N^2 \cdot v - 1$) in $\mathcal{D}_2$. Note that $x = N^2 \cdot v + N \cdot u + \kappa$, where u is the parent of v and κ is the label of (u, v); therefore, the leaf node v 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, κ), first retrieve the integer x which is just larger than $N^2 \cdot u + N \cdot \kappa - 1$ (or the successor of $N^2 \cdot u + N \cdot \kappa - 1$) in $\mathcal{D}_1$. Then, if $x < N^2 \cdot u + N \cdot (\kappa + 1)$, *Child*$(u, \kappa)$ exists and it equals the remainder when x is divided by N, otherwise it does not exist.

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

Alternatively, we can make use of a data structure known as the *dynamic dictionary* [5, 24], whose properties are summarized below.

Lemma 6 *The dynamic dictionary data structure of Blandford and Blelloch* [5] *or Raman and Rao* [24] *can maintain a set of ℓ key-data integer pairs (x, y), where both x and y are $O(\log N)$-bit integers, using $O(\ell \log N)$ bits under insertions and deletions while supporting membership queries so that retrieving data of a corresponding key takes $O(1)$ time and each insert/deletion operation takes $O(1)$ expected amortized time.*

Equipped with Lemma 6, we obtain:

Lemma 7 *An edge-labeled tree T can be maintained dynamically in $O(|T| \log N)$ bits of space so that the Child operation takes $O(1)$ time and each of the operations Insert and Delete can be supported in $O(1)$ expected amortized time.*

Proof Maintain T with a dynamic dictionary $\mathcal{D}$ from Lemma 6. For each edge (u, v) labeled by κ, store two key-data pairs $(N \cdot u + \kappa,\ v)$ and $(v,\ N \cdot u + \kappa)$ in $\mathcal{D}$.

To insert a leaf node v which is a child of u with edge label κ, insert the two key-data pairs $(v,\ N \cdot u + \kappa)$ and $(N \cdot u + \kappa,\ v)$ into $\mathcal{D}$. To delete a leaf node v, first retrieve the data x associated with the key v. Here, $x = N \cdot u + \kappa$, where u is the parent of v and κ is the label of (u, v). Hence, by deleting the two key-data pairs (v, x) and (x, v) from $\mathcal{D}$, the leaf node v is removed. Both of the insert and delete operations take $O(1)$ expected amortized time.

The *Child*(u, κ) operation is implemented by returning the data associated with the key $N \cdot u + \kappa$ from $\mathcal{D}$, which takes $O(1)$ worst-case time. If no such key exists in $\mathcal{D}$ then *Child*(u, κ) does not exist. $\square$

2.3 LZ78 Encoding and the LZ-Trie

The *Ziv-Lempel encoding scheme* [29], or LZ78, is a data compression scheme for strings. For a given string $S = S[1 \dots n]$, it partitions S into substrings called *phrases* and a constructs a so-called LZ-*trie* procedurally using the following method: First, initialize a trie T consisting of a single node (the root) corresponding to the empty string, the current position p in S as $p = 1$, and the number of phrases $c = 0$. Next, parse S from left to right until $p > n$ as follows. Find the longest string, $t \in T$, that appears as a prefix of $S[p \dots n]$. Set $c = c + 1$. Define the phrase $s_c = S[p \dots (p + |t|)] = t \cdot S[p + |t|]$ and insert s_c into T. Then, let $p = p + |t| + 1$ 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 $s_1, s_2, \dots, s_c$ is called the *phrase list*. Note that T has $c + 1$ nodes. The output of the LZ78 encoding scheme of the given string S is called *the* LZ78 *encoding of S* and consists of a list of c pairs of the form $(id_i, char_i)$, where the ith pair represents phrase s_i and where id_i is the id of the parent of the node which encodes s_i in the LZ-trie and $char_i$ is the last symbol of s_i. Clearly, the number of bits needed to represent the LZ78 encoding of S, i.e., the space usage of the compressed data, is $c(\log c + \log \sigma)$. (The LZ-trie itself is not part of the output.) Ziv and Lempel [29] proved that the LZ78 encoding scheme achieves an asymptotically optimal compression ratio.

By [29], it holds that $\sqrt{n} \le c \le n / \log_\sigma n$. Also, the LZ-trie can be stored in $c \log c + \Theta(c \log \sigma)$ bits. For any integer $k = o(\log_\sigma n)$, this is $nH_k + O(n \log \sigma \log \log_\sigma n / \log_\sigma n) = nH_k + o(n \log \sigma)$ bits [16].

3 Dynamically Maintaining a Trie with Good Worst-Case Update Time

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

- *Insert_leaf*(T, u, a): Inserts a leaf v as a child of u such that the label of (u, v) is a, where $a \in \mathcal{A}$.
- *Delete_leaf*(T, u): Deletes the leaf u and the edge between u and its parent (if any).
- *Lcp*(T, P): Reports the length ℓ such that $P[1 \dots \ell]$ is the longest prefix of a string which exists in T.

Let w be the (fixed) word size of the assumed RAM model, and define $N = 2^w$. Below, we first show how to maintain a trie of size $O(\log_\sigma N)$. Then, we focus on how to maintain a trie of height at most $O(\log_\sigma N)$. Finally, we show how to maintain a general trie with at most N nodes by partitioning it into a set of linked tries, each of height at most $O(\log_\sigma N)$.

3.1 Maintaining a Trie of Size $O(\log_\sigma N)$

We now explain how to dynamically maintain a trie T of size $O(\log_\sigma N)$, in which every leaf represents a string. In addition to the operations *Lcp*, *Insert_leaf*, and *Delete_leaf*, the dynamic trie also supports computing the preorder of any node in a trie, the splitting of a trie, and the merging of two tries. For the preorder operation, we report the preorder number of any node in a trie. For the split operation, a trie T is split into two subtries T_1 and T_2 such that T_1 stores the s lexicographically smallest strings for some integer s while T_2 stores the rest of the strings in T. For the merge operation, two tries T_1 and T_2 are merged into one trie where the leaves in T_1 are smaller than those in T_2.

Lemma 8 *We can precompute six tables of size $O(N^{5\epsilon})$ bits, for any fixed constant $0 < \epsilon < 0.2$. Based on these tables, we can maintain a trie T of size at most $\epsilon \log_\sigma N$ by using at most $3\epsilon \log N$ bits so that each of the operations Lcp, Insert_leaf, and Delete_leaf takes $O(1)$ time, and also the preorder of any node can be computed in $O(1)$ time. In addition, we can also split a trie of size at most $\epsilon \log_\sigma N$ or merge two tries of size at most $\frac{\epsilon}{2} \log_\sigma N$ in $O(1)$ time.*

Proof The data structure has two parts. First, the topology of T is stored in $2|T| \le 2\epsilon \log_\sigma N$ bits using parenthesis encoding [12, 20]. Second, the edge labels of all edges are stored in preorder using $|T| \log \sigma \le \epsilon \log N$ bits. Therefore, the total space is at most $3\epsilon \log N$ bits.

In addition, the data structure requires six precomputed tables. The first table *Lcp_STrie* 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_STrie*(R, Q), which is the preorder of any string Q of length at most $\epsilon \log_\sigma N$ in any trie R of size at most $\epsilon \log_\sigma N$. Since there are $O(2^{3 \cdot \epsilon \log N} \cdot \sigma^{\epsilon \log_\sigma N}) = O(N^{4\epsilon})$ different combinations of R and Q, both tables can be stored in $O(N^{4\epsilon} \cdot \log \log_\sigma N) = O(N^{5\epsilon})$ bits of space.

The third table stores the value of *Ins_STrie*(R, x, c), which is the trie formed by inserting character c to the leaf x of the trie R of size at most $\epsilon \log_\sigma N - 1$. The table *Ins_STrie* has $O(2^{3\epsilon \log N} \cdot \sigma^{(\epsilon \log_\sigma N)-1} \cdot \sigma) = O(N^{4\epsilon})$ entries and each entry requires $O(\epsilon \log N)$ bits, so the third table uses $O(N^{5\epsilon})$ bits of space. The fourth table stores the value of *Del_STrie*(R, x), which is the trie formed by deleting the leaf x from the trie R of size at most $\epsilon \log_\sigma N$. The table *Del_STrie* has $O(2^{3\cdot\epsilon \log N} \cdot \epsilon \log_\sigma N)$ entries and each entry requires $O(\epsilon \log N)$ bits, so the fourth table uses $O(N^{4\epsilon})$ bits of space.

The fifth table stores the value of *Split_STrie*(R, s), which is a pair of tries (R_1, R_2) where R_1 is of size s and R_2 is of size $|R| - s$ such that R_1 and R_2 split R of size at most $\epsilon \log_\sigma N$. Since there are $O(2^{3\cdot\epsilon \log N} \cdot \epsilon \log_\sigma N)$ entries and each entry takes $O(\epsilon \log_\sigma N)$ bits, this table can be stored in $O(N^{4\epsilon})$ bits of space. The sixth table stores the value of *Merge_STrie*(R_1, R_2) which is the trie formed by merging R_1 and R_2, where the total size of R_1 and R_2 is $\frac{\epsilon}{2} \log_\sigma N$. There are $O(2^{3\epsilon \log N} \cdot \epsilon \log_\sigma N)$ different combinations of (R_1, R_2). For each combination, the merged trie can be stored in $O(\epsilon \log_\sigma N)$ bits. Hence, this table can be stored in $O(N^{4\epsilon})$ bits of space.

Then, the six operations can be supported in $O(1)$ time by using the precomputed tables:

- To insert/delete a node x, we update the topology and the edge label in $O(1)$ time utilizing the tables *Ins_STrie* and *Del_STrie*.
- *Lcp*(T, P) can be computed in $O(1)$ time by utilizing the table *Lcp_STrie*.
- Preorder of any string in T can also be computed in $O(1)$ time utilizing the table *preorder_STrie*.
- We can split/merge tries in $O(1)$ time by using the tables *Split_STrie* and *Merge_STrie*.

□

Lemma 9 *For any* $0 \le \epsilon \le 0.2$ *and a fixed* N*, the six tables needed by Lemma* 8 *can be precomputed using* $O(N^{4\epsilon} \log_\sigma N)$ *time.*

Proof Observe that every table has $O(N^{4\epsilon})$ entries. Furthermore, each entry can be computed in $O(\epsilon \log_\sigma N)$ time. Hence, the lemma follows. □

3.2 Maintaining a Trie of Height $O(\log_\sigma N)$

This subsection describes how to dynamically maintain a trie of height $O(\log_\sigma N)$, given the precomputed tables from Lemma 8.

Lemma 10 *Given the six precomputed tables in Lemma* 8 *of* $O(N^{5\epsilon})$ *bits for any constant* $0 < \epsilon < 0.2$*, we can dynamically maintain a trie* T *of height at most* $\frac{\epsilon}{2} \log_\sigma N$ *using* $O(\epsilon^{-1}|T| \log \sigma)$ *bits of space such that all operations Lcp, Insert_leaf, and Delete_leaf 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.

Observe 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 $i \le j \le |T|$. Note that all strings in a block can be represented as a subtrie of T. The nodes $u_i, u_{i+1}, \ldots, u_j$ may not be connected.

In this case we include the nodes on the path from the root to u_i to make the nodes connected. 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 $B_1 \cup B_2 \cup \cdots \cup B_{|\mathcal{B}|} = S$. 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 $\frac{\epsilon}{4}\log_\sigma N$ strings. From here on, for each $B_i \in \mathcal{B}$, let $s_{b(i)}$ be the lexicographically smallest string in B_i.

Now, the trie T is represented by a two-level data structure:

- Top-level: Using the data structure in Lemma 2, store $\{s_{b(1)}, \ldots, s_{b(|\mathcal{B}|)}\}$.
- Block-level: For each block $B_i \in \mathcal{B}$, represent the strings in B_i as a trie T_i of size at most $\epsilon \log_\sigma N$ and store the trie using Lemma 8. Also, store a bit vector $V_i[1 \ldots \epsilon \log_\sigma N]$ such that $V_i[j] = 1$ if the node of preorder j in T_i represents a string in B_i.

We first show that the space required is $O(\epsilon^{-1}|T|\log\sigma)$ bits. Observe that $|\mathcal{B}| = O(\frac{|T|}{\epsilon \log_\sigma N})$ blocks. The space needed to store the top-level structure is $O(|\mathcal{B}|\log N) = O(\epsilon^{-1}|T|\log\sigma)$ bits. Each block requires $O(\epsilon \log N)$ bits of space by Lemma 8. The space for the block-level structure is $O(|\mathcal{B}|\epsilon \log N) = O(|T|\log\sigma)$.

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 and T, we 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. Let lcp_1 and lcp_2 be the lengths of the longest common prefixes of P' in the trie for B_j and B_{j+1} respectively. By Lemma 8, lcp_1 and lcp_2 can be computed in $O(1)$ time. Then, $Lcp(T, P) = \max\{lcp_1, lcp_2\}$.

– *Insert_leaf*(T, u, a): Suppose u represents a string $s \in S$. This operation is equivalent to inserting a new string $s \cdot a$ after s. Let B_j be the block containing s. We insert $s \cdot a$ into B_j using $O(1)$ time by Lemma 8, and update V_j accordingly. 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 bit vector V_j is used to recover the set of strings in B_j. The split takes $O(1)$ time by Lemma 8 because B_j corresponds to a trie of size $\frac{\epsilon}{2}\log_\sigma N$. 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_leaf*(T, u): The analysis is similar to the *Insert_leaf* operation. If B_j contains at least $\frac{\epsilon}{4}\log_\sigma N$ strings after deletion, we are done. Otherwise, if a neighbor B_{j-1} or B_{j+1} contains more than $\frac{\epsilon}{4}\log_\sigma N$ strings, we move a string from the neighbor to B_j. If both B_{j-1} and B_{j+1} contain exactly $\frac{\epsilon}{4}\log_\sigma N$ strings, we merge B_j with one of them. We also update the top-level structure.

□

3.3 Maintaining a Trie with no Height Restrictions

This subsection gives a data structure for dynamically maintaining a general trie T. Assuming that $|T| \le N$ and that the six precomputed tables in Lemma 8 are available,

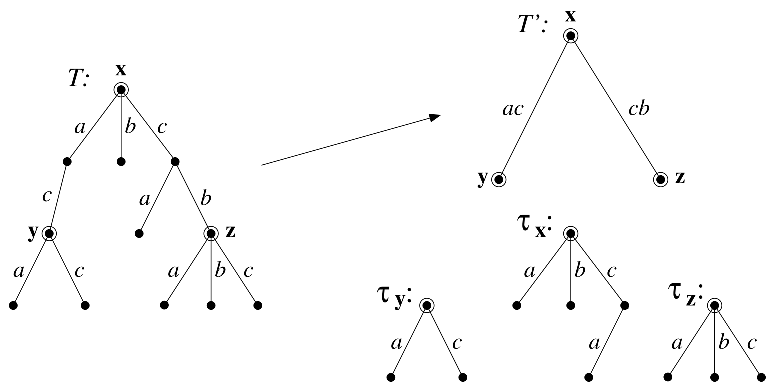

Fig. 1 Assuming $\delta = 2$, T is represented by storing T' and τ_x, τ_y, τ_z

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 the 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.

The data structure represents T by partitioning it into smaller tries of height at most $h = \frac{\epsilon}{2}\log_\sigma N$ which are linked together. To formally describe the representation, we need some definitions.

Set $\delta = \lceil h/3 \rceil$. Any node $u \in T$ is called a *linking node* if: (1) the distance from the root to u is a multiple of δ; and (2) the subtrie rooted at u has more than δ nodes.

Let LN be the set of linking nodes of T. For every non-root node $v \in T$, we denote by $p(v)$ the linking node in LN which is the lowest proper ancestor of v in T. For any $u \in LN$, let τ_u be the subtrie of T rooted at u including all descendents v of u such that $p(v) = u$ and $v \notin LN$. Next, let T' be a tree whose vertex set is LN and whose edge set is $\{(p(u), u) \mid u \in LN \text{ and } u \text{ is not the root}\}$. The label of each edge $(p(u), u)$ in T' is the string of length δ represented by the path from $p(u)$ to u in T. We remark that T' corresponds to a "macroset" and each τ_u to a "microset" in [11].

Based on the above discussion, T can be represented by storing T' along with τ_u for all $u \in LN$. (See Fig. 1 for an example.) The next lemma bounds the space usage of LN.

Lemma 11 $|LN| \le |T|/\delta$. *Moreover, for every* $u \in LN$, τ_u *is of height smaller than* 2δ.

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

The lemma below states how to maintain a trie of size at most N.

Lemma 12 *For any* $0 < \epsilon < 0.2$ *and* $N \le 2^w$, *we can dynamically maintain a trie* T *of size at most* N *using* $O(\epsilon^{-1}|T|\log\sigma + N^{5\epsilon})$ *bits so that Lcp*(T, P) *takes*

$O(\frac{|P|}{\epsilon \log_\sigma N} \frac{(\log\log N)^2}{\log\log\log N})$ time while insertion/deletion of a leaf takes $O((\log\log N)^2/\log\log\log N)$ *time.*

Proof Store the precomputed tables from Lemma 8 in $O(N^{5\epsilon})$ bits of space and use Lemma 5 to represent T' in $O(|T'|\log N) = O(|LN|\log N) = O(\frac{|T|}{\delta}\log N) = O(\frac{|T|}{\epsilon \log_\sigma N}\log N) = O(\epsilon^{-1}|T|\log\sigma)$ bits. For every $u \in LN$, the height of τ_u is bounded according to Lemma 11, so we can represent τ_u as in Lemma 10 using $O(|\tau_u|\log\sigma)$ bits. Since $\sum_{u\in LN}|\tau_u| = |T|$, all τ_u's can be represented in $O(|T|\log\sigma)$ bits. We also store the pointer to τ_u for each $u \in LN$ in $O(\log N)$ bits. The total space is also $O(|T'|\log N)$ bits.

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 $O(\frac{|P|}{\epsilon \log_\sigma N} \frac{(\log\log N)^2}{\log\log\log N})$ time using the child operation of the data structure in Lemma 5. Suppose u is the node in T' corresponding to the longest prefix $P[1\dots x]$ of P ending in a node. Second, we find the longest prefix of $P[x+1\dots|P|]$ in τ_u. By Lemma 10, it takes an additional $O(\frac{(\log\log N)^2}{\log\log\log N})$ time.

For *Insert_leaf*(T, u, a), suppose u is in the subtrie τ_v where $v \in LN$. By Lemma 10, it takes $O(\frac{(\log\log N)^2}{\log\log\log N})$ time to insert a as a child edge of u. Moreover, if the insertion creates a new linking node v' in τ_v, we need to do the following additional steps: (1) insert a new leaf in T' corresponding to v' (this step can be performed in $O(\frac{(\log\log N)^2}{\log\log\log N})$ time by Lemma 5); and (2) split τ_v into $\tau_{v'}$ and $\tau_v - \tau_{v'}$ (this step can be done in $O(1)$ time using the tables *Split_STrie* and *Merge_STrie* in Lemma 8).

Delete_leaf(T, u) can be computed in $O(\frac{(\log\log N)^2}{\log\log\log N})$ time, similarly to the insertion operation. □

The previous lemma stores a trie T using $O(\epsilon^{-1}|T|\log\sigma + N^{5\epsilon})$ bits of space. The proof of the next theorem explains how to eliminate the $N^{5\epsilon}$-term from the space complexity.

Theorem 1 *For any fixed constant* $0 < \epsilon < 0.2$, *we can dynamically maintain a trie* T *using* $O(\epsilon^{-1}|T|\log\sigma)$ *bits of space such that the query Lcp*(T, P) *takes* $O(\frac{|P|}{\epsilon \log_\sigma |T|} \frac{(\log\log|T|)^2}{\log\log\log|T|})$ *time while insertion or deletion of a leaf takes* $O((\log\log|T|)^2/\log\log\log|T|)$ *time.*

Proof We use Overmars' global rebuilding technique [23]. Our data structure keeps two versions of the trie and maintains four invariants:

(1) Suppose $2^{\ell-2} < |T| \le 2^{\ell-1}$, the two versions of the trie are implemented based on Lemma 12 using $N = 2^\ell$ and $N = 2^{\ell+1}$, respectively.
(2) At least one of the tries is complete, that is, equals T.
(3) For the incomplete trie, it is an induced subtrie of T which contains all nodes with preorder value smaller than some value.
(4) We incrementally maintain the six precomputed tables of at most $2^{5(\ell+1)\epsilon}$ entries using Lemma 9.

$Lcp(T, P)$ can be computed in $O(\frac{|P|}{\log_\sigma N}\frac{(\log\log N)^2}{\log\log\log N}) = O(\frac{|P|}{\log_\sigma |T|}\frac{(\log\log |T|)^2}{\log\log\log |T|})$ time by querying the complete trie using Lemma 12.

For the operation *Insert_leaf*(T, u, a), a new leaf a is inserted as a child of u in the complete trie using Lemma 12. For the incomplete trie, if u exists, we insert a as a new leaf of the incomplete trie to maintain invariant (3).

For the operation *Delete_leaf*(T, u), we delete u in the complete trie. For the incomplete trie, if u exists, we need to delete it to maintain invariant (3).

To maintain invariant (1), we do the following. After an insertion, if $|T| > 2^{\ell-1}$, we remove the trie implemented using $N = 2^\ell$ and introduce an empty trie implemented using $N = 2^{\ell+2}$. After a deletion, if $|T| \le 2^{\ell-2}$, we remove the trie implemented using $N = 2^{\ell+1}$ and introduce an empty trie implemented using $N = 2^{\ell-1}$.

To maintain invariant (2), we grow the incomplete trie and ensure that the incomplete trie is grown from empty to complete when the trie T is doubled or halved. More precisely, when the incomplete trie is introduced, it consists of only the root node, and a pointer P pointing to it is stored. We also store another pointer Q pointing to the root node of the complete trie. After each *Insert_leaf* or *Delete_leaf* operation, we start an Euler tour traversal from Q in the complete trie, and visit four nodes. If their corresponding nodes do not exist in the incomplete trie, we create them. We update the pointers to resume the traversal. Because the length of the Euler tour is $2t$ where t is the size of the trie when an empty trie is introduced, the traversal is done after $t/2$ updates. This guarantees that when the trie T is doubled or halved after an incomplete trie is introduced, the incomplete trie becomes complete. □

4 Dynamically Maintaining a Trie with Good Expected Amortized Update Time

In cases where the expected amortized time complexity (as opposed to the worst-case time complexity) is of primary concern, we can update the dynamic trie even more efficiently than in the previous section. More precisely, we can dynamically maintain a trie T using $O(|T|\log\sigma)$ bits of space so that $Lcp(T, P)$ takes $O(|P|/\log_\sigma |T| + \log\log|T|)$ worst-case time while insertion/deletion of a leaf takes $O(\log\log|T|)$ expected amortized time only.

The idea is to simulate Lemma 2 by using the data structure from Lemma 4. Then, we can dynamically maintain a trie T of height $O(\log_\sigma N)$ as follows:

Lemma 13 *Given the six precomputed tables in Lemma 8 of $O(N^{5\epsilon})$ bits for any constant $0 < \epsilon < 0.2$, we can dynamically maintain a trie T of height at most $\frac{\epsilon}{2}\log_\sigma N$ using $O(|T|\log\sigma)$ bits of space such that Lcp takes $O(\log\log N)$ time while Insert_leaf and Delete_leaf take $O(\log\log N)$ expected amortized time.*

Proof The trie representation is the same as in Lemma 10 except that the top-level data structure is implemented using Lemma 4 instead of Lemma 2. □

By utilizing Lemmas 7 and 13, we can dynamically maintain a trie T with no height restrictions according to the following theorem.

Theorem 2 *For any fixed constant* $0 < \epsilon < 0.2$, *we can dynamically maintain a trie* T *using* $O(\epsilon^{-1}|T|\log\sigma)$ *bits of space such that the query* $Lcp(T, P)$ *takes* $O(\frac{|P|}{\log_\sigma |T|} + \log\log|T|)$ *time while insertion or deletion of a leaf takes* $O(\log\log|T|)$ *expected amortized time.*

Proof The trie representation is the same as in Theorem 1 except that Lemma 5 is replaced by Lemma 7 and Lemma 10 is replaced by Lemma 13. □

5 LZ-Compression

This section presents a two-phase algorithm for constructing the LZ78 encoding of an input text $S[1\dots n]$ where $n \le 2^w$. Phase 1 first constructs the LZ-trie based on our dynamic trie data structure from Theorem 1 or Theorem 2 and then enhances the LZ-trie with the auxiliary data structure described in Lemma 14 so that the preorder of any node can be computed efficiently. Next, Phase 2 scans the text S to output the list of preorders of the phrases. Figure 2 describes the details.

5.1 Auxiliary Data Structure for Answering Preorder Queries

The next lemma shows how to build an auxiliary data structure for T to answer preorder queries efficiently. We assume that T has been constructed by using our dynamic data structure, but is fixed from now on.

Lemma 14 *Given a trie* T *represented by the dynamic data structure in Theorem* 1 (*or Theorem* 2), *we can generate an auxiliary data structure of* $O(|T|\log\sigma)$ *bits in* $O(|T|)$ *time from which the preorder of any query node can be computed in* $O(\log\log|T|)$ *time.*

Proof The auxiliary data structure stores information about every linking node u (that is, every $u \in T'$). Firstly, it stores the preorder of u. Next, for the corresponding subtrie τ_u, define $\mathcal{B}$ and the set $\{s_{b(1)}, s_{b(2)}, \dots s_{b(|\mathcal{B}|)}\}$ as in Lemma 10 (or Lemma 13). The auxiliary data structure also stores the following information:

- The set $\{s_{b(1)}, s_{b(2)}, \dots, s_{b(|\mathcal{B}|)}\}$. By Lemma 2 (or Lemma 4), we can extract all strings in $\{s_{b(1)}, s_{b(2)}, \dots, s_{b(|\mathcal{B}|)}\}$ in $O(|\mathcal{B}|(\log\log|T|)^2/\log\log\log|T|)$ time (or $O(|\mathcal{B}|\log\log|T|)$ time), and store them in $O(|\mathcal{B}|\log|T|)$ bits of space using $O(|\mathcal{B}|\log\log|T|)$ time with the y-fast trie data structure [27]. After that, given any string P, we can report the largest i such that $s_{b(i)}$ is alphabetically smaller than or equal to P in $O(\log\log|T|)$ time.
- An array $V[1\dots|\mathcal{B}|]$, where $V[j]$ equals the preorder values of the $s_{b(i)}$. Since each preorder value can be stored in $\log|T|$ bits, the array V can be stored in $|\mathcal{B}|\log|T| = O(|T|)$ bits.
- For each $B_i \in \mathcal{B}$, all strings in B_i are stored in a trie of $O(\log|T|)$ bits using Lemma 8.

```
Algorithm   LZcompress
Input:   A sequence $S[1\dots n]$.
Output: The compressed text of $S$.

    /* Phase 1: Construct the LZ-trie $T$ */
1   Initialize $T$ as consisting of a single node (the root) corresponding to the empty string.
2   Denote the empty phrase as phrase 0.
3   $p = 1$;
4   while  $p \le n$  do
4.1        Find the longest phrase $t \in T$ that appears as a prefix of $S[p\dots n]$.
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 14.

    /* Phase 2: Construct the phrase list $s_1 s_2 \dots s_c$ */
6   $p = 1$; $j = 1$
7   while  $p \le n$  do
7.1        Obtain the length $\ell$ of the next phrase stored by delta-code.
7.2        Find the phrase $t = S[p\dots(p + \ell - 1)] \in T$.
7.3        $q_j$ = preorder index of $t$ in $T$
7.4        Output $(q_j, S[p + \ell - 1])$.
7.5        $p = p + |t| + 1$; $j = j + 1$;
    endwhile
End  LZcompress
```

Fig. 2 Algorithm for LZ-compression

For any query node $v \in T$, let u be the linking node $p(v) \in LN$, let B be the block in τ_u which contains v, and let v' be the node in τ_u corresponding to the smallest string in B. Clearly, the preorder of v equals the sum of: (1) the preorder of u in T; (2) the preorder of v' in τ_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), using the y-fast trie we can find the preorder of v' in τ_u in $O(\log\log|T|)$ time. For (3), by Lemma 8, the preorder v in B can be determined in $O(1)$ time. The lemma follows. □

5.2 Complexity Analysis

We now analyze the complexity of the algorithm. We assume a unit-cost RAM model with word size $\lceil \log n \rceil$, and $\sigma \le n$. As in Sect. 2.3, c denotes the number of phrases. Recall that $\sqrt{n} \le c \le n/\log_\sigma n$. We first prove a simple lemma used in the proof of Theorem 3 below.

Lemma 15 $c \ln \frac{n}{c} \le \frac{n \ln \log_\sigma n}{\log_\sigma n}$.

Proof Write $f(c) = c \ln \frac{n}{c}$. Observe that the function $f(c)$ is increasing with respect to c in the interval $[0 \dots \frac{n}{e}]$, where e denotes the base of the natural logarithm (i.e., $e = 2.71828\dots$), and that $f(c) \le \frac{n}{e}$ for all c. There are two cases:

- Case 1: $\log_\sigma n > e$
  $\frac{n}{\log_\sigma n} < \frac{n}{e}$ together with $c \le \frac{n}{\log_\sigma n}$ implies that $c < \frac{n}{e}$. Hence, $f(c)$ is increasing with respect to c and we obtain $f(c) \le f(\frac{n}{\log_\sigma n}) = \frac{n}{\log_\sigma n} \cdot \ln(\log_\sigma n) = \frac{n \ln \log_\sigma n}{\log_\sigma n}$.
- Case 2: $\log_\sigma n \le e$
  We have $f(c) \le \frac{n}{e} \le \frac{n}{\log_\sigma n} < \frac{n \ln \log_\sigma n}{\log_\sigma n}$.

□

Theorem 3 *Suppose we use the trie data structure in Theorem* 1. Algorithm *LZcompress in Fig.* 2 *builds the* LZ*-trie T and the phrase list using $O(\frac{n}{\log_\sigma n} \frac{(\log\log n)^2}{\log\log\log n})$ time and $O(\frac{n(\log\sigma + \log\log_\sigma n)}{\log_\sigma n})$ bits of working space.*

Proof Phase 1 builds the LZ-trie T during the while-loop in Step 4. Since there are c phrases, the while-loop will execute c times and generate c phrases $s_1, s_2, \dots, s_c$. In the i-th iteration, Step 4.1 identifies phrase s_i in $O(\frac{|s_i|}{\log_\sigma n} \frac{(\log\log n)^2}{\log\log\log n})$ time according to Theorem 1, and Step 4.2 stores the length of s_i by delta-code. Then, Step 4.3 inserts s_i into T using $O((\log\log n)^2 / \log\log\log n)$ time (again, by Theorem 1). Since $\sum_{i=1}^{c} |s_i| = n$, the c iterations take $O(\sum_{i=1}^{c} \frac{|s_i|}{\log_\sigma n} \frac{(\log\log n)^2}{\log\log\log n}) = O(\frac{n}{\log_\sigma n} \frac{(\log\log n)^2}{\log\log\log n})$ time. Lastly, in Step 5, T is enhanced with the auxiliary data structure from Lemma 14 for computing preorder, which takes $O(c) = O(\frac{n}{\log_\sigma n})$ time because the LZ-trie has $O(c)$ nodes.

Next, in Phase 2, for each phrase s_i, the algorithm first retrieves its length ℓ stored by delta-code in Step 7.1. Then, Step 7.2 searches the trie for the node representing the phrase $s_i = S[p \dots p + \ell - 1]$, which takes $O(\frac{|s_i|}{\log_\sigma n} \frac{(\log\log n)^2}{\log\log\log n})$ time by Theorem 1. In Step 7.3, the preorder of phrase s_i is obtained in $O(\log\log n)$ time from the auxiliary data structure of Lemma 14. In total, Phase 2 takes $O(\sum_{i=1}^{c} \frac{|s_i|}{\log_\sigma n} \frac{(\log\log n)^2}{\log\log\log n} + c \cdot \log\log n) = O(\frac{n}{\log_\sigma n} \frac{(\log\log n)^2}{\log\log\log n})$ time.

Hence, the total running time of Phase 1 and Phase 2 is $O(\frac{n}{\log_\sigma n} \frac{(\log\log n)^2}{\log\log\log n})$.

The working space required to build and to store the LZ-trie in Phase 1 is $O(c \log\sigma) = O(\frac{n \log\sigma}{\log_\sigma n})$ bits according to Theorem 1. Step 4.2 stores the length of each phrase s_i by delta-code in $1 + \lceil \log s_i \rceil + 2\lceil \log(1 + \lceil \log s_i \rceil) \rceil$ bits; thus, the total space for storing the lengths of all phrases is $\sum_{i=1}^{c} O(1 + \log s_i) = O(c \log \frac{n}{c}) = O(c \ln \frac{n}{c}) = O(\frac{n \log\log_\sigma n}{\log_\sigma n})$ bits by Lemma 15. Finally, the space required by the preorder auxiliary data structure is $O(c \log\sigma) = O(\frac{n \log\sigma}{\log_\sigma n})$ bits. In total, the algorithm uses $O(\frac{n(\log\sigma + \log\log_\sigma n)}{\log_\sigma n})$ bits. □

In the same way, we obtain:

Theorem 4 *Suppose we use the trie data structure in Theorem 2.* Algorithm *LZcompress in Fig. 2 builds the* LZ*-trie T and the phrase list using $O(\frac{n \log\log n}{\log_\sigma n})$ expected time and $O(\frac{n(\log\sigma + \log\log_\sigma n)}{\log_\sigma n})$ bits of working space.*

Proof Analogous to the proof of Theorem 3. □

Finally, two comments are in order.

Firstly, the working space of our LZ78 encoding algorithm is $O(c\log\sigma + c\log\frac{n}{c})$ according to the proof of Theorem 3. Since $\sqrt{n} \le c$, i.e., $\frac{n}{c} \le c$, the working space must be asymptotically the same as or smaller than that of the output compressed text, which is $\Omega(c\log\sigma + c\log c)$. This shows that representing the LZ-trie by our new dynamic trie data structure yields a highly workspace-efficient LZ-compression algorithm. Also note that the output size is larger than $c\log c \ge \frac{1}{2}\sqrt{n}\log n$, while the tables used by our dynamic data structure have size $O(n^\epsilon)$ for arbitrarily small $\epsilon > 0$.

Secondly, the output code of the algorithm in Fig. 2 differs from the original LZ78. The algorithm outputs the same code as reference [25]; more precisely, the output code represents preorders of the trie, while in the original LZ78 phrases are numbered in the order of their creation. To convert it into the original LZ78, we need one more scan of S using the trie. In the scan, we replace the preorders of the phrases by the original numbers. The output size of [25] is asymptotically the same as the original LZ78.

6 Concluding Remarks

In this paper, we have proposed *linked dynamic tries*, dynamic data structures for storing a set of strings. Using our data structures, a trie T can be stored in $O(|T|\log\sigma)$ bits, where σ is the alphabet size. We have described two variants: the first one supports the insertion or deletion of a leaf in $O((\log\log|T|)^2/\log\log\log|T|)$ worst-case time and computing the longest common prefix between a query pattern P and T in $O(\frac{|P|}{\log_\sigma|T|}\frac{(\log\log|T|)^2}{\log\log\log|T|})$ worst-case time [Theorem 1]; the second one supports insertion/deletion in $O(\log\log|T|)$ expected amortized time and the longest common prefix query in $O(\frac{|P|}{\log_\sigma|T|} + \log\log|T|)$ worst-case time [Theorem 2].

We have also shown how to apply our new data structure to LZ-compression. The previously most efficient algorithms for constructing the Ziv-Lempel encoding (LZ78) of a given string S of length n over an alphabet of size σ run in either: (1) $O(n\log\sigma)$ time and $O(n)$ words of working space; or (2) $O(n(\log\sigma + \log\log n))$ time and $(1+\epsilon)nH_k + O(n(k\log\sigma + \log\log_\sigma n)/\log_\sigma n)$ bits of working space, where $H_k \le \log\sigma$ denotes the k-th order empirical entropy of S and $\epsilon > 0$ is an arbitrary constant [3]. Our new linked dynamic trie data structure yields an algorithm for generating the LZ78 encoding of S in only $O(n(\log\sigma + \log\log_\sigma n)/\log_\sigma n)$ bits of working space which runs in either: (1) $O(n(\log\log n)^2/(\log_\sigma n \cdot \log\log\log n))$ worst-case time [Theorem 3]; or (2) $O(n\log\log n/\log_\sigma n)$ expected time [Theorem 4], i.e., in sublinear ($o(n)$) time and sublinear ($o(n\log\sigma)$ bits) working space for small alphabets ($\sigma = 2^{o(\log n \frac{\log\log\log n}{(\log\log n)^2})}$).

Our linked dynamic trie data structure cannot store any satellite information in nodes directly. However, after constructing a trie, we can add a static auxiliary data structure to compute the preorder of a node in $O(\log\log|T|)$ time. Using the nodes' preorders as indices of an array, we can then store satellite information if needed.

Future work will be to reduce the space from $O(|T|\log\sigma)$ bits to the information-theoretic lower bound, and to improve the time complexity, e.g., by using other predecessor data structures. One might also consider supporting an extended set of operations on a dynamic trie. For example, reporting the degree (the number of children) of a node is easy; it can be done simply by storing the answer for each node using $O(|T|\log\sigma)$ bits. Computing the depth of a node may be done by explicitly storing the depths for the nodes in T' and storing relative depths inside each τ_u. On the other hand, reporting subtree sizes seems difficult because it requires information for all the nodes from a leaf to the root of the trie to be updated.

Acknowledgements The authors would like to thank the anonymous referees for their helpful comments.

References

1. Andersson, A., Nilsson, S.: Improved behaviour of tries by adaptive branching. Inf. Process. Lett. **46**(6), 295–300 (1993)
2. Arroyuelo, D.: An improved succinct representation for dynamic k-ary trees. In: Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008). Lecture Notes in Computer Science, vol. 5029, pp. 277–289. Springer, Berlin (2008)
3. Arroyuelo, D., Navarro, G.: Space-efficient construction of Lempel-Ziv compressed text indexes. Inf. Comput. **209**(7), 1070–1102 (2011)
4. Beame, P., Fich, F.E.: Optimal bounds for the predecessor problem and related problems. J. Comput. Syst. Sci. **65**(1), 38–72 (2002)
5. Blandford, D.K., Blelloch, G.E.: Dictionaries using variable-length keys and data, with applications. In: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005), pp. 1–10 (2005)
6. Darragh, J.J., Cleary, J.G., Witten, I.H.: Bonsai: a compact representation of trees. Softw. Pract. Exp. **23**(3), 277–291 (1993)
7. Devroye, L., Szpankowski, W.: Probabilistic behavior of asymmetric level compressed tries. Random Struct. Algorithms **27**(2), 185–200 (2005)
8. Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications. J. ACM **46**(2), 236–280 (1999)
9. Ferragina, P., Manzini, G.: Indexing compressed texts. J. ACM **52**(4), 552–581 (2005)
10. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
11. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. J. Comput. Syst. Sci. **30**(2), 209–221 (1985)
12. Geary, R.F., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theor. Comput. Sci. **368**(3), 231–246 (2006)
13. Hagerup, T.: Sorting and searching on the word RAM. In: Procedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998). Lecture Notes in Computer Science, vol. 1373, pp. 366–398. Springer, Berlin (1998)
14. Hon, W.-K., Lam, T.-W., Sadakane, K., Sung, W.-K., Yiu, S.-M.: A space and time efficient algorithm for constructing compressed suffix arrays. Algorithmica **48**(1), 23–36 (2007)
15. Kärkkäinen, J., Sutinen, E.: Lempel-Ziv index for q-grams. Algorithmica **21**(1), 137–154 (1998)
16. Kosaraju, S.R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM J. Comput. **29**(3), 893–911 (1999)
17. Lippert, R.A., Mobarry, C.M., Walenz, B.P.: A space-efficient construction of the Burrows-Wheeler transform for genomic data. J. Comput. Biol. **12**(7), 943–951 (2005)

18. Mehlhorn, K., Näher, S.: Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. Inf. Process. Lett. **35**(4), 183–189 (1990)
19. Morrison, D.R.: PATRICIA—Practical Algorithm To Retrieve Information Coded In Alphanumeric. J. ACM **15**(4), 514–534 (1968)
20. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. **31**(3), 762–776 (2001)
21. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. Discrete Algorithms **2**(1), 87–114 (2004)
22. Nilsson, S., Karlsson, G.: IP-address lookup using LC-tries. IEEE J. Sel. Areas Commun. **17**(6), 1083–1092 (1999)
23. Overmars, M.H.: The Design of Dynamic Data Structures. Lecture Notes in Computer Science., vol. 156. Springer, Berlin (1983)
24. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP 2003). Lecture Notes in Computer Science, vol. 2719, pp. 357–368. Springer, Berlin (2003)
25. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006), pp. 1230–1239 (2006)
26. Welch, T.A.: A technique for high-performance data compression. IEEE Comput. **17**(6), 8–19 (1984)
27. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. Inf. Process. Lett. **17**(2), 81–84 (1983)
28. Willard, D.E.: New trie data structures which support very fast search operations. J. Comput. Syst. Sci. **28**(3), 379–394 (1984)
29. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)