Information and Computation 285 (2022) 104794

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

c-trie++: A dynamic trie tailored for fast prefix searches ☆

Kazuya Tsuruta, Dominik Köppl *, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda

ARTICLE INFO

Article history:
Received 7 October 2020
Received in revised form 9 May 2021
Accepted 18 August 2021
Available online 28 August 2021

Keywords:
String dictionary
Compact trie
Hashing
Word-packing
Prefix searches

ABSTRACT

Given a dynamic set $\mathcal{K}$ of k strings of total length n whose characters are drawn from an alphabet of size σ, a keyword dictionary is a data structure built on $\mathcal{K}$ that provides lookup, prefix search, and update operations on $\mathcal{K}$. Under the assumption that $\alpha = w/\lg\sigma$ characters fit into a single machine word of w bits, we propose a keyword dictionary that represents $\mathcal{K}$ in either $n\lg\sigma + \Theta(k\lg n)$ or $|T|\lg\sigma + \Theta(kw)$ bits of space, where $|T|$ is the number of nodes of a trie representing $\mathcal{K}$. It supports all operations in $\mathcal{O}(m/\alpha + \lg\alpha)$ expected time on an input string of length m in the word RAM model. An evaluation of our implementation highlights the practical usefulness of the proposed data structure, especially for prefix searches — one of the most essential keyword dictionary operations.

1. Introduction

A *keyword* K is a string that is uniquely associated with an integer called the *identifier* of K. A *keyword dictionary* is a data structure that maintains a dynamic set of keywords $\mathcal{K}$, and provides the following operations for a string S on it:

- insert(S, i) inserts S into $\mathcal{K}$ and assigns S the identifier i. The identifier is a constant integer unique among all keywords stored in $\mathcal{K}$.
- lookup(S) returns the identifier of S if $S \in \mathcal{K}$, or returns the invalid identifier $\perp$ otherwise.
- delete(K) removes the keyword K from $\mathcal{K}$.
- locatePrefix(S) returns an iterator on the set of identifiers of all keyword in $\mathcal{K}$ having S as a prefix. The iterator can report the next occurrence in constant time.[1]

Unlike standard string dictionaries, we omit the operation access(i) returning the keyword of an identifier i, as this function can be realized by a separate data structure (in case of a trie, e.g., an array of pointers in which the i-th entry points to the node of the trie representing the keyword access(i)). For the performance of practical keyword dictionaries like RDF stores (e.g., [31]), insertions, lookups, and prefix queries are the most crucial operations, on which we want to focus in this article.

A space-efficient representation of a keyword-dictionary is the *compact trie*, in which all unary trie paths are merged such that each internal node has at least two children. In applications such as information retrieval [21] and database

☆ Parts of this work have already been presented at the Data Compression Conference (DCC) 2020 [40].

* Corresponding author.
E-mail address: koeppl.dsc@tmd.ac.jp (D. Köppl).

[1] We return an iterator instead of this set, since most of the later explained data structures support all operations in the same time $\mathcal{O}(t)$ for some t, while this operation would take $\mathcal{O}(t+s)$ time, if the returned set has size s.

https://doi.org/10.1016/j.ic.2021.104794

systems [31,45], when designing compact tries, the main focus is put on properties such as its size and the operation time for insertions, lookups, and prefix queries. In this article, we propose a new compact trie tailored for excelling at these properties.

1.1. Preliminaries

Let lg denote the logarithm to the base two. Our model of computation is the standard word RAM model of word size w. We can read and process $\mathcal{O}(w)$ bits in constant time. Let n be a natural number with $n = \mathcal{O}(2^w)$. Storing an integer of the domain $[1..n]$ costs $\lg n$ bits such that pointers for the problem size n can be represented in $\lg n$ bits (like in the transdichotomous model). The choice of this model (severing the connection between word size and the logarithm of the problem size) is justified by the fact that the register sizes of SIMD instruction sets are increasing since the recent years significantly (e.g., AVX512 with 512-bit registers).

Let Σ be an integer alphabet of size $\sigma \leq 2^w$. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. We write $S[i]$ for the i-th character of S, for $1 \leq i \leq |S|$. The *empty string* is the string with length zero. For a string $S = XYZ$ with $X, Y, Z \in \Sigma^*$, X, Y, and Z are called a *prefix*, *substring*, and *suffix* of S, respectively. Since X, Y, or Z may be empty, X and Z are also substrings of S at the same time by this definition. The word RAM model allows us to process $\alpha = \mathcal{O}(w/\lg\sigma)$ characters in constant time.

For the rest of the article, let $\mathcal{K}$ denote a set consisting of k keywords with a total length of $n = \sum_{K\in\mathcal{K}} |K|$. We suppose that $\mathcal{K}$ is dynamic, and that the integers k and n are variable. The keywords of $\mathcal{K}$ do not have to be prefix-free.

1.2. Related work

Keyword dictionaries are an integral data structure with a plethora of applications (e.g., n-gram language models [35], compression [17], input method editors [28], query auto-completion [21], or range query filtering [45]). As a well-studied abstract data type they also have many representations. We refer to standard literature like [37, Chapter 5.2], [32, Chapter 28], or [33, Chapter 8.5.3] for an introduction to common representations like tries. Here, we highlight some of the most recent representations, which are all (compact) tries. A major design choice for a trie is whether to *compact* the unary edges, spawning two lines of research, which we want to analyze in the following.

For the analysis, let $|T| \leq n$ denote the number of nodes of a trie T storing $\mathcal{K}$, and let m be the length of an input string for one of the trie operations. We start with the non-compact representations:

- The *HAT-trie* [4] is a practically optimized version of the burst trie [20]. It suppresses the number of trie nodes by selectively collapsing subtries into cache-conscious hash tables of strings [5]. Although there is no discussion of prefix searches in [4], the implementation of Tessil[2] supports locatePrefix. We are unaware of any theoretical results regarding space or time.
- The double array [2] simulates a trie by using two integer arrays to find a child in constant time, and thus can perform lookup in $\mathcal{O}(m)$ time. Although the double array includes some vacant slots and consumes $\Omega(n \lg n)$ bits, those vacant slots have a negligible memory effect in practical implementations such as the Cedar trie [44]. In the static setting, Kanda et al. [25] proposed a practically compressed data structure for the two arrays. However, for any of these data structures, it is not clear to us what time is needed for answering locatePrefix.
- The Bonsai trie [13] is a trie whose nodes are maintained in a compact hash table [12]. Modern variants [36] use $\mathcal{O}(n \lg\sigma)$ bits of space in expectancy, and perform insert and lookup in $\mathcal{O}(m)$ expected time. However, it is not clear how to perform locatePrefix efficiently.
- Kanda et al. [24] proposed a dynamic variant of the path decomposed trie of Ferragina et al. [16] by means of *incremental* path decomposition. This dynamic trie supports insert and lookup in $\mathcal{O}(m)$ expected time. However, there is no discussion about prefix searches. Actually, as Kanda et al.'s trie is based on the Bonsai trie, it faces the same problem for locatePrefix.

Considering compact tries, we are aware of the following representations:

- Jansson et at. [23] presented a dynamic trie using $\mathcal{O}(|T| \lg\sigma)$ bits, in which a leaf can be inserted or deleted in $\mathcal{O}((\lg\lg|T|)^2/\lg\lg\lg|T|)$ time. This trie can compute a prefix search in $\mathcal{O}((m/\log_\sigma|T|)(\lg\lg|T|)^2/\lg\lg\lg|T|)$ time [23, Thm. 1]. In an alternative representation, this trie supports insertions and deletions of leaves in $\mathcal{O}(\lg\lg|T|)$ expected amortized time while supporting a prefix search in $\mathcal{O}(m/\lg_\sigma|T| + \lg\lg|T|)$ worst-case time [23, Thm. 2].
- The *(dynamic) z-fast trie* is a keyword dictionary of Belazzougui et al. [7], which uses $|T|\lg\sigma + \Theta(kw)$ bits of space, and supports all operations in $\mathcal{O}(m/\alpha + \lg m + \lg\lg\sigma)$ expected time.[3]
- Takagi et al. [39] proposed the *dynamic packed compact trie*, whose name we abbreviate to *packed c-trie*. The packed c-trie uses $|T|\lg\sigma + \Theta(kw)$ bits of space, and supports all operations in $\mathcal{O}(m/\alpha + \lg w)$ expected time.

[2] https://github.com/Tessil/hat-trie.

[3] This time bound can be achieved by omitting the jump pointers in [7, Sect. 3.4] since their maintenance needs additional time. The jump pointers are used to enable additional operations on the trie such as predecessor queries, which we omit in this article.

Table 1
Space complexities of the packed tries addressed in Sect. 1.2 for maintaining k keywords of total length n under different settings: In Setting 1, we concatenate all keywords to a large string of length n. In this large string, we can address every substring with two pointers of n bits. We omit Setting 1 for the other compact trie data structures as these (except the plain compact trie) use auxiliary data structures taking $\Theta(kw)$ bits. In Setting 2, we represent each keyword K with front coding [42, Sect. 4.1], i.e., we represent K by $K[\ell+1..|K|]$ if the longest common prefix of K with its lexicographically preceding keyword in $\mathcal{K}$ is ℓ. Hence, we store the suffix $K[\ell+1..|K|]$ in the trie data structure explicitly as a string. By doing so for each keyword, we store k strings with a total length of $|T|$. Except the plain compact trie, all listed tries store additionally parts of a keyword in $\Theta(w)$ bits (to apply word-packing techniques), causing $\Theta(kw)$ bits of additional space.

Trie	Space in bits	Setting
c-trie♯	$n\lg\sigma+\Theta(k\lg n)$	1
c-trie♯	$\lvert T\rvert\lg\sigma+\Theta(kw)$	2
compact trie	$\lvert T\rvert\lg\sigma+\Theta(k\lg\lvert T\rvert)$	2
z-fast trie [7]	$\lvert T\rvert\lg\sigma+\Theta(kw)$	2
c-packed trie [39]	$\lvert T\rvert\lg\sigma+\Theta(kw)$	2

- HOT [10] is an algorithmically engineered trie that applied different strategies depending on the distribution of the common prefix lengths of the keywords to obtain high fanouts and minimize the depth of the trie. It also applies AVX2 instructions for lookup queries.

The following keyword dictionaries are static, but share common traits with our proposed data structure:

- Grossi and Ottaviano [18] proposed a cache-friendly trie dictionary through path decomposition [16]. An operation can be carried out in $\mathcal{O}(m+h\log\sigma)$ time, where h is the height of the path-decomposed trie. The data structure is stored in compressed space by exploiting text compression techniques and succinct data structures.
- The Marisa trie, developed by Yata [43], is a static trie that consists of recursively compressed Patricia tries stored in the level-order unary degree sequences (LOUDS) representation [22]. It recursively encodes edge labels in a Patricia trie using another Patricia trie. Yata's implementation[4] supports prefix searches.
- Arz and Fischer [3] proposed a static compressed trie by adapting the LZ78 trie to basic dictionary operations such as lookup. Their trie uses $\mathcal{O}(k\lg n+n\lg\sigma)$ bits of space. It can answer lookup in $\mathcal{O}(m)$ expected time. However, we are not aware of whether this data structure supports efficient prefix searches.
- Bille et al. [8] presented a static keyword dictionary using $\mathcal{O}(n\lg n)$ bits of space and $\mathcal{O}(n)$ time to represent $\mathcal{K}$. It supports queries in $\mathcal{O}(m/\alpha+\lg m+\lg\lg\sigma)$ time.
- A recent approach is due to Bille et al. [9], who proposed a static keyword dictionary with $\mathcal{O}(n\lg\sigma)$ bits of space using $\mathcal{O}(\min(m\lg\sigma, m+\lg n))$ time for an operation in the pointer machine model.
- The fast succinct trie (FST) is a trie data structure used in the succinct range filter [45]. An FST is divided into two layers at a specific height. The top layer is represented by a *speed*-optimized trie while the bottom layer is represented by a *space*-optimized trie. Both tries are represented in the LOUDS representation [22].

In this article, we present a new keyword dictionary based on techniques used for devising one of the aforementioned compact trie representations:

Theorem 1.1. Given a dynamic set $\mathcal{K}$ of k keywords whose characters are drawn from an integer alphabet of size $\sigma\le 2^w$, there is a keyword dictionary representing $\mathcal{K}$ in either $n\lg\sigma+\Theta(k\lg n)$ or $|T|\lg\sigma+\Theta(kw)$ bits of space, where $n=\sum_{K\in\mathcal{K}}|K|$ is the total length of all keywords of $\mathcal{K}$ and $|T|$ is the number of nodes of a trie representing $\mathcal{K}$. It supports all keyword dictionary operations in $\mathcal{O}(m/\alpha+\lg\alpha)$ expected time with $\alpha=w/\lg\sigma$ on an input string of length m.

Depending on how we represent the input keywords, we obtain two different space complexities in Theorem 1.1, which we put into comparison in Table 1. The time and space bounds of Theorem 1.1 are an improvement to all previous compact trie representations we are aware of (e.g., the z-fast trie becomes inferior for string lengths $m>w/\lg^2\sigma$).

Prefix searches arise in various uses of suffix trees, e.g., computing matching statistics [19], online suffix tree construction [41], online Lempel-Ziv 77 factorization [46], just to name a few. Hence, the time bound for prefix search is of significant theoretical interest, and our compact trie moves the best known upper bound for the *expected* time closer to the trivial lower bound $\Omega(m/\alpha)$ for reading a pattern of length m word-packed. Also, with delete and insert operations, one can efficiently maintain the *sparse suffix tree* [26] for a dynamic set of suffixes to index.

Our experiments in Sect. 3 reveal that the above improvements are also practically significant. We note that other previous trie data structures mentioned earlier have the following drawbacks: (1) For the HAT-trie or the double array, there

[4] https://github.com/s-yata/marisa-trie.

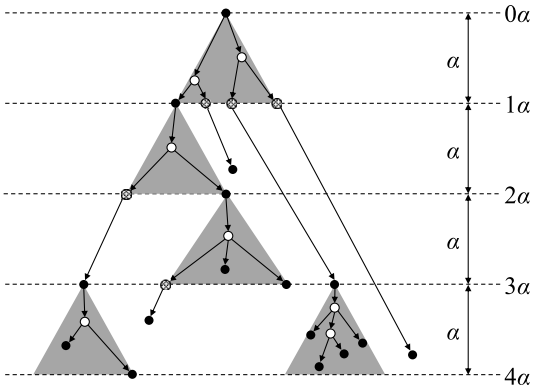

Fig. 1. The macro trie of a c-trie♯ instance. Micro tries are represented by shaded triangles (cf. [39, Fig. 2]). Circles filled with black color are macro trie nodes. Hollow circles are nodes stored exclusively in a micro trie. Cross-hatched circles are nodes of a micro trie that are not present in the standard compact trie (as they have only one child). These nodes are leaves of a micro trie, and are needed for navigating between the micro trie and the macro trie nodes below of it.

Fig. 2. Splitting the edge (w, v) protruding the boundary of the micro trie rooted at node u by introducing an artificial micro trie node x. The string depth of u is $d\alpha$ while v with a string depth larger than $(d+1)\alpha$ does not belong to u's micro trie.

are no known nontrivial space and construction time bounds as their constructions are based on heuristics. In practice, they are also not favorable for prefix queries. (2) Trie data structures based on the Bonsai trie have the major drawback that enumerating children is done by querying for each possible edge label in a brute force manner. So they are no-good candidates for prefix search queries, and are therefore omitted in our practical evaluation. (3) The trie data structure of Jansson et at. [23] is theoretically appealing, but uses theoretically sophisticated data structures for which an efficient implementation seems cumbersome.

2. Keyword dictionary c-trie♯

Tuning for fast prefix searches, our idea is to devise a new keyword dictionary based on the compact trie data structures, as they are practically faster than approaches based on the double array when the prefixes in question are relatively short to the stored keywords. Our approach, called c-trie♯ for *improved compact trie*, is a hybrid of the z-fast trie and the packed c-trie. Like these two trie representations, the compact trie is decomposed into a macro trie storing micro tries.

For a formal explanation of this decomposition, let the *string depth* of a node u denote the length of the concatenation of all labels on the path from the macro trie root to u. To keep the following explanation simple, let us assume, for the time being, that the keyword set $\mathcal{K}$ is prefix-free such that each leaf corresponds to one keyword. (In the general case, we do not only consider leaves but also internal nodes corresponding to a keyword.) Our starting point is a compact trie. If there is an edge leading to an internal node, we split up this edge by creating additional nodes on this edge whose string depths are multiples of α. Subsequently, we put all nodes whose string depths are multiples of α into the macro trie. Let u be one of these nodes, and let $d\alpha$ be its string depth. Then u becomes the root of a micro trie if it has more than one descendant in the compact trie whose string depth is at most $(d+1)\alpha$. Suppose that u is the root of a micro trie, then this micro trie stores all of u's descendants (of the compact trie) whose string depths are at most $(d+1)\alpha$. Every edge (w, v) from a node w of u's micro trie leading to a descendant v of u with a string depth larger than $(d+1)\alpha$ is split into (w, x) and (x, v) for an artificial node x with string depth $(d+1)\alpha$ (cf. the cross-hatched circles in Fig. 1 and Fig. 2 for a schematic illustration). Finally, leaves of the compact trie are macro trie nodes. As previously explained, there can additionally be micro trie nodes if (a) their string depths are between $d\alpha$ and $(d+1)\alpha$ and (b) they have an ancestor with string depth $d\alpha$ that is the root of the respective micro trie. Consequently, the total number of micro and macro trie nodes is bounded by $\mathcal{O}(\tilde{k})$, where $\tilde{k} = \Theta(k)$ is the number of nodes in the compact trie. Fig. 1 captures this schematically.

For c-trie♯, we apply the explained trie decomposition, which coincides with the trie decomposition of the packed c-trie for the macro trie. Our micro tries are *alphabet-aware z-fast tries*, whose definition follows. The z-fast trie proposed by Belazzougui et al. [7] works on binary strings. Their results on micro trees work for binary strings up to length $\mathcal{O}(w)$. Here, we propose a variant, the *alphabet-aware z-fast tries*, that manages strings on the alphabet Σ up to length $\mathcal{O}(w/\lg\sigma) = \mathcal{O}(\alpha)$ by packing $\mathcal{O}(\alpha)$ characters in a constant number of machine words:

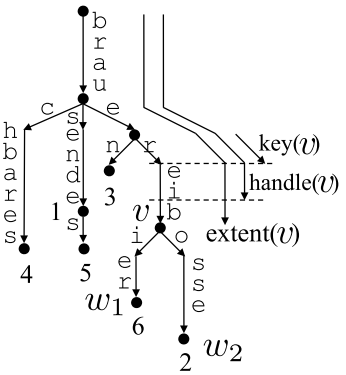

Fig. 3. The micro trie built on our running example $\mathcal{K} = \{K_1 = \texttt{brausende}, K_2 = \texttt{brauereibosse}, K_3 = \texttt{brauen}, K_4 = \texttt{brauchbares}, K_5 = \texttt{brausendes}, K_6 = \texttt{brauereibier}\}$, which is not prefix-free. A leaf u storing number i is associated with the identifier i, i.e., $\text{extent}(u) = K_i$. In this example, the node v storing the extent brauereib has the leaves w_1 and w_2 representing the keywords K_6 and K_2, respectively, as its children, which are determined by their keys $\text{key}(w_1) = \texttt{i}$ and $\text{key}(w_2) = \texttt{o}$, respectively. If we assume that eight characters fit into a computer word, then the extent of v is outside of the micro trie containing the root node. This fact is symbolized by the dashed line separating the eighth and the ninth character of extent(v).

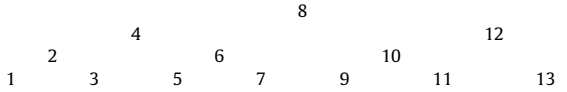

Fig. 4. Geometric interpretation of the 2-fattest numbers. Based on the number line of natural numbers, we move each natural number as many position upwards as it has trailing zeros in its binary representation. In this example of numbers from 1 to 13, the highest number is 8. Now the 2-fattest number of a query interval is the highest number within this range.

Lemma 2.1. *Let $\mathcal{K}$ be a dynamic set of k keywords whose characters are drawn from an alphabet of size $\sigma \le 2^w$. Given that each keyword of $\mathcal{K}$ has a length of $\mathcal{O}(\alpha)$, there is a keyword dictionary representing $\mathcal{K}$ in either $n \lg \sigma + \Theta(k \lg n)$ or $|T| \lg \sigma + \Theta(kw)$ bits of space, where $\alpha = w/\lg\sigma$, $n = \sum_{K\in\mathcal{K}} |K| \le \alpha|\mathcal{K}|$ is the total length of all keywords of $\mathcal{K}$, and $|T|$ is the number of nodes of a trie representing $\mathcal{K}$. It supports all keyword dictionary operations in either $\mathcal{O}(\lg\alpha)$ expected time or $\mathcal{O}(\lg\alpha \lg^2 \lg\sigma / \lg\lg\sigma)$ deterministic time.*

An operation with a string of length m with $m = \Omega(\alpha)$ (but with $m = \mathcal{O}(2^w)$) involves the traversal of the macro tree, which is done in $\mathcal{O}(m/\alpha)$ expected time[5] for all keyword dictionary operations [39]. Combining the operations in the macro trie and in the micro tries gives $\mathcal{O}(m/\alpha + \lg\alpha)$ total time, and concludes Theorem 1.1.

2.1. Micro tries

For explaining c-trie♯ in detail, we start with a review of the z-fast trie under the light of our alphabet-aware variant. We say that a node v is *associated with the identifier* of a keyword K if we can read K by following the path from the root to v. The alphabet-aware z-fast trie is a compact trie in which each leaf v is associated with the identifier of a keyword. An internal node has at least two children unless it is also associated with the identifier of a keyword. If the set of keywords $\mathcal{K}$ is prefix-free, then there are no nodes with a single child.

Fig. 3 presents an instance of such a trie. The figure also depicts the following definitions that are substrings or nodes associated to each node of an alphabet-aware z-fast trie.

- $\text{key}(v)$ is the first character in the label of the edge connecting v with its parent. It is undefined if v is the root.
- $\text{extent}(v)$ is the string obtained by concatenating the edge labels of the path from the root node to v.
- $\text{exit}(S)$ is the highest node v for which, among all other nodes, the longest common prefix between S and $\text{extent}(v)$ is the longest.
- $\text{parex}(S)$ is the parent node of $\text{exit}(S)$, or a special symbol $\perp$ with $\text{extent}(\perp) = 1$ if $\text{exit}(S)$ is the root node.

It is left to explain for what $\text{handle}(v)$ stands in the figure. For that we need the notion of 2-fattest numbers [7, Def. 1]. The *2-fattest number* of an interval $[\ell..r]$ of positive integers $0 < \ell \le r$ is the integer in $[\ell..r]$ with the most trailing zeros in its binary representation (see Fig. 4 for a geometric interpretation). Given a node v with its parent u, we can compute the 2-fattest number f of $[|\text{extent}(u)| + 1..|\text{extent}(v)|]$ to determine the handle of v, which is $\text{handle}(v) := \text{extent}(v)[1..f]$. In case that v is the root, we set $\text{handle}(v)$ to the empty string.

[5] See Sect. 2.2 for a detailed description of the macro trie.

For supporting the keyword dictionary operations, we need operations to descend in a micro tree. For that, the trie maintains a dictionary DicHandle that can address each internal node u by its handle $\mathrm{handle}(u)$. Additionally, we need a way to navigate from a node to one of its children. This can be done in constant time in the original z-fast trie since it works on a binary alphabet (hence, each node has at most two children). For the alphabet-aware variant, each internal node u stores a dictionary $\mathsf{DicChild}_u$ to access one of its child nodes v by the character $\mathrm{key}(v)$. The proof of Lemma 2.1 gives us two different, possible representations for DicChild:

Proof of Lemma 2.1. Since the edge labels in the alphabet-aware z-fast trie are characters drawn from the integer alphabet Σ, traversing from a node to a specific child costs $\mathcal{O}(\sigma)$ time. We improve this time by augmenting each node with a data structure maintaining its children such that, given a node v and a character c, we can navigate from v to its child connected with the edge starting with c by querying this data structure having stored c and v as key and value, respectively. This data structure can be realized with a hash table with constant expected time, or with a dynamic predecessor data structure like [6] (combined with the transformation of Andersson and Thorup [1]) taking $\mathcal{O}(m \lg n)$ bits and supporting all operations in $\mathcal{O}(\lg\lg\sigma \lg\lg m / \lg\lg\lg\sigma) = \mathcal{O}(\lg^2\lg\sigma / \lg\lg\lg\sigma)$ deterministic time when storing $m \le \sigma$ elements (the space bounds are due to the fact that we store pointers to the specific children as satellite data). This sums up to $\Theta(k \lg n)$ bits because we have $\Theta(k)$ trie nodes. □

For the algorithmic part, we follow Algorithm 1 and Section 3.3 of [7]. Given a pattern P of length $\mathcal{O}(\alpha)$, this algorithm locates exit(P) and parex(P). Having exit(P) and parex(P), we can perform all keyword dictionary operations as in the z-fast trie. The idea of the algorithm is to perform a search on the interval $[\ell..r]$, which is set to $[1..|P|]$ at the beginning. The search handles this interval similarly to a binary search with the aim to find the lowest node whose handle is a prefix of P. For explanation, the algorithm is divided into rounds. In each round, it (a) either enlarges ℓ or shrinks r, (b) computes the 2-fattest number f of $[\ell..r]$, and (c) queries DicHandle with the handle $P[1..f]$. If there is a node v with $\mathrm{handle}(v) = P[1..f]$, the algorithm has matched $P[1..f]$ with this node and simulates the descending to this trie node by setting $\ell \leftarrow |\mathrm{extent}(v)|$. Otherwise (there is no such node v), the algorithm sets $r \leftarrow f - 1$ to aim for jumping to a node whose extent is less than f. The algorithm stops when it finds either exit(P) and parex(P) [7, Thm. 3], which is after $\mathcal{O}(\lg|P|)$ rounds. If exit(P) is found, it has previously already computed parex(P). Otherwise, it takes that child of parex(P) whose edge connected to parex(P) leads us to exit(P). For finding this child, the algorithm uses $\mathsf{DicChild}_{\mathrm{parex}(P)}$. Finally, the updates can be conducted with a constant number of pointer updates (detailed are described in [7, Sect. 5]).

In the context of the example of Fig. 3, this algorithm applied to $P = \texttt{brauereibock}$ gives us the node exit(P), which is the node v visualized in Fig. 3. From there, we can query $\mathsf{DicChild}_{\mathrm{exit}(P)}$ for the predecessor (resp. successor) with the character ○ to find the predecessor (resp. successor) of P, which is K_6 (resp. K_2).

2.2. Macro trie

It is left to describe the macro trie borrowed from the packed c-trie, and to analyze the space and time complexity of c-trie♯. The macro trie is needed to cope with keywords longer than α characters, or w bits. The rough idea is to partition a long keyword into chunks of w bits, and maintain the chunks in a dictionary DicChunk similar to DicHandle, mapping w-bit chunks to macro trie nodes. Given that the root is at height 0, a node on a height h of the macro trie is endowed with

- a micro trie representing its descendants whose extents are at most $(h+1)w$ bits long, and with
- a DicChunk representing its children whose extents are longer than $(h+1)w$ bits.

Its DicChunk stores the w-bit substring starting at the $(hw+1)$-th bit of the extents of its respective children, where the *extent of a macro trie node* v is the binary representation of the string read from the path from the macro tree root to v. (Consequently, the string depth of a node is the length of its extent.) An update of the trie involves a lookup of the insertion or deletion position, and a modification of a DicChunk or a micro trie.

Space complexity Our keyword dictionary c-trie♯ maintains $\mathcal{O}(k)$ macro and $\mathcal{O}(k)$ micro nodes. Each node stores a pointer to a substring of a keyword. The keywords are stored either in a concatenated string of length $n\lg\sigma$, or are compressed via front coding [42, Sect. 4.1] taking $|T|\lg\sigma$ bits in total. We store $\mathrm{extent}(v)$ of a node v either as two $(\lg n)$-bit pointers to the concatenated string (former case) or verbatim in w-bits (latter case). Since the number of total nodes stored in the DicChilds, the DicHandles and the DicChunks is $\mathcal{O}(k)$, the data structure needs in total either $n\lg\sigma + \Theta(k\lg n)$ or $|T|\lg\sigma + \Theta(kw)$ bits. This gives the bounds in Table 1.

Time complexity Given a pattern P of length m, we can traverse the macro trie by visiting at most m/α macro trie nodes to find the micro trie τ storing the node whose extent has the longest common prefix with P. After reaching τ, we can compute the handle of a node from its extent in constant time, since the 2-fattest number in $[\ell..r]$ is the integer $(-1 \ll \mathtt{msb}((\ell-1) \oplus r))\&r$, where $\ll$, $\mathtt{msb}$, $\oplus$ and & denote the bitwise left shift, the function retrieving the most significant bit, the bitwise exclusive-OR and the bitwise AND operators, respectively. In total, we query $\mathcal{O}(m/\alpha)$ DicChunks, τ's

DicHandle $\mathcal{O}(\lg\alpha)$ times, and $\mathsf{DicChild}_{\mathrm{parex}(P)}$ at most one time, yielding $\mathcal{O}(m/\alpha + \lg\alpha)$ expected time as claimed in Theorem 1.1 if all dictionaries can lookup an entry in constant expected time. Choosing a suitable representation for DicHandle, DicChild, and DicChunk is the major task of the next subsection dealing with practical aspects of c-trie♯.

2.3. Implementation techniques

On the practical side, our major improvements are based on the three ideas:

Task 1 representing each node by an identifier (ID) to store IDs instead of node pointers,
Task 2 storing a global mapping from $\mathrm{extent}(v)$ to node IDs, and
Task 3 representing the dictionaries with different data structures with focus on either speed or memory efficiency.

Micro tries Each node v stores $\mathrm{extent}(v)$, which can be represented in a constant number of computer words. From $\mathrm{extent}(v)$ we can deduce $\mathrm{handle}(v)$ and $\mathrm{key}(v)$ in constant time. Therefore, the dictionaries DicChild and DicHandle have no need to store the keys of their entries since it suffices to maintain the nodes with which a dictionary can restore the respective keys on demand. By doing so, a lookup of a node v with a key $\mathrm{handle}(v)$ (resp. $\mathrm{key}(v)$) needs to compute $\mathrm{handle}(w)$ (resp. $\mathrm{key}(w)$) of each node w in question for comparison. By conducting the comparisons in this way, we save memory by omitting the keys at the expense that the benefits of current processors featuring large cache lines become negligible in this context. (Imagine a linear probing hash table storing the keys explicitly, where we can fetch keys stored at successive cells at once for collision probing.) Here, we embrace the cuckoo hashing [34] technique, which has strong theoretical results in the pointer machine model. This concludes our approach for Task 1.

Node factory In our setting, we assume that k is much smaller than n. Otherwise, c-trie♯ becomes unfavorable with respect to other trie data structures like the Bonsai trie. That is because our trie data structure contains $\Theta(k)$ nodes in total. However, using w bits for a pointer to a node is wasteful. Instead, we want to store node pointers in $\Theta(\lg k)$ bits as hinted in the description of our computational model in Sect. 1.1. For that (as highlighted in Task 2), we store each node in a global two-dimensional array that assigns each node an integer represented in $\Theta(\lg k)$ bits, which we set to 32 bits for the experiments. By storing 32-bit integers instead of pointers on commodity computers with a word size of $w = 64$ bits, we can roughly halve the memory requirement for maintaining DicChild and DicHandle.

Macro trie Like for DicHandle, we use a cuckoo hash table for representing the DicChunks. We again just store the nodes in the cuckoo hash table, since we can restore their keys by extracting the respective w-bit substring in constant time. We also maintain a separate node factory storing the macro trie nodes.

Cuckoo hashing and practical considerations Our cuckoo hash table H uses three hash functions. We restrict the hash table size $|H|$ to be a power of two such that we can map a hash value to $[1..|H|]$ more quickly by using bit shifts instead of a modulo operation (cf. the discussion in [38, Sect. 1]; however, new techniques [29] can speed this up). An insertion collision occurs if each of the entries located by the hash functions is already occupied. Given such a collision on inserting an element e, we start a random walk by selecting the i-th hash function h_i for a random i, swapping $H[h_i]$ with e and recurse. If this walk is unsuccessful after a certain number of steps, the hash table doubles its size. To keep the memory requirement at minimum, the chosen hash functions are determined at startup and are the same across all cuckoo hash tables. The hash functions are based on three xorshift operations borrowed from MurmurHash[6] and two multiplications with different 64-bit integer seeds. Unwisely chosen seeds can result in a failure of the data structure, as the hash functions are immutable (changing would cause to rehash *all* cuckoo hash table instances). However, this was not a problem in our experiments. While insertions take $\mathcal{O}(1)$ expected time for a sufficiently small *load factor*, i.e., the maximum ratio between the number of stored elements and $|H|$ before doubling the size of H, a lookup takes $\mathcal{O}(1)$ worst case time. The load factor in combination with the threshold on the maximal number of iterations for collision handling does not have much influence on the final size, since a higher load factor makes it more probable that an insertion collision exceeds the threshold. Setting this threshold to a smaller value boosts the insertion speed at the expense of a higher risk of creating an unnecessarily large table. However, preliminary experiments were in favor for a small threshold around 100 iterations. For the experiments in the following section, we fixed the threshold to 100, and set the load factor to 0.9.

First-child next-sibling representation In practice, the Cuckoo hash tables used for representing the dictionaries DicChild waste non-negligible space as (a) each micro trie node stores such a hash table, and (b) the hash tables may not always become full. For space efficiency, we did not follow this approach, but instead represent all DicChild dictionaries of a micro trie with a single trie data structure in the *first-child next-sibling (FNCS)* representation (see [30] for a definition). In this representation, we maintain two arrays for (a) the first children and (b) the next siblings, where (a) and (b) are pointers gained from the node factory. For navigation in the FNCS representation it is necessary to know the character of the in-going edge of each node v, but this information is already given by querying $\mathrm{key}(v)$. This concludes our strategy for Task 3.

[6] https://github.com/aappleby/smhasher/wiki/MurmurHash3.

Table 2
Characteristics of our keyword sets. The total length of all keywords is n. The number of keywords is k. The average and maximum length of a keyword is written in the columns $\varnothing len$ and *max-len*, respectively. The columns $\varnothing LCP$ and *max-LCP* show, respectively, the average length and the maximal length of the longest common prefixes of all keywords. The number of nodes a compact trie C stores is given by $|C|$.

$\mathcal{K}$	$\frac{n}{10^6}$	σ	$\frac{k}{10^3}$	$\varnothing$len	max-len	$\varnothing$LCP	max-LCP	$\frac{\|T\|}{10^6}$	$\frac{\|C\|}{10^3}$
`proteins`	903	26	2,982	302.8	36,805	38.8	16,190	787	5,778
`urls`	1,413	98	18,564	76.1	2,048	60.9	2,006	282	35,343
`dblp.xml`	169	96	2,950	57.6	685	34.4	104	68	5,900
`geographic`	107	134	7,308	14.6	151	8.5	247	45	12,802
`commoncrawl`	121	113	1,995	61.0	1,194,988	12.9	119,276	96	3,740
`vital`	243	203	494	493.3	9,794	12.7	1,806	238	986

Number of nodes The implementations of the compact trie, the packed c-trie, the z-fast trie, and c-trie♯ have same number of nodes. This seems to contradict the prior statement (cf. Fig. 2) that the packed c-trie and c-trie♯ additionally introduce nodes at string depths $k\alpha$ for an integer k. However, we introduced these nodes only for didactic reasons. The actual implementations do not create these nodes as we can use the respective (actually created) descendants of these nodes as well.

3. Experiments

Finally, we analyze the empirical performance of c-trie♯ with respect to time and memory consumption. In particular, we are interested in the running time of insert, lookup, locatePrefix, and delete. Our implementation of c-trie♯ is written in C♯, and available at https://gitlab.com/habatakitai/ctriepp. For the experiments, we set up a machine equipped with CentOS 6.10, with an Intel Xeon X5560 processor running at 2.80 GHz, and with 198 GB of main memory.

3.1. Datasets

For an objective evaluation, we took a variety of datasets having different characteristics (cf. Table 2):

- `proteins` contains different sequences of amino acids.
- `dblp.xml` is part of the XML dump of the dblp.org website.
- `urls` is a crawl of webpages of the `.uk` domain from the WebGraph framework.[7]
- `geographic` contains names of different geographic locations collected by the GeoNames database.[8] Our keywords are extracted from the `ascii name` column.
- `commoncrawl` is a web crawl containing the ASCII-encoded content (without HTML tags) of random web pages extracted from Common Crawl.[9]
- `vital` is the main text extracted from the most vital Wikipedia articles.

The datasets `proteins` and `dblp.xml` are from the Pizza&Chili Corpus.[10] The datasets `commoncrawl` and `vital` are provided by the tudocomp framework [15].

We interpreted each dataset as a single string on the byte alphabet. We partitioned this string into keywords by splitting it either at newline characters or at full stops, and removed all duplicates afterwards. The resulting keyword sets are the input of our experiments.

Table 3 lists additional characteristics of the computed keywords of each dataset regarding their lengths and longest common prefixes. We observe that the lengths have a distribution that is more Gaussian, and by no means uniform. The lengths have also an impact on the sizes and shapes of the dictionaries, as can be seen in Table 4.

3.2. Dictionaries of c-trie♯

In Table 4, we give insights in how large the dictionaries DicChild and DicHandle of the c-trie♯ become when indexing one of our datasets. The distributions in Tables 4a and 4b justify our selection of a lightweight data structure with worse asymptotic behavior (FNCS representation) for DicChild, and the use of the more heavyweight cuckoo hash table for DicHandle. We also did experiments with representing each DicChild as a sorted (or unsorted) list storing newly inserted children with insertion sort (or just at the end of the list). These experiments showed that lists feature a small speed-up for

[7] http://law.di.unimi.it/webdata/uk-2002.
[8] http://download.geonames.org/export/dump/allCountries.zip.
[9] https://commoncrawl.org/.
[10] http://pizzachili.dcc.uchile.cl.

Table 3
Histogram of (a) keyword lengths and (b) the lengths of the longest common prefixes (LCPs) of the keywords. While Table 2 captures the average and maximal lengths of the keywords and their LCPs, these tables give an insight in the distributions of the lengths and the LCPs. A length is counted in the i-th row if is i for $i = 1$ and $i = 2$, or belongs in $[2^{i-2} + 1..2^{i-1}]$ for $i \geq 3$.

i	proteins	urls	dblp.xml	geographic	commoncrawl	vital
1	19	85	2	11	97	39
2	132	851	1	262	1,546	26
4	5,485	7,888	0	31,036	31,931	131
8	36,973	25,921	5	1,270,765	137,074	726
16	75,796	24,188	25	3,899,303	636,922	2,298
32	66,530	197,634	395,244	1,838,186	445,153	4,932
64	130,527	8,620,706	1,801,952	263,086	369,674	12,007
128	481,117	8,463,502	723,011	5,398	255,830	32,038
256	818,538	1,100,909	29,782	7	61,018	75,871
512	955,403	100,867	213	0	36,936	166,775
1,024	343,983	19,207	2	0	11,627	165,169
2,048	57,653	2,946	0	0	4,464	33,599
4,096	8,691	0	0	0	1,878	857
8,192	1,145	0	0	0	795	14
16,384	83	0	0	0	256	1
32,768	15	0	0	0	99	0
65,536	2	0	0	0	52	0
131,072	0	0	0	0	39	0
262,144	0	0	0	0	5	0
524,288	0	0	0	0	3	0
1,048,576	0	0	0	0	2	0
2,097,152	0	0	0	0	1	0

(a) $\#len \leftrightarrow |len|$ Histogram

i	proteins	urls	dblp.xml	geographic	commoncrawl	vital
0	22	91	2	84	101	111
1	490	2,633	19	2,225	6,012	1,850
2	9,014	11,115	20	19,636	50,615	6,079
4	470,608	29,492	5	635,924	306,121	28,013
8	1,432,010	26,723	2,663	3,838,361	574,787	118,627
16	203,019	76,180	556,906	2,457,041	780,370	240,884
32	207,474	1,459,143	862,179	319,203	173,276	92,179
64	205,067	10,668,966	1,398,593	34,830	77,137	4,730
128	204,307	5,814,357	129,715	749	21,026	1,043
256	155,849	429,835	134	0	3,870	559
512	73,927	37,058	0	0	1,247	309
1,024	17,440	8,263	0	0	507	93
2,048	2,468	847	0	0	193	5
4,096	335	0	0	0	48	0
8,192	60	0	0	0	18	0
16,384	1	0	0	0	70	0
32,768	0	0	0	0	0	0
65,536	0	0	0	0	0	0
131,072	0	0	0	0	3	0

(b) $\#LCP \leftrightarrow |LCP|$ Histogram

tiny instances while becoming early slow after a number of insertions, while additionally consuming space for each node, even if they are empty — remember that we represent DicChild of all nodes belonging to the same micro trie *by a single* FNCS trie structure, cf. Sect. 2.3.

3.3. Other trie implementations

We compared c-trie♯ with keyword dictionary representations featuring also a low memory footprint. We present two groups of trie implementations. The first group consists of two non-compact trie data structures:

- DA: the double array [2] implementation of the Cedar library.[11]

[11] http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/.

Table 4
Histogram of (a) micro tries or (b) internal micro trie nodes storing a specific number of (a) child nodes or (b) internal nodes representing the sizes of (a) all DicHandle instances or (b) all DicChild instances. A (a) micro trie or (b) internal node is counted in the i-th row if the number of its stored nodes is i for $i = 1$ and $i = 2$, or in $[2^{i-2} + 1..2^{i-1}]$ for $i \geq 3$. None of the keyword sets is prefix-free, as can be seen by the fact that there are nodes with only a single child.

i	proteins	urls	dblp.xml	geographic	commoncrawl	vital
1	692,786	1,996,651	233,983	474,823	180107	57649
2	72,926	419,911	46,975	126,163	36273	13791
4	26,863	278,813	27,291	70,145	19255	8641
8	7,696	143,852	16,392	30,265	8500	4097
16	1,705	66,594	1,1386	13,357	3449	1651
32	420	27,161	6,411	6,424	1195	618
64	89	11,108	3,214	2,952	488	254
128	24	4,574	1,152	1,241	194	105
256	5	1,633	302	472	100	25
512	1	580	110	191	38	13
1024	1	75	37	68	37	3
2048	0	0	18	21	1	0
4096	0	0	8	9	0	0
8192	0	0	4	0	0	0
16384	0	1	0	1	0	0
32768	0	1	0	0	0	0
65536	0	0	0	0	0	1
131072	0	0	1	0	0	0
262144	0	0	0	0	0	0
524288	0	0	0	0	1	0
1048576	1	0	0	0	0	0

(a) #DicHandle ↔ |DicHandle|Histogram

i	proteins	urls	dblp.xml	geographic	commoncrawl	vital
1	27,933	189,554	106	204,565	30,276	279
2	1,220,896	3,939,539	808,559	1,644,531	468,330	164,154
4	231,439	1,594,225	313,011	716,500	175,809	54,646
8	86,483	886,825	116,020	288,994	69,654	19,579
16	42,571	507,437	53,258	104,526	47,619	6,272
32	13,894	34,609	14,298	28,445	6365	1,466
64	0	1,221	656	301	1,201	283
128	0	5	7	8	124	7

(b) #DicChild ↔ |DicChild| Histogram

- HAT-T: the HAT-trie [4] implementation of Tessil.[12] Tessil's implementation exploits that keywords have a small length in practice. The default implementation assumes that all these lengths can be stored in 16 bits, which is not true for the dataset commoncrawl. We therefore evaluated the HAT-trie with 16 and 32 bits for the lengths, and took the minimum time and minimum space of both variants throughout the evaluation.

As we will see in the following, the keyword dictionaries of the first group are lightweight and overall efficient but perform prefix searches poorly. The second group consists of other compact trie data structures:

- CT: a compact trie without word packing.
- PCT_{bit}: a packed c-trie using bit parallelism to compare compact words.
- PCT_{hash}: a packed c-trie using additionally the hash table implementation unordered_map of the C♯ standard library as a dictionary in each micro trie for retrieving a node by its extent (it is similar to our DicHandle, but uses the extents instead of the handles as keys).
- ZFT: our z-fast trie portation from an implementation in Java[13] to C♯. We added an evaluation of the original Java version to the appendix.

The implementations of the compact trie and the packed c-tries are due to Takagi et al. [39]. The implementations PCT_{bit} and PCT_{hash} pack characters in 32-bit integers, whereas all other implementations use 64-bit integers, which reflect the ma-

[12] https://github.com/Tessil/hat-trie.
[13] This implementation is part of Vigna's Sux4J library, located at https://github.com/vigna/Sux4J.

Table 5
Insertion of all keywords in *random* order. We measured (a) the average time per keyword and (b) the memory needed for inserting all keywords of the respective dataset. The (a) fastest time and the (b) lowest memory footprint for each keyword set and for each group of trie representations (compact tries or double array tries) are highlighted in bold font. For each instance, we measured the maximal virtual memory resident set size (VmRSS), which is the second integer in the file `/proc/self/statm`.

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	45,508.6	45,041.0	51,994.3	3,683.2	**2,349.2**	2,088.1	**1,805.5**
urls	13,459.0	10,580.3	8,659.0	**4,216.7**	4,646.1	2,702.8	**1,228.9**
dblp.xml	10,066.5	8,595.6	8,413.1	3,309.3	**3,035.1**	**1,202.8**	1,371.4
geographic	4,711.8	4,791.4	4,548.5	**2,223.4**	2,427.5	961.6	**595.6**
commoncrawl	11,077.5	11,029.6	12,269.6	2,368.5	**2,260.2**	904.9	**824.3**
vital	71,666.6	75,433.8	96,319.3	3,515.9	**2,002.2**	**1,869.5**	2,151.1

(a) Time in Nanoseconds

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	3,053.12	3,053.12	4,424.64	549.88	**418.10**	2,142.68	**892.66**
urls	8,551.47	8,551.48	9,465.49	3,731.14	**2,046.61**	**932.01**	1,317.75
dblp.xml	1,450.76	1,450.77	1,871.57	552.14	**305.74**	187.41	**144.77**
geographic	3,029.85	3,029.86	5,252.34	1,204.07	**719.36**	234.96	**164.29**
commoncrawl	1,040.79	1,040.77	1,685.80	330.03	**214.35**	269.03	**140.81**
vital	743.69	743.70	1,130.68	84.29	**58.12**	322.22	**239.12**

(b) Memory in Megabytes

Table 6
Average time for lookup(K) in nanoseconds. We created a list L storing all keywords $K \in \mathcal{K}$, and shuffled it. We measured the time of a linear scan over L during which we locate each visited keyword in the respective trie created in Table 5, and divided this time by $|\mathcal{K}|$, which yields the average times shown in this table.

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	42,199.7	33,678.0	20,011.6	2,530.4	**1,332.2**	1,413.3	**609.0**
urls	14,411.8	13,087.0	10,279.9	3,067.1	**2,801.6**	2,624.1	**559.1**
dblp.xml	10,454.9	8,990.6	6,869.7	2,205.5	**1,161.4**	989.6	**439.6**
geographic	4,764.3	5,016.1	2,726.1	1,449.5	**711.1**	423.0	**243.6**
commoncrawl	10,667.9	9,071.7	5,423.6	1,646.8	**742.4**	636.8	**299.6**
vital	71,552.6	52,391.1	29,774.7	2,806.9	**1,204.6**	1,138.9	**682.6**

chine word size of commodity computers nowadays. All implementations (of both groups) are written in C♯, and compiled with `gcc-8.2.0` in the highest optimization mode `-O3`.

In what follows, we evaluate these trie implementations on the aforementioned datasets.

3.4. Construction

In the first experiment, we measured the time it takes to insert all keywords of a dataset into a keyword dictionary in random order. We give the results in Table 5. This table reveals that the construction of c-trie♯ is faster than the construction of every packed trie (i.e., CT, PCT_{bit}, PCT_{hash}, and ZFT). Except for ZFT, its final size is also an improvement to the sizes of those data structures. If the average keyword length is sufficiently large, c-trie♯ is memory-friendlier than DA and HAT-T (e.g. `proteins` or `vital`) while it is inferior in both space and time when maintaining mostly short keywords.

3.5. Locate prefix queries

A major highlight is the time needed for locatePrefix(S) queries shown in Fig. 5. Instead of returning an iterator to a set as requested at the beginning of this article, we require each keyword dictionary to return the complete set of all keywords having S as a prefix. In this setting, c-trie♯ dominates most of the time. We observe that DA becomes faster for longer prefixes. This effect can be explained as follows: First recognize by Table 6 that DA has competitive lookup times, allowing the trie to match a pattern at high speed. The matching locates the lowest node v whose extent is a prefix of S. After locating v, it resorts to exploring the entire subtree of v, which is a slow operation for large subtrees. If v is a deep node, chances are that its subtree size is rather small, enabling DA to process v's subtree quickly.

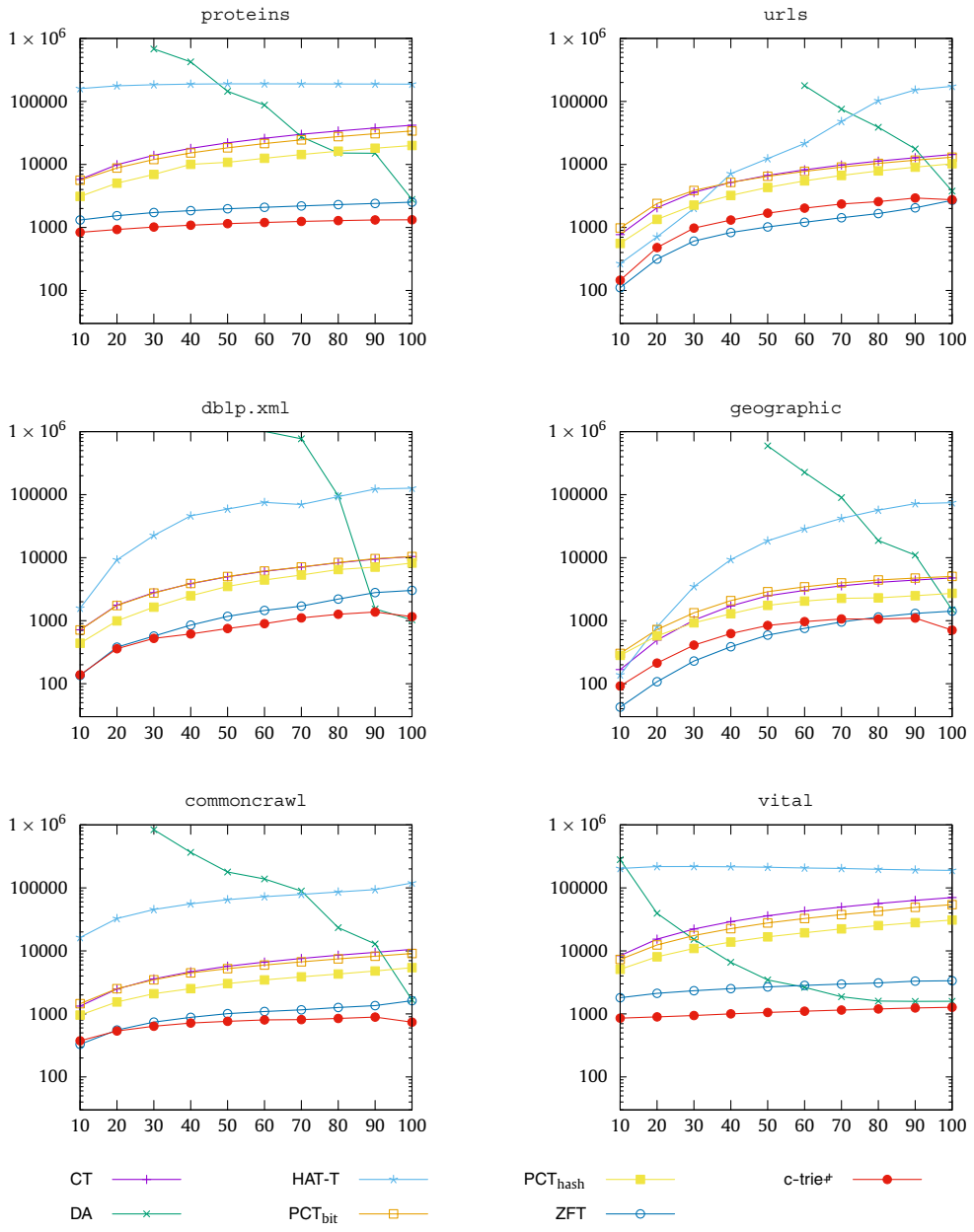

Fig. 5. Time for answering locatePrefix(S). The y-axis is the average time in nanoseconds (logarithmic scale) for one query. The x-axis is the prefix length (in percentage) of the original keyword lengths, i.e., we search the prefix of length $p|S|/100$ of S at the x-axis position p for each keyword S.

3.6. Lookup queries

The results for lookup are collected in Table 6. In all instances, c-trie♯ answered lookup queries faster than all packed tries. However, HAT-T, followed by DA, provide the fastest solutions for answering lookup.

3.7. Deletions

We also ran experiments for the delete operation, which we conducted in the same fashion as the experiments for lookup. We put the results in Table 7. There, we omit the implementations for CT, PCT_{bit}, and PCT_{hash} since they do not provide a delete operation. We observe that c-trie♯ is always faster than ZFT, but at most 3 times slower than DA, and 2 to 5 times slower than HAT-T.

Table 7
Average time for delete(K) in nanoseconds.

$\mathcal{K}$	ZFT	c-trie♯	DA	HAT-T
proteins	3,676.4	**2,012.0**	1,606.1	**1,187.7**
urls	5,677.7	**4,045.5**	3,060.8	**886.5**
dblp.xml	3,501.5	**2,219.4**	1,211.7	**667.4**
geographic	2,254.6	**1,761.8**	787.8	**494.3**
commoncrawl	2,526.7	**1,645.4**	868.5	**573.1**
vital	3,727.4	**1,780.8**	**1,042.0**	1,302.7

Table 8
Insertion of all keywords in *lexicographical* order. Except to the ordering of the keywords, the setting is the same as in Table 5.

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	39,716.6	38,547.4	48,384.0	2,623.4	**1,369.1**	1,225.3	**853.3**
urls	9,849.2	6,398.6	4,786.9	2,604.1	**709.5**	**610.9**	480.7
dblp.xml	7,736.4	5,713.0	5,645.8	2,051.3	**736.6**	**451.9**	810.4
geographic	2,342.1	2,089.6	2,605.7	1,305.1	**1,035.8**	**237.0**	258.3
commoncrawl	8,419.2	8,012.2	9,930.3	1,485.4	**1,072.3**	**370.3**	385.4
vital	63,719.1	65,684.8	90,066.2	3,187.2	**865.9**	1,313.2	**1,266.6**

(a) Time in Nanoseconds

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	2,889.31	2,889.32	4,376.2	549.87	**422.68**	1,779.47	**890.14**
urls	8,533.40	8,533.41	10,027.4	3,731.14	**2,046.45**	**1,017.18**	1,302.21
dblp.xml	1,445.39	1,445.40	1,850.2	552.14	**305.70**	173.62	**141.59**
geographic	3,029.50	3,029.51	4,952.8	1,204.07	**719.35**	251.86	**159.23**
commoncrawl	1,023.88	1,023.87	1,598.8	330.03	**220.18**	174.45	**139.61**
vital	695.96	695.97	1,098.4	84.29	**58.12**	261.09	**238.09**

(b) Memory in Megabytes

3.8. Sorted insertions

Up to now, we covered the case of creating a trie on keywords shuffled in a random order R, and subsequently queried the trie with the keywords in another random order R'. However, one might question whether other possibilities like building a keyword dictionary with lexicographically sorted keywords, or querying it with keywords arranged in the same order as in the construction is advantageous. For that, we revisit the construction in Table 8, filling a keyword dictionary now with keywords in lexicographically sorted order. Compared to Table 5, the space requirement in both scenarios is nearly the same for each keyword dictionary. However, a lexicographically sorted insertion speeds up the construction of all of instances. Especially the construction times of c-trie♯ seem to be order sensitive, as we now observe a large gap in the running times between c-trie♯ and ZFT. In the sorted insertion order, c-trie♯ is always the best among all packed trie representations, and for vital, it is even the overall best representation.

3.9. Order of queries

Having two scenarios for trie construction, we can also think about different orders of how to query the data structures. Here, we present a Cartesian product of these orders, shown in Table 9 for lookup, and in Figs. 6, 7, 8, and 9 for locatePrefix. We see a remarkable speedup of the query operations of all keyword dictionary implementations when they are fed with keywords in lexicographically order. The best bets can be placed on the setting of Table 9a and Fig. 6, where especially c-trie♯ shows a performance boost, being competitive to DA on some dataset instances for lookup (proteins or vital). A slightly slower variant is to query in random order (Table 9b and Fig. 7). The execution times of the keyword dictionaries fed in random order follow with a large gap. Here, the order in which the queries are executed has also an impact on the execution times, but is not as large as we have seen for the case where we inserted the keywords in lexicographic order. We obtain the fastest execution times when querying the keywords in lexicographic order (Table 9c and Fig. 8). Regarding the query order, the compact trie and the packed c-tries have roughly the same query times for different orders, unlike other trie data structure having a noticeable speed-up. Especially ZFT and c-trie♯ can take advantage of the case when the queries are in lexicographic order (Table 9c and Fig. 8). Finally, Table 10 complements Table 7 for the study of the

Table 9
Average time for lookup(K) in nanoseconds. We create a trie by inserting keywords contained a list L whose elements are (a-b) lexicographically sorted or (c-d) in a random order R. We stick to the setting of Table 9, where we used L for the queries. However, before the querying, we (b) shuffled L, (a,c) sorted the elements in L lexicographically, or (d) kept L as it is.

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	39,357.5	30,758.8	18,392.0	1,748.6	**421.1**	391.6	**256.0**
URLs	10,296.2	8,781.5	5,945.6	1,223.0	**240.6**	155.2	**138.7**
dblp.xml	7,957.8	6,372.7	4,481.0	1,121.3	**190.4**	**111.9**	136.0
geographic	1,839.2	1,925.0	1,436.4	717.3	**179.6**	**44.8**	65.4
commoncrawl	8,273.8	6,555.0	4,294.6	930.4	**169.7**	**95.9**	99.6
vital	69,059.5	49,871.5	28,850.4	2,046.7	**404.6**	526.3	**346.1**

(a) Sorted - Sorted

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	40,154.2	31,487.0	18,817.7	2,440.1	**1,155.6**	1,084.5	**627.3**
urls	10,725.0	9,287.6	6,376.8	**2,476.8**	2,470.9	1,739.7	**575.4**
dblp.xml	8,194.8	6,609.8	4,781.5	2,054.1	**1,084.9**	746.0	**459.7**
geographic	2,054.8	2,130.7	1,376.0	1,288.4	**636.4**	353.5	**246.9**
commoncrawl	8,697.6	6,892.4	4,220.5	1,575.9	**627.5**	470.1	**305.6**
vital	71,081.0	53,701.4	29,366.9	2,726.1	**1,111.8**	1,020.7	**681.9**

(b) Sorted - Order R

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	42,934.5	33,231.1	19,988.6	2,050.3	**805.6**	691.6	**309.3**
urls	14,563.1	12,500.3	9,321.5	1,598.1	**635.6**	361.4	**177.5**
dblp.xml	10,180.9	8,702.2	6,496.8	1,451.4	**473.0**	297.9	**161.5**
geographic	4,408.0	4,665.0	3,746.0	959.1	**407.6**	162.3	**84.9**
commoncrawl	10,370.6	8,761.5	6,016.1	1,175.2	**423.4**	267.0	**123.3**
vital	71,992.7	53,526.8	30,583.7	2,341.1	**778.6**	788.0	**411.4**

(c) Order R - Sorted

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie♯	DA	HAT-T
proteins	42,134.8	33,626.8	19,904.1	2,299.5	**1,016.8**	1,211.9	**605.4**
urls	14,329.6	13,008.3	10,187.0	2,462.7	**2,410.2**	2,491.6	**556.6**
dblp.xml	10,398.2	8,938.6	6,801.3	1,979.3	**920.32**	863.5	**436.3**
geographic	4,703.1	4,966.8	2,644.0	1,296.5	**501.7**	387.6	**240.4**
commoncrawl	10,624.8	9,040.9	5,353.6	1,496.8	**550.5**	553.6	**295.7**
vital	71,523.2	52,342.0	29,680.5	2,579.2	**943.7**	952.8	**665.1**

(d) Order R - Order R

delete operations regarding different orders, where the setting of Table 7 can be interpreted as inserting the keywords and querying all keywords in two different random orders R and R', respectively. Here, we observe similar characteristics to the study of lookup, with the difference that the performance gap between DA and c-trie♯ is unfortunately larger.

4. Conclusion

We have presented the trie data structure c-trie♯ to cope with the demands for fast prefix searches in practical applications such as auto-completion [11]. In settings where the amount of prefix queries is expected to be significantly larger than the amount of dynamic operations like insertions, the keyword dictionary c-trie♯ offers one of the best trade-offs among all tested candidates.

For future work, we can speed up the insertions of keywords that share long prefixes with other keywords by vectorization. That is because the word packing approach for comparing two strings interpreted as two packed strings can be vectorized. Recent instruction sets like AVX feature instructions for this task. An application[14] shows that the computation time roughly halves for long enough common prefixes when exploiting the AVX2 instruction set.

[14] https://github.com/koeppl/packed_string.

Table 10
Average time for delete(K) in nanoseconds under different orders. The setting with two different random orders R and R' (Order R - Order R') is already presented in Table 7, which has the same setting as Table 6. For the other sub-tables, the setting is given in Table 9.

$\mathcal{K}$	ZFT	c-trie♯	DA	HAT-T
proteins	2,457.1	**875.4**	**476.1**	568.3
urls	2,471.2	**526.4**	**211.6**	293.1
dblp.xml	2,059.1	**520.6**	**153.3**	236.3
geographic	1,161.2	**608.1**	**77.6**	143.4
commoncrawl	1,465.1	**575.0**	**129.5**	207.6
vital	2,704.5	**813.3**	**437.0**	733.8

(a) Sorted - Sorted

$\mathcal{K}$	ZFT	c-trie♯	DA	HAT-T
proteins	2,455.5	**874.1**	**476.0**	567.4
urls	2,476.7	**525.6**	**211.5**	292.6
dblp.xml	2,065.0	**522.4**	**153.4**	237.1
geographic	1,158.1	**608.3**	**77.6**	144.1
commoncrawl	1,561.8	**574.5**	**129.5**	207.5
vital	2,692.4	**814.1**	**435.7**	764.0

(b) Sorted - Order R

$\mathcal{K}$	ZFT	c-trie♯	DA	HAT-T
proteins	3,651.9	**2,012.4**	1,603.5	**1,187.7**
urls	4,257.0	**3,976.2**	3,051.1	**883.9**
dblp.xml	3,493.1	**2,141.1**	1,221.0	**671.1**
geographic	2,296.8	**1,750.3**	782.9	**494.6**
commoncrawl	2,513.5	**1,632.9**	875.8	**568.9**
vital	3,701.9	**1,781.7**	**1,026.0**	1,305.5

(c) Order R - Sorted

$\mathcal{K}$	ZFT	c-trie♯	DA	HAT-T
proteins	3,959.9	**2,006.1**	1,607.1	**1,194.2**
urls	4,340.2	**3,986.6**	3,061.0	**885.7**
dblp.xml	3,438.7	**2,147.2**	1,227.4	**668.4**
geographic	2,236.0	**1,765.64**	783.1	**493.7**
commoncrawl	2,516.1	**1,629.7**	878.6	**570.9**
vital	3,696.0	**1,779.0**	**1,041.6**	1,297.9

(d) Order R - Order R

Table 4a reveals that some instances of DicHandle grow extremely large while most of the other instances maintain only few entries. For the large ones, we can use a compact hash table such as [27][15] that stores quotients instead of the values, where a quotient has bit length $v - \lg M$ if the values can be represented in v bits (we set v to 32 bits in Sect. 2.3), where M is the number of cells of the hash table.

Considering different hash table layouts, we conducted an experiment with the linear probing hash table of Rigtorp[16] storing nodes along with the (redundant) keys. While using considerably more space, this hash table performed only slightly better than the cuckoo hash table, even when storing the keys explicitly and with a load factor of 0.5. Dropping the keys as we did in Sect. 2.3, a hash table with linear probing will likely be outperformed by our cuckoo hash table as cache effects become negligible.

As stated in the caption of Table 4, none of our datasets is prefix-free. In a more enhanced evaluation, we would like to conduct our experiments after a preprocessing step in which we discard every keyword that is a prefix of another keyword.

[15] We target to change this hash table to a bucketized cuckoo hash table like [14].
[16] https://github.com/rigtorp/HashMap.

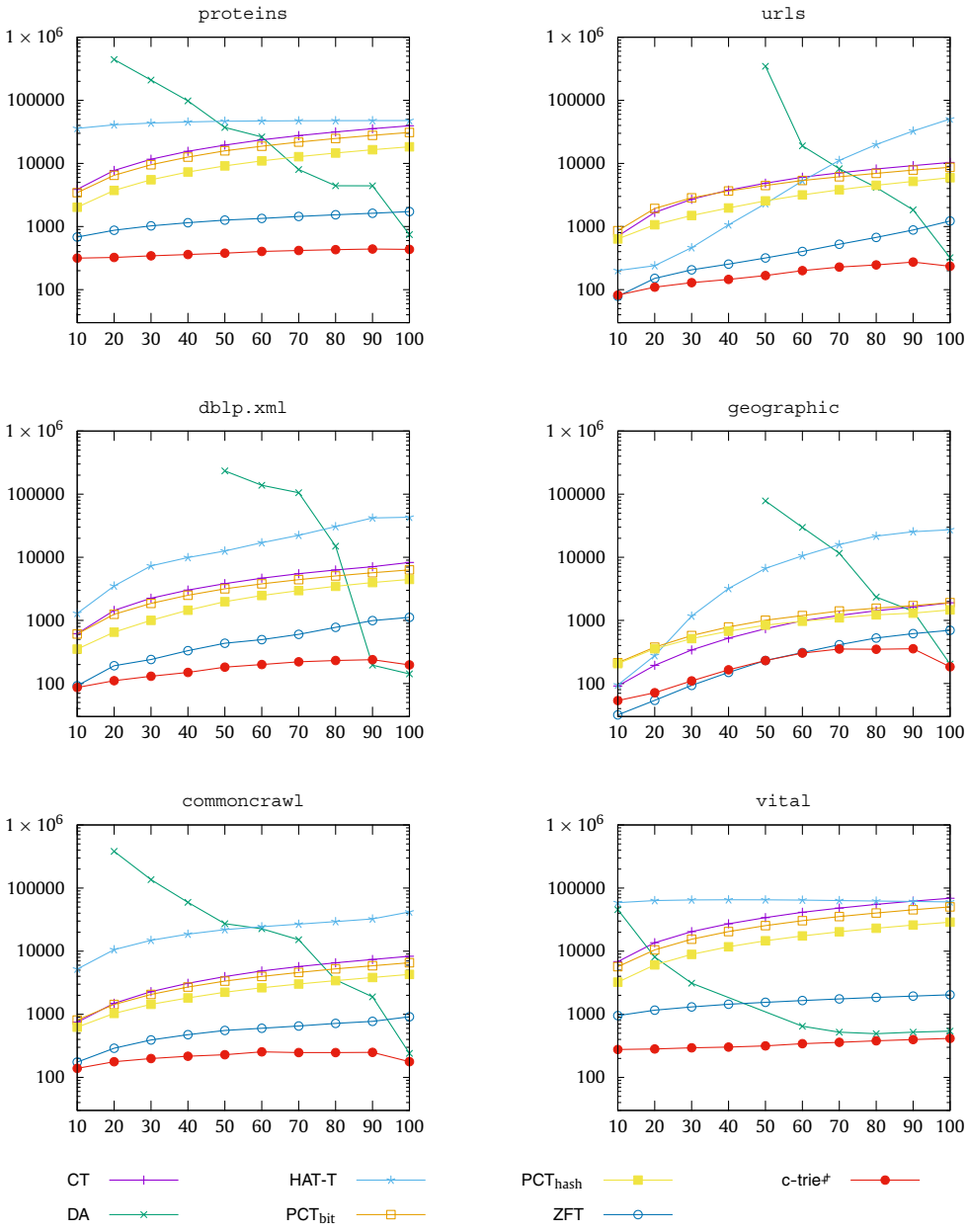

Fig. 6. Time for answering locatePrefix when the data structures are built and queried with the keywords in lexicographical sorted order. The setting is, except from the different order, the same as in Fig. 5.

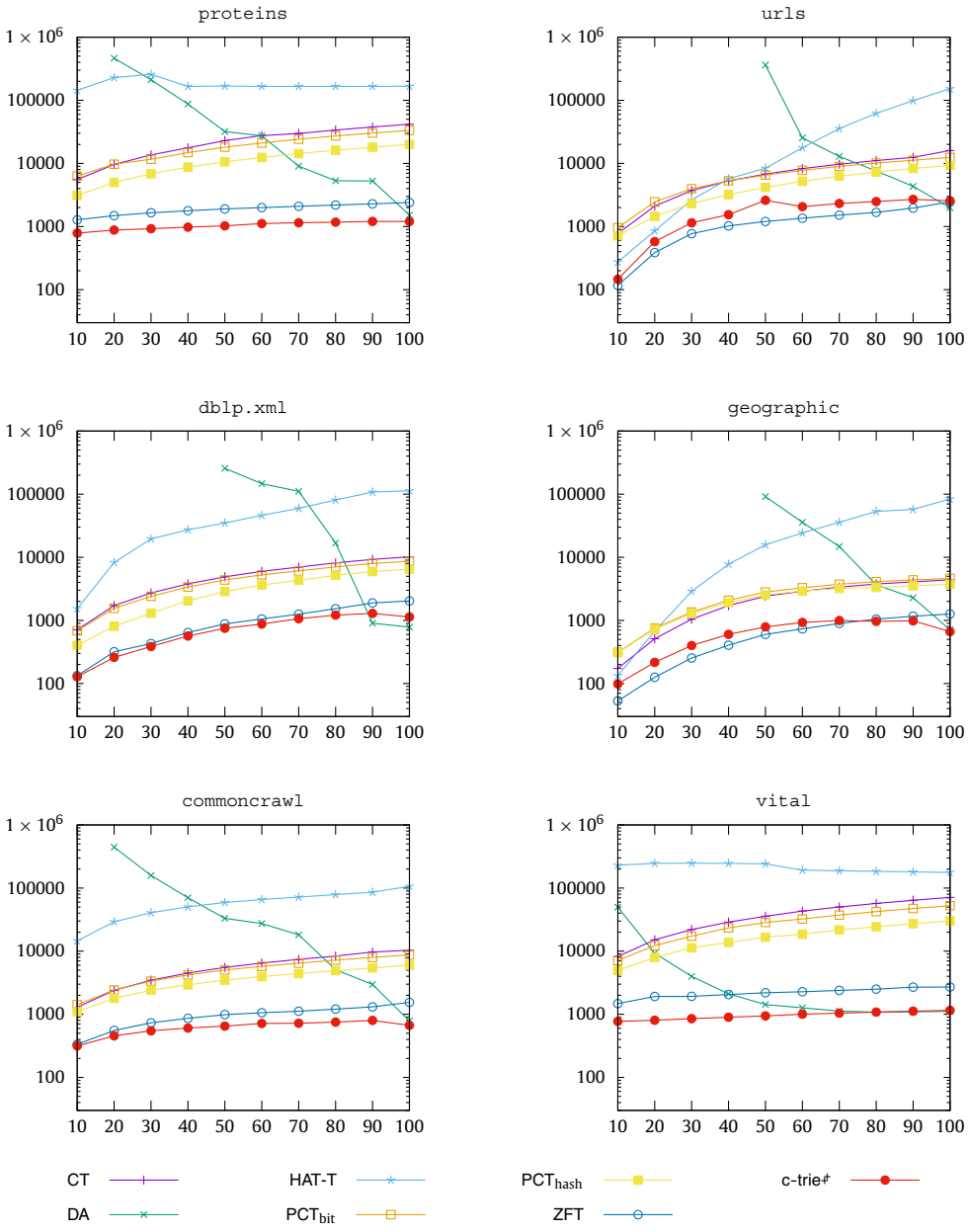

Fig. 7. Time for answering locatePrefix when the data structures are built with the keywords in lexicographical sorted order, but queried with the keywords in random order. The setting is, except from the different orders, the same as in Fig. 5.

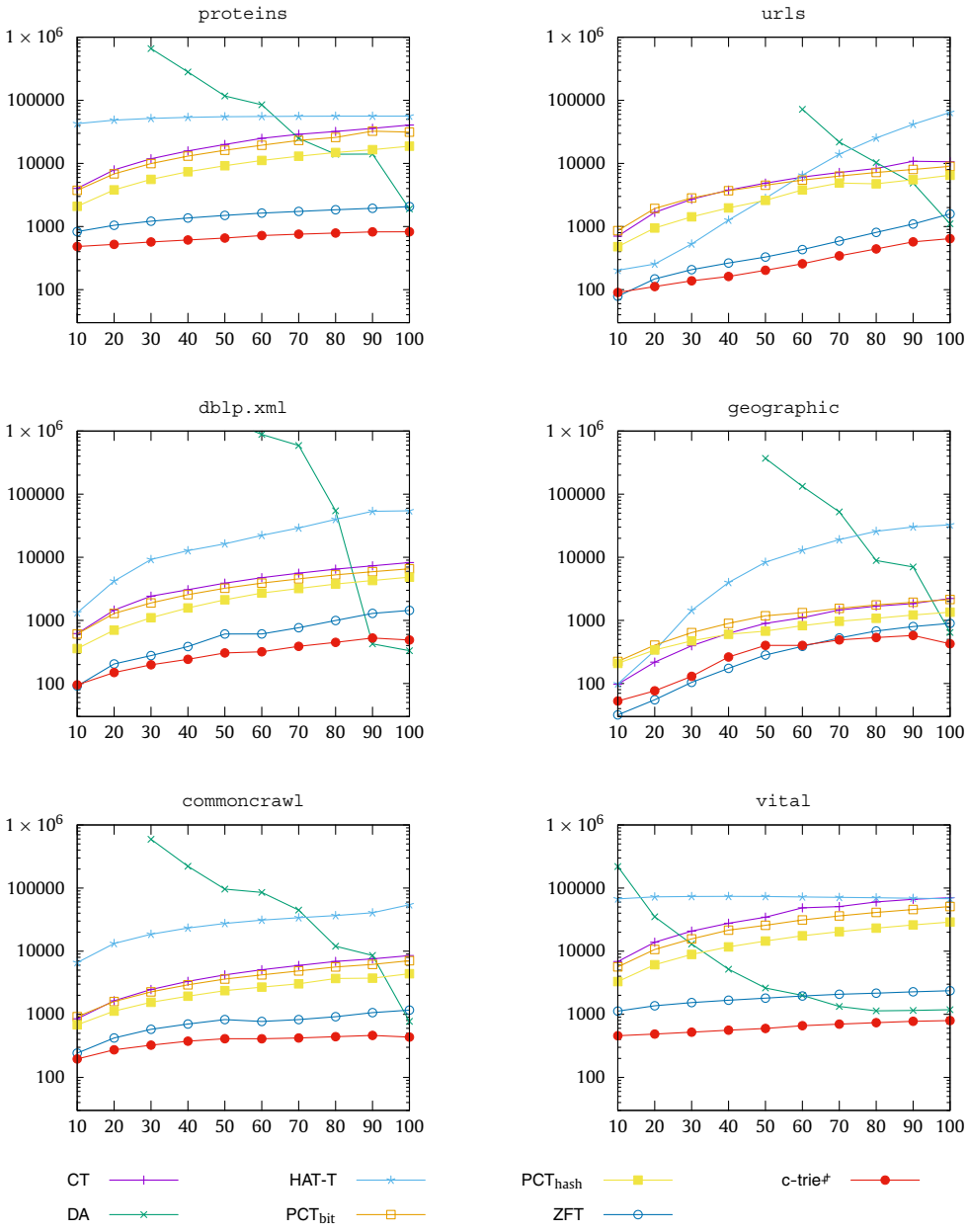

Fig. 8. Time for answering locatePrefix when the data structures are built with the keywords in random order, but queried with the keywords sorted in lexicographical order. The setting is, except from the different orders, the same as in Fig. 5.

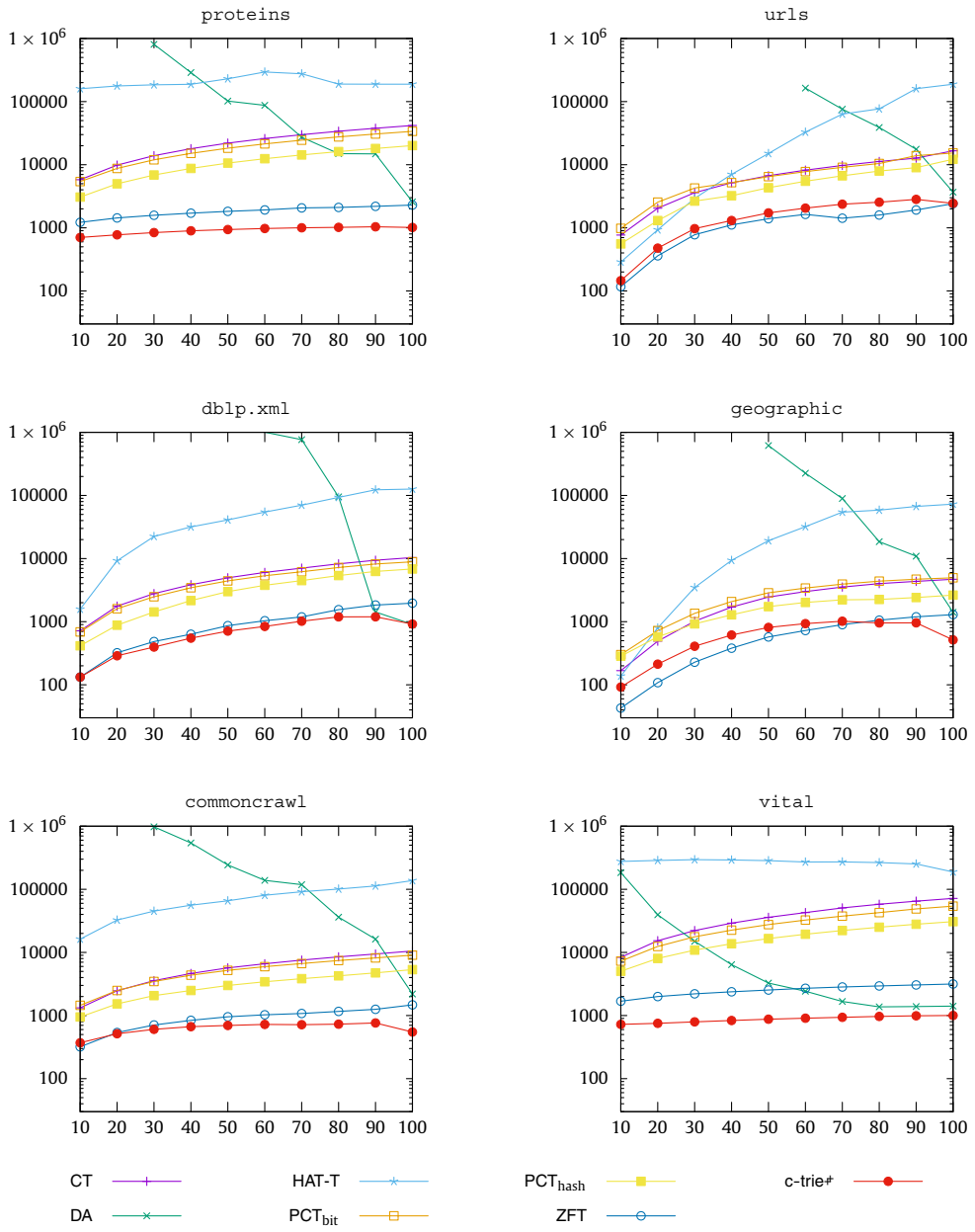

Fig. 9. Time for answering locatePrefix when the data structures are built with the keywords in a random order O, and queried with the keywords in the same order O. The setting is, except from the different order, the same as in Fig. 5.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers JP18F18120 (DK), JP21K17701 (DK), JP18K18002 (YN), JP17H01697 (SI), JP16H02783 (HB), JP20H04141 (HB), JP18H04098 (MT), and by JST PRESTO Grant Number JPMJPR1922 (SI).

Appendix A. Original z-fast trie

The original implementation of the z-fast trie of Vigna is written in Java as part of his Sux4J library. As a supplement, we conducted our experiments of this implementation on the same machine. However, we could not build this trie for the keyword set `vital`. The time and space needed for the trie construction are given in Table 11. Its time for lookup and locatePrefix are shown in Table 12 and Fig. 10, respectively. Its time for delete is given in Table 13. Unfortunately, we received runtime failures on several instances, which we marked with *N/A* (for not available) in the experiments.

Table 11
Inserting of all keywords in the z-fast trie Java-implementation.

$\mathcal{K}$	Random	Sorted
proteins	3,896.6	2,764.8
urls	3,056.7	2,038.7
dblp.xml	2,727.1	1,693.6
geographic	2,802.9	1,831.0
commoncrawl	2,883.3	1,714.9
vital	N/A	N/A

(a) Time in Nanoseconds

$\mathcal{K}$	Random	Sorted
proteins	1,629.60	1,630.81
urls	2,764.73	2,341.69
dblp.xml	989.38	1,026.62
geographic	1,043.65	1,075.91
commoncrawl	244.94	245.73
vital	N/A	N/A

(b) Memory in Megabytes

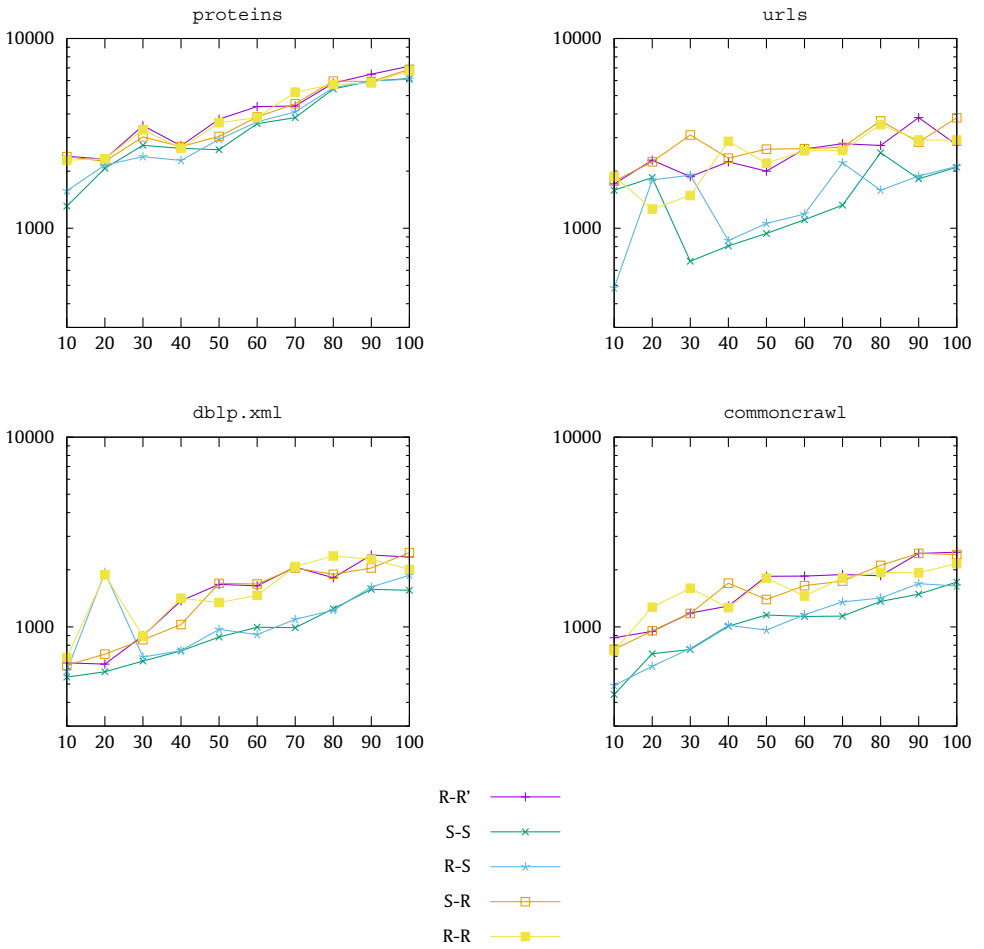

Fig. 10. Time for answering locatePrefix with the z-fast trie Java-implementation. The plots cover the settings of Figs. 5 (Order R - Order R'), 6 (Sorted - Sorted), 7 (Order R - Sorted), 8 (Sorted - Order R), and 9 (Order R - Order R), where R and R' are two different random orderings.

Table 12
Average time for answering lookup(K) with the z-fast trie Java-implementation. Times are in nanoseconds. The table covers the settings of Tables 6 (Order R - Order R'), 9a (Sorted - Sorted), 9c (Order R - Sorted), 9b (Sorted - Order R), and 9d (Order R - Order R), where R and R' are two different random orderings.

$\mathcal{K}$	R-R'	S-S	S-R	R-S	R-R
proteins	5,093.5	4,798.0	5,403.4	5,136.4	5,052.2
urls	2,384.2	1,655.2	2,615.3	1,730.9	2,438.1
dblp.xml	1,778.6	1,322.2	1,848.7	1,265.9	2,165.1
geographic	1,254.7	749.2	1,416.0	1,154.6	1,233.8
commoncrawl	2,032.3	1,351.9	1,870.8	1,404.8	1,648.0
vital	N/A	N/A	N/A	N/A	N/A

Table 13
Average time for answering delete(K) with the z-fast trie Java-implementation. The meaning of the column captions is the same as in Table 12.

$\mathcal{K}$	R-R'	S-S	S-R	R-S	R-R
proteins	5358.4	4173.3	4899.6	4205.4	4421.9
urls	4412.0	2904.0	3997.9	2848.4	4253.2
dblp.xml	N/A	N/A	N/A	N/A	N/A
geographic	2476.4	1223.1	1525.9	2690.6	2344.4
commoncrawl	N/A	N/A	N/A	N/A	N/A
vital	N/A	N/A	N/A	N/A	N/A

References

[1] A. Andersson, M. Thorup, Tight(er) worst-case bounds on dynamic searching and priority queues, in: Proc. STOC, 2000, pp. 335–342.
[2] J. Aoe, An efficient digital search algorithm by using a double-array structure, IEEE Trans. Softw. Eng. 15 (9) (1989) 1066–1077.
[3] J. Arz, J. Fischer, Lempel-Ziv-78 compressed string dictionaries, Algorithmica 80 (7) (2018) 2012–2047.
[4] N. Askitis, R. Sinha, Engineering scalable, cache and space efficient tries for strings, VLDB J. 19 (5) (2010) 633–660.
[5] N. Askitis, J. Zobel, Cache-conscious collision resolution in string hash tables, in: Proc. SPIRE, in: LNCS, vol. 3772, 2005, pp. 91–102.
[6] P. Beame, F.E. Fich, Optimal bounds for the predecessor problem and related problems, J. Comput. Syst. Sci. 65 (1) (2002) 38–72.
[7] D. Belazzougui, P. Boldi, S. Vigna, Dynamic z-fast tries, in: Proc. SPIRE, in: LNCS, vol. 6393, 2010, pp. 159–172.
[8] P. Bille, I.L. Gørtz, F.R. Skjoldjensen, Deterministic indexing for packed strings, in: Proc. CPM, in: LIPIcs, vol. 78, 2017, pp. 6:1–6:11.
[9] P. Bille, I. Li Gørtz, P. Gawrychowski, G.M. Landau, O. Weimann, Top tree compression of tries, arXiv:1902.02187, 2019.
[10] R. Binna, E. Zangerle, M. Pichl, G. Specht, V. Leis, HOT: a height optimized trie index for main-memory database systems, in: Proc. SIGMOD, 2018, pp. 521–534.
[11] F. Cai, M. de Rijke, A survey of query auto completion in information retrieval, Found. Trends Inf. Retr. 10 (4) (2016) 273–363.
[12] J.G. Cleary, Compact hash tables using bidirectional linear probing, IEEE Trans. Comput. 33 (9) (1984) 828–834.
[13] J.J. Darragh, J.G. Cleary, I.H. Witten, Bonsai: a compact representation of trees, Softw. Pract. Exp. 23 (3) (1993) 277–291.
[14] M. Dietzfelbinger, C. Weidling, Balanced allocation and dictionaries with tightly packed constant size bins, Theor. Comput. Sci. 380 (1–2) (2007) 47–68.
[15] P. Dinklage, J. Fischer, D. Köppl, M. Löbel, K. Sadakane, Compression with the tudocomp framework, in: Proc. SEA, in: LIPIcs, vol. 75, 2017, pp. 13:1–13:22.
[16] P. Ferragina, R. Grossi, A. Gupta, R. Shah, J.S. Vitter, On searching compressed string collections cache-obliviously, in: Proc. PODS, 2008, pp. 181–190.
[17] J. Fischer, D. Köppl, Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries, in: Proc. SPIRE, in: LNCS, vol. 10508, 2017, pp. 191–207.
[18] R. Grossi, G. Ottaviano, Fast compressed tries through path decompositions, ACM J. Exp. Algorithmics 19 (1) (2014).
[19] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[20] S. Heinz, J. Zobel, H.E. Williams, Burst tries: a fast, efficient data structure for string keys, ACM Trans. Inf. Syst. 20 (2) (2002) 192–223.
[21] B.P. Hsu, G. Ottaviano, Space-efficient data structures for top-k completion, in: Proc. WWW, 2013, pp. 583–594.
[22] G. Jacobson, Space-efficient static trees and graphs, in: Proc. FOCS, 1989, pp. 549–554.
[23] J. Jansson, K. Sadakane, W. Sung, Linked dynamic tries with applications to LZ-compression in sublinear time and space, Algorithmica 71 (4) (2015) 969–988.
[24] S. Kanda, D. Köppl, Y. Tabei, K. Morita, M. Fuketa, Dynamic path-decomposed tries, ACM J. Exp. Algorithmics 25 (1) (2020) 1.13:2–1.13:28.
[25] S. Kanda, K. Morita, M. Fuketa, Compressed double-array tries for string dictionaries supporting fast lookup, Knowl. Inf. Syst. 51 (3) (2017) 1023–1042.
[26] J. Kärkkäinen, E. Ukkonen, Sparse suffix trees, in: Proc. COCOON, in: LNCS, vol. 1090, 1996, pp. 219–230.
[27] D. Köppl, S.J. Puglisi, R. Raman, Fast and simple compact hashing via bucketing, in: Proc. SEA, in: LIPIcs, vol. 160, 2020, pp. 7:1–7:14.
[28] T. Kudo, T. Hanaoka, J. Mukai, Y. Tabata, H. Komatsu, Efficient dictionary and language model compression for input method editors, in: Proc. of the Workshop on Advances in Text Input Methods, 2011, pp. 19–25.
[29] D. Lemire, O. Kaser, N. Kurz, Faster remainder by direct computation: applications to compilers and software libraries, arXiv:1902.01961, 2019.
[30] A. Lovrencic, P.E. Black, Binary Tree Representation of Trees, U.S. National Institute of Standards and Technology, 2008.
[31] R. Mavlyutov, M. Wylot, P. Cudré-Mauroux, A comparison of data structures to manage uris on the web of data, in: Proc. ESWC, in: LNCS, vol. 9088, 2015, pp. 137–151.
[32] D.P. Mehta, S. Sahni, Handbook of Data Structures and Applications, Chapman & Hall/CRC Computer and Information Science Series, CRC Press, 2004.
[33] G. Navarro, Compact Data Structures – a Practical Approach, Cambridge University Press, 2016.
[34] R. Pagh, F.F. Rodler, Cuckoo hashing, J. Algorithms 51 (2) (2004) 122–144.
[35] G.E. Pibiri, R. Venturini, Efficient data structures for massive n-gram datasets, in: Proc. SIGIR, 2017, pp. 615–624.
[36] A. Poyias, R. Raman, Improved practical compact dynamic tries, in: Proc. SPIRE, in: LNCS, vol. 9309, 2015, pp. 324–336.
[37] R. Sedgewick, K. Wayne, Algorithms, Pearson Education, 2014.

[38] J. Sheldon, W. Lee, B. Greenwald, S.P. Amarasinghe, Strength reduction of integer division and modulo operations, in: Proc. LCPC, in: LNCS, vol. 2624, 2001, pp. 254–273.
[39] T. Takagi, S. Inenaga, K. Sadakane, H. Arimura, Packed compact tries: a fast and efficient data structure for online string processing, IEICE Trans. Fundam. Electron. Commun. Comput. Sci. 100-A (9) (2017) 1785–1793.
[40] K. Tsuruta, D. Köppl, S. Kanda, Y. Nakashima, S. Inenaga, H. Bannai, M. Takeda, c-trie++: a dynamic trie tailored for fast prefix searches, in: Proc. DCC, 2020, pp. 243–252.
[41] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[42] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann Series in Mult., Morgan Kaufmann Publishers, 1999.
[43] S. Yata, Dictionary compression by nesting Prefix/Patricia tries, in: Proc. of the 17th Annual Meeting of the Association for Natural Language, 2011.
[44] N. Yoshinaga, M. Kitsuregawa, A self-adaptive classifier for efficient text-stream processing, in: Proc. COLING, 2014, pp. 1091–1102.
[45] H. Zhang, H. Lim, V. Leis, D.G. Andersen, M. Kaminsky, K. Keeton, A. Pavlo, SuRF: practical range query filtering with fast succinct tries, in: Proc. SIGMOD, 2018, pp. 323–336.
[46] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory IT-23 (3) (1977) 337–343.