Delta-Fast Tries: Local Searches in Bounded Universes with Linear Space*

Marcel Ehrhardt† Wolfgang Mulzer†

Abstract

Let $U = \{0, 1, \ldots, 2^w - 1\}$ be a bounded universe. We present a dynamic data structure for predecessor searching in U that needs $O(\log\log\Delta)$ time for queries and $O(\log\log\Delta)$ expected time for updates, where Δ is the difference between the query element and its nearest neighbor in the structure. Our data structure requires linear space, improving a result by Bose *et al.*. It can be applied for answering approximate nearest neighbor queries in low dimensions.

1 Introduction

Predecessor searching is one of the oldest problems in theoretical computer science [3]. Let U be a totally ordered universe. The goal is to maintain a set $S \subseteq U$, $|S| = n$, while supporting *predecessor* and *successor* queries: given $q \in U$, find the largest element in S smaller than q (q's predecessor), or the smallest element in S larger than q (q's successor). In the *dynamic* version of the problem, we also want to be able to modify S by inserting and/or deleting elements.

In the *word-RAM model* of computation, all input elements are w-bit words, where $w \geq \log n$ is a parameter, and we are allowed to manipulate the input elements at the bit level. In this case, we may assume that the universe is $U = \{0, \ldots, 2^w - 1\}$. A classic solution for predecessor searching on the word-RAM is due to van Emde Boas, who described a dynamic data structure that requires space $O(n)$ and supports insertions and deletions in $O(\log\log U)$ time [6, 7].

In 2013, Bose *et al.* [2] described a word-RAM data structure for the predecessor problem that is *local* in the following sense. Let $q^+ := \min\{s \in S \mid s \geq q\}$ and $q^- := \max\{s \in S \mid s \leq q\}$ be the successor and predecessor of q. The structure by Bose *et al.* can answer predecessor and successor queries in $O(\log\log\Delta)$ time with $\Delta = \min\{|q - q^-|, |q - q^+|\}$. Their solution requires $\mathcal{O}(n \log\log\log|U|)$ words of space. Bose *et al.* apply their structure to obtain a fast data structure for approximate nearest neighbor queries in low dimensions.

We show how to reduce the space requirement to $O(n)$, while keeping the guarantees for the query times. This solves an open problem from [2], and also improves the space requirement for their nearest neighbor data structure. The full details can be found in the Master's thesis of the first author [5].

2 Preliminaries

Compressed Tries. Our data structure is based on *compressed tries* [3]. We interpret the elements from S as bitstrings of length w. The *trie* T' for S is a binary tree of height w. Each node $v \in T'$ corresponds to a bitstring $p_v \in \{0,1\}^*$. The root r has $p_r = \varepsilon$. For each inner node v, the left child u of v has $p_u = p_v 0$, and the right child w of v has $p_w = p_v 1$ (one of the two children may not exist). The bitstrings of the leaves correspond to the elements of S, and the bitstrings of the inner nodes are prefixes for the elements in S.

The *compressed trie* T for S is obtained from T' by contracting each maximal path of nodes with only one child into a single edge. Each inner node in T has exactly two children, and T has $O(n)$ nodes.

Let q be a bitstring of length at most w. The *longest common prefix* of q with S, $\mathrm{lcp}_S(q)$, is the longest prefix that q shares with an element in S. We say that q *lies on an edge* $e = (u, v)$ of T if p_u is a prefix of q and q is a proper prefix of p_v. If q lies on the edge (u, v), we call u the *least common ancestor* of q in T, denoted by $\mathrm{lca}_T(q)$. One can show that $\mathrm{lca}_T(q)$ is uniquely defined.

Associated Keys. Our algorithm uses the notion of *associated keys*. This notion was introduced for *z-fast tries* [1,10], and it is also useful in our data structure.

Associated keys provide a quick way to compute $\mathrm{lca}_T(q)$, for any element $q \in U$. A natural way to find $\mathrm{lca}_T(q)$ is to do binary search on the depth of $\mathrm{lca}_T(q)$: we initialize $(l, r) = [0, w]$ and let $m = (l + r)/2$. We denote by q_m the first m bits of q, and we check whether T has an edge $e = (u, v)$ such that q_m lies on e. If not, we set $r = m - 1$ and continue. Otherwise, we check if u is $\mathrm{lca}_T(q)$, by inspecting the other endpoint of e. If u is not $\mathrm{lca}_T(q)$, we set $l = m + 1$ and continue. In order to perform this search quickly, we need to find the edge e that contains a given prefix q_m. For this, we precompute for each edge e of T the first time that the binary search encounters a prefix that lies on e, and we let α_e be this prefix. We call α_e the *associated key* for $e = (u, v)$.

*Supported by DFG project MU/3501-1.

†Institut für Informatik, Freie Universität Berlin, Germany {marehr,mulzer}@inf.fu-berlin.de

The associated key can be computed in $O(1)$ as follows: consider the $\log w$-bit binary expansions $\ell_u = |p_u|_2$ and $\ell_v = |p_v|_2$ of the *lengths* of the prefixes p_u and p_v, and let ℓ' be the longest common prefix of ℓ_u and ℓ_v. Let ℓ be obtained by taking ℓ', followed by 1 and enough 0's to make a $\log w$-bit word. Let l be the number encoded by ℓ, and set α_e to the first l bits of p_v, see [5] for a detailed explanation.

Hash Maps. Our data structure also makes extensive use of hashing. In particular, we will maintain several succinct hashtables that store additional information for supporting fast queries. We will need the following theorem due to Demaine *et al.* [4].

Theorem 1 *For any $r \geq 1$, there exists a dynamic dictionary that stores entries with keys from U and with associated values of r bits each. The dictionary supports updates and queries in $O(1)$ time, using $O(n \log\log(|U|/n) + nr)$ bits of space. The bounds for the space and the queries are worst-case, the bounds for the updates hold with high probability.*

3 Static Δ-fast Tries

We begin by describing our data structure for the static case. The dynamic version will be discussed in the next section.

3.1 The Data Structure

Our data structure is organized as follows: We store S in a compressed trie T. The leaves of T are linked in sorted order. Furthermore, each node v of T stores pointers to the minimum and the maximum element in the subtree T_v of v. In addition to T, we maintain three hash maps H_Δ, H_z, and H_b.

We first describe the hash map H_Δ. Set $m = \log\log w$. For $i = 0, \ldots, m$, we let $h_i = 2^{2^i}$ and $d_i = w - h_i$. The hash map H_Δ stores the following information: for each $s \in S$ and each d_i, $i = 1, \ldots, m$, let $s_i = s_0 \ldots s_{d_i - 1}$ be the first d_i-bits of s and let $e = (u, v)$ be the edge of T such that s_i lies on e. Then, H_Δ stores the entry $H_\Delta[s_i] = u$.

Next, we describe the hash map H_z. It is defined similarly as the hash map used for z-fast tries [1, 10]. For each edge e of T, let α_e be the associated key of e (see Section 2). Then, H_z stores the entry $H_z[\alpha_e] = e$.

Finally, the hash map H_b is used to obtain linear space. It will be described below.

3.2 The Predecessor Query

Let $q \in U$ be the query, and let q^- and q^+ be the predecessor and the successor of q. We first show how to get a running time of $O(\log\log\Delta)$ for the queries, with $\Delta = |q - q^+|$. In Theorem 3, we will improve this to $\Delta = \min\{|q - q^-|, |q - q^+|\}$.

The predecessor search works in several *iterations*. In iteration i, we let q_i be the first d_i bits of q.

First, we check whether H_Δ contains an entry for q_i. If so, we know that T contains an edge e such that q_i lies on e, and that q_i is a prefix of $\mathrm{lcp}_S(q)$. We consider the two edges emanating from the lower endpoint of e, finding the e' that lies on the path to q (if the lower endpoint of e is $\mathrm{lca}_T(q)$, we are done). We take the associated key $\alpha_{e'}$ of e', and we use it to continue the binary search for $\mathrm{lca}_T(q)$, as described in Section 2. Since $|q_i| = d_i$, this binary search takes $O(\log(w - d_i)) = O(\log h_i)$ steps to complete. Once the lowest common ancestor $v = \mathrm{lca}_T(q)$ is available, we can find the predecessor of q in $O(1)$ additional time by inspecting the minimum and maximum elements in the subtrees for the two children of v and by using the pointers between the leaves in T. Details can be found in [5].

If H_Δ contains no entry for q_i and if q_i does not consist of all 1's, we check if H_Δ contains an entry for $q_i + 1$. Notice that $q_i + 1$ is the successor of q_i, e.g., if $q_i = 0000$, then $q_i + 1 = 0001$. If such an entry exists, first obtain $u = H_\Delta[q_i + 1]$, and the child v of u such that q_i lies on the edge $e = (u, v)$. Then, we find the minimum element in the subtree T_v. This is the successor q^+ of q. The predecessor q^- can be found by following the leaf pointers. This takes $O(1)$ time overall.

Finally, if both entries do not exist, we continue with iteration $i + 1$.

The total time for the predecessor query is $O(k + \log h_k)$, where k is the number of iterations and $\log h_k$ is the worst-case time for the predecessor search once one of the lookups in an iteration succeeds. By our predecessor algorithm, we know that S contains no element with prefix q_{k-1} or $q_{k-1} + 1$, but an element with prefix q_k or $q_k + 1$. Thus, we have $\Delta \geq 2^{h_{k-1}} = 2^{2^{2^{k-1}}}$, so $k \leq O(\log\log\log\Delta)$. Furthermore, since $h_k = (h_{k-1})^2$, it follows that $h_k = O(\log^2 \Delta)$.

3.3 Obtaining Linear Space

We now analyze the space requirement for our data structure. Clearly, the trie T and the hash map H_z require $O(n)$ words of space. Furthermore, as described so far, the space needed for H_Δ is $O(n \log\log w)$ words, since we store at most n entries for each height h_i, $i = 0, \ldots, m$.

Using a trick due to Pătraşcu [9], we can reduce the space requirement to linear. The idea is to store in H_Δ the *depth* d_u of each branch node u in T_Δ, instead of storing u itself. We then use an additional hash map H_b to obtain u.

This is done as follows: when trying to find the branch node u for a given prefix q_i, we first get the depth $d_u = |u|$ of u from H_Δ. After that, we look up the branch node $u = H_b[q_0 \ldots q_{d_u - 1}]$ from the hash

map H_b. Finally, we check whether u is actually the lowest branch node of q_i. If any of those steps fails, we return $\perp$.

Let us analyze the needed space: clearly, H_b needs $O(n)$ space, since it stores $O(n)$ entries. Furthermore, we have to store $O(n \log\log w)$ entries in H_Δ, each mapping a prefix q_i to the depth of its lowest branch node. This depth requires $\lceil \log w \rceil$ bits.

By Theorem 1, a retrieval only hash map for n' items and r bits of data is $O(n' \log\log \frac{|U|}{n'} + n'r)$ bits Therefore, the space for H_Δ is proportional to

$$n \log\log w \cdot \log\log \frac{|U|}{n \log\log w} + n \log\log w \cdot \lceil \log w \rceil$$
$$= O(n \log\log w \cdot \log w) = o(n \cdot w) \text{ bits},$$

where $n' = n \log\log w$, $r = \lceil \log w \rceil$ and $w = \log |U|$. Thus, we can store H_Δ in $O(n)$ words of w bits each. The following lemma summarizes the discussion

Lemma 2 *The Δ-fast trie needs $O(n)$ words space.*

3.4 Putting it Together

We can now obtain our result for the static predecessor problem.

Theorem 3 *The static Δ-fast trie solves the static predecessor problem with each operation in time $O(\log\log \min\{|q-q^-|, |q-q^+|\})$ and linear space. The preprocessing time is $O(n \log\log\log |U|)$ on sorted input.*

Proof. The normal search for $q \in S$ can be done in $O(1)$ time by a lookup in H_z. We have seen that the predecessor of q can be found in $O(\log\log |q - q^+|)$ time, and a symmetric result also holds for successor queries.

Similarly, we can achieve query time $O(\log\log |q - q^-|)$ by exchanging $H_\Delta[q_i - 1]$ with $H_\Delta[q_i + 1]$ in the query algorithm.

Therefore, by interleaving both searches, we obtain the desired running time of $O(\log\log \min\{|q-q^-|, |q-q^+|\})$. Of course, the pragmatic solution would be to add the case $H_\Delta[q_i - 1] \neq \perp$ to the query algorithm.

The preprocessing time is dominated by the time to fill the hash map H_Δ, since T and the hash maps H_z and H_b can be computed in linear time. Thus, the preprocessing time is $O(n \log\log\log |U|)$, because $\mathcal{O}(n \log\log w)$ nodes have to be inserted into H_Δ. The space requirement is linear (Lemma 2). □

4 Dynamic Δ-fast tries

We will now consider the dynamic case. For this, we need to explain how the update operations are implemented. Furthermore, we need a way to find and maintain the minimum and maximum element in each subtree of T. In the static case, this could be done by simply maintaining explicit pointers from each node $v \in T$ to the minimum and maximum element in T_v. In the dynamic case, we need a data structure which allows finding and updating these elements in in $O(\log\log \Delta)$ time.

4.1 Computing Lowest Common Ancestor

To perform the update operation, we need to be able to compute $\text{lca}_T(q)$ for any given element $q \in U$. For this, we will proceed as in the query algorithm from Section 3.2, but without the lookups for $H_\Delta[q_i - 1]$ and $H_\Delta[q_i + 1]$. By the analysis in Section 3.2, this will find $\text{lca}_T(q)$ in time $O(\log\log l)$, where l is height of $\text{lca}_T(q)$ in T.

Unfortunately, this height l might be as large as w. To get around this, we use a trick of Bose *et al.* [2]. Their idea is to perform a random shift of the universe. More precisely, we pick a random number $r \in U$, and we add r to all queries and update in the data structure (modulo $|U|$).

Lemma 4 (Lemma 4 in [2]) *After a random shift by r of U, the expected height of the lowest common ancestor of two fixed elements x and y is $O(\log |x-y|)$.*

Corollary 5 *Let $q \in U$, and let T be a randomly shifted Δ-fast trie. We can find $\text{lca}_T(q)$ in expected time $O(\log\log \Delta)$, where the expectation is over the random choice of the shift r.*

Proof. Set $x = q$, $y = q^+$ and let $\Delta = |q - q^+|$. We perform the doubly exponential search on the prefixes of q, as in Section 3.2 (without checking $q_i + 1$) to find the height h_k. After that, we resume the lowest common ancestor search on the remaining h_k bits. Since the number of remaining bits h_k is $O(\log \Delta)$ in expectation and by Jensen's inequality, the number of loop iterations k is $O(\log\log\log \Delta)$ in expectation. The expected running time is $k + \log h_k = O(\log\log \Delta)$. □

4.2 Managing the Minimum and Maximum Elements of the Subtrees

We also need to maintain the minimum and maximum elements in each subtree of T. In the static case, it suffices to have a pointer from a node to its minimal and maximal leaf, but in the dynamic case, we need an additional data structure.

For this, we use the fact that a given minimum (or maximum) leaf is common to at most w nodes of T. All these nodes form a subpath of a leaf-to-root path in T. Hence, if we maintain the nodes of this subpath in a concatenable queue data structure [8], we can obtain $O(\log w)$ update and query time to find the minimum (or maximum) element. However, we need

that the update and query time depend on the height h_i (i.e, the remaining bits) of the query. Thus, we partition the heights $\{0, 1, \ldots, w\}$ of a subpath into the sets $T_{-1} = \{0\}$, $T_i = [2^i, 2^{i+1})$, for $i = 0, \ldots, \log w - 1$, and $T_{\log w} = \{w\}$. Each of the sets is managed by a balanced binary tree, and all roots of those trees are linked together. The height of the i-th binary search tree is $\log |T_i| = O(i)$. Conversely, if a height h is given, the set $T_{\lfloor \log h \rfloor}$ is responsible for it.

Furthermore, T_{-1} is a leaf (the depth of that node is w) in the trie and therefore the minimum of the whole subpath. Thus, the minimum of a subpath can be found from a given node $v \in T_i$ in $O(i)$ time by following the pointers to the root of T_i and the pointers down to T_{-1}.

If a node v has $h_k = O(\log \Delta)$ remaining bits, the node is within the tree $T_{\lfloor \log h_k \rfloor}$. Thus, it takes $O(\log h_k) = O(\log \log \Delta)$ time to find the minimum. Furthermore, we can split and join the subpaths represented in this way in $O(\log h_k)$ time, where h_k is the height of the node where the operation occurs. Details can be found in [5].

4.2.1 Performing an Update

We know from the Lemma 4, that the lowest common ancestor has expected height $h_k = O(\log \Delta)$.

Lemma 6 *Inserting or deleting an element q into a Δ-fast trie takes $\mathcal{O}(\log \log \Delta)$ expected time, where the expectation is over the random choice of r.*

Proof. Inserting q into T splits an edge (u, v) of T into two edges (u, b) and (b, v). This creates two new nodes in T, a branch node and a leaf. The branch node is $\mathrm{lca}_T(q)$, and it has expected height $h_k = O(\log \Delta)$. So, finding it will take $O(\log \log \Delta)$ expected time, by Corollary 5.

The hash maps H_z and H_u can be updated in constant time. Now let us consider the update time of the hash map H_Δ. Remember that H_Δ stores the lowest branch nodes for all prefixes of the elements in S that have certain lengths. That means that all prefixes on the edge (b, v) which are stored in the hash map T_Δ need to be updated. Furthermore, prefixes at certain depths which are on the edge (b, q) need to be added. Note that for the edge (b, v), we will enumerate all prefixes at certain depths, but only select those that are on the edge. We will argue that a leaf-to-b path needs $O(\log \log \log \Delta)$ insertions or updates: We have to insert $Q_i := q_0 \ldots q_{d_i}$ for all $i = 1, \ldots$ until $d_i < |b|$. Since $d_i = w - h_i$, $|b| = w - O(\log \Delta)$ and $h_i = 2^{2^i}$, $c \log \Delta < 2^{2^i}$ when $i > \log \log(c \log \Delta)$. Thus, $i = \Theta(\log \log \log \Delta)$.

After that, the minimum and maximum elements for the subtrees of T have to be updated. This may update a subpath at a node of height $h_k = O(\log \Delta)$. As we have seen, this takes $O(\log h_k) = O(\log \log \Delta)$ time. Deletions are performed similarly. □

The following theorem summarizes our result.

Theorem 7 *Suppose we perform a random shift of U by r. Then, the Δ-fast trie solves the dynamic predecessor problem such that query operations take $O(\log \log \Delta)$ worst-case time and update operations take $O(\log \log \Delta)$ expected time. Here, $\Delta = \min\{|q - q^+|, |q - q^-|\}$, where $q \in U$ is the argument for the current operation and q^+ and q^- are the predecessor and successor of q in the current set. At any time, the data structure needs $O(n)$ words of space, where n is the size of the current set.*

References

[1] D. Belazzougui, P. Boldi, and S. Vigna. Dynamic z-fast tries. In *Proc. 17th Int. Symp. String Processing and Information Retrieval (SPIRE)*, pages 159–172, 2010.

[2] P. Bose, K. Douïeb, V. Dujmovic, J. Howat, and P. Morin. Fast local searches and updates in bounded universes. *Comput. Geom. Theory Appl.*, 46(2):181–189, 2013.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, third edition, 2009.

[4] E. D. Demaine, F. Meyer auf der Heide, R. Pagh, and M. Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Proc. 7th LATIN*, pages 349–361, 2006.

[5] M. Ehrhardt. An in-depth analysis of data structures derived from van-Emde-Boas-trees. Master's thesis, Freie Universität Berlin, 2015. `http://www.mi.fu-berlin.de/inf/groups/ag-ti/theses/download/Ehrhardt15.pdf`.

[6] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6(3):80–82, 1977.

[7] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10(2):99–127, 1976.

[8] F. P. Preparata and M. I. Shamos. *Computational geometry. An introduction*. Springer-Verlag, 1985.

[9] M. Pătraşcu. vEB space: Method 4. `http://infoweekly.blogspot.de/2010/09/veb-space-method-4.html`, 2010.

[10] M. Ružić. Making deterministic signatures quickly. *TALG*, 5(3):26:1–26:26, 2009.