Engineering Predecessor Data Structures for Dynamic Integer Sets

Patrick Dinklage
TU Dortmund University, Germany

Johannes Fischer
TU Dortmund University, Germany

Alexander Herlez
TU Dortmund University, Germany

Abstract

We present highly optimized data structures for the dynamic predecessor problem, where the task is to maintain a set S of w-bit numbers under insertions, deletions, and predecessor queries (return the largest element in S no larger than a given key). The problem of finding predecessors can be viewed as a generalized form of the membership problem, or as a simple version of the nearest neighbour problem. It lies at the core of various real-world problems such as internet routing.

In this work, we engineer (1) a simple implementation of the idea of universe reduction, similar to van-Emde-Boas trees (2) variants of y-fast tries [Willard, IPL'83], and (3) B-trees with different strategies for organizing the keys contained in the nodes, including an implementation of dynamic fusion nodes [Pătraşcu and Thorup, FOCS'14]. We implement our data structures for $w = 32, 40, 64$, which covers most typical scenarios.

Our data structures finish workloads faster than previous approaches while being significantly more space-efficient, e.g., they clearly outperform standard implementations of the STL by finishing up to four times as fast using less than a third of the memory. Our tests also provide more general insights on data structure design, such as how small sets should be stored and handled and if and when new CPU instructions such as advanced vector extensions pay off.

2012 ACM Subject Classification Theory of computation → Predecessor queries

Keywords and phrases integer data structures, dynamic data structures, predecessor, universe reduction, y-fast trie, fusion tree, B-tree

Digital Object Identifier 10.4230/LIPIcs.SEA.2021.7

Related Version *Full Version*: `https://arxiv.org/abs/2104.06740`

Supplementary Material *Software*: `https://github.com/pdinklag/tdc/tree/sea21-predecessor` archived at `swh:1:dir:40d4000cd5f3302b2d037401280dcf1bea2c8b31`

Funding *Patrick Dinklage*: Supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).
Alexander Herlez: Supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).

1 Introduction

Finding the predecessor of an integer key in a set of keys drawn from a fixed universe is a fundamental algorithmic problem in computer science at the core of real-world applications such as internet routing [8]. It can be considered a generalized form of the membership problem or a simple version of the nearest neighbour problem. Navarro and Rojas-Ledesma [20] recently gave a thorough survey on the topic, recapping the past four decades of research.

Data structures for the predecessor problem are designed to beat the $\Omega(\lg n)$ lower time bound for comparison-based searching. While optimal data structures have been shown

for static sets [21] that are known in advance and do not change, they do not necessarily translate to the most practical implementations. Dinklage et al. [10] face the symmetrical *successor* problem for a small universe and develop a simple data structure that accelerates binary search. Despite not optimal in theory, it is the most efficient in their setting.

In this work, we focus on the *dynamic* problem, where the set of integers can be changed at any time by inserting, deleting or updating keys. A prominent example of a dynamic predecessor data structure is the *van Emde Boas tree* [24], which, despite near-optimal query times in theory, has been proven irrelevant in practice due to its memory consumption [25]. Dementiev et al. [9] implemented a *stratified trie* as a heavily simplified practical variant of the van Emde Boas tree for keys drawn from a 32-bit universe. Nowadays, with 64-bit architectures dominating the landscape, the limitation to 32-bit keys can be considered significant. The authors gave no hints as to how the data structure can be altered to properly handle larger universes, and simply applying the same structure on a larger universe exceeds practical memory limitations quite quickly. Nash and Gregg [19] thoroughly evaluated various dynamic predecessor data structures in practice, including the aforementioned stratified tree. They also implemented AVL trees, red-black trees and B-trees, as well as the trie hybrid by Korda and Raman [17] and their self-engineered adaptation of *burst tries* [14] for integer keys, which outperform the other data structures regarding both speed and memory usage.

Our contributions. We engineer new practical solutions for the dynamic predecessor problem that are both faster and more memory efficient than the current best known to us. First, we apply the idea of *universe sampling* following [10] to the dynamic case. Second, we engineer *y-fast tries* [26], which, in our view, offer room for many practical optimizations. Finally, we implement *dynamic fusion nodes* [22], for which Pătraşcu and Thorup give a very practical description but no implementation. We embed them into B-trees and make use of modern CPU instructions to accelerate some key low-level operations.

We note that our data structures are designed in a way often not optimal in theory. A recurring observation that we made is that thanks to large CPU caches, naïve solutions for queries on small datasets often outperform sophisticated data structures on modern hardware, including linear scans of unsorted lists, or binary search in naïvely organized sorted lists, where updates potentially require all items to be shifted. This observation has been confirmed in the contexts of balanced parentheses [4, 11] and finding longest common extensions in strings [10, 15]. We make use of this and replace predecessor data structures for small input sets by sorted or unsorted lists without any auxiliary information.

This paper is organized as follows: we begin with definitions and notations in Section 2 and a description of our experimental setup in Section 3. Then, in Sections 4–6, we describe our engineered data structures and give individual experimental results. In Section 7, we conclude with a comparison of the best configurations with existing implementations.

2 Preliminaries

Let S be a set of $n = |S|$ positive integers called *keys* drawn from a fixed universe $U := [u] = [0, u-1]$. For any $x \in U$, we call $\text{pred}_S(x) = \max\{y \in S \mid y \leq x\}$ the *predecessor* of x, which is the largest key in S no larger than x. We consider the dynamic scenario, where keys may be inserted into or deleted from S and the data structure must be updated accordingly.

In our analysis, we use the word RAM model, where we assume that we can perform arithmetic operations on words of size $w = \Theta(\lg u)$ in time $\mathcal{O}(1)$ (by default, logarithms are to the base of two). Additionally, the binary logic operations OR ($\vee$), AND ($\wedge$) and XOR

($\oplus$) on words take constant time. With this, we can access the i-th bit in a word x, denoted by $x\langle i\rangle$, as well as the bits i to j (inclusively) of x, denoted by $x\langle i\,..\,j\rangle$, in constant time. We refer to bit positions in MSBF order, e.g., $x\langle 0\rangle$ is the most significant bit of x.

We also require some advanced operations on words to be answered in constant time. Let $\mathrm{msb}(x)$ denote the position of the most significant set bit of x and $\mathrm{select}_1(x, k)$ the position of the k-th set bit in x. Another needed operation is counting the number of trailing zero bits of x. In theory, these queries can be answered in constant time using the folklore approach of precomputing universal tables of size $o(u^{1/c})$ bits, where we can look up the answers for all possible queries on a constant number of $c > 1$ blocks of size w/c. In practice, we can make use of special CPU instructions: LZCNT and TZCNT count the number of leading or trailing zeroes, respectively, and POPCNT reports the number of set bits in a word. These instructions are fairly widely spread, being implemented by current versions of the x86-64 (both Intel [16] and AMD [1]) instruction sets as well as ARM [3].

Tries. Tries are a long-known information retrieval data structure [12]. Here, we consider *binary* tries for strings over a binary alphabet. Consider a key $x \in U$ to be inserted into a binary trie. We navigate the trie top-down according to the bits of the binary representation of x in MSBF order: when reading a 0-bit, we go to the current node's left child, and otherwise to the right child. Inserting a new element x works accordingly, creating any node that does not yet exist. Since the binary trie for a set of keys drawn from U has height $\lceil \lg u \rceil$, this takes total time $\mathcal{O}(\lg u)$. An example of a binary trie is shown in Figure 5a. Binary tries are suitable for solving the dynamic predecessor problem: to find $\mathrm{pred}_S(x)$, we navigate down the trie as if we were to insert it. If we reach a leaf labeled x, then $x \in S$ and it is its own predecessor. Otherwise, we eventually reach an inner node v that is missing the left or right edge that we want to navigate, respectively. If the right edge is missing, the predecessor of x is the label of the rightmost leaf in the left subtrie of v. Otherwise, if the left edge is missing, we first navigate back up to the lowest ancestor v' of v that has two children and where v is in the right subtrie; then the predecessor is the label of the rightmost leaf in the left subtrie of v'. In either case, we can report the predecessor of x in time $\mathcal{O}(\lg u)$. Deleting is done by locating a key's leaf, removing it, and navigating back up removing any inner node no longer connected to any leaves, all in time $\mathcal{O}(\lg u)$. The number $\mathcal{O}(n \lg u)$ of nodes in the binary trie can be reduced to $\mathcal{O}(n)$ by contracting paths of branchless inner nodes to single edges [18]. We call a trie *compact* if it does not contain any inner nodes with one child.

3 Methodology

We conduct the following three-step experiment for our data structures:

(1) insert n keys drawn uniformly at random from U into the initially empty data structure,
(2) perform *ten million* random predecessor queries for keys in the range of the inserted keys, guaranteeing that there is always a predecessor that is never trivially the maximum, and
(3) delete the n keys from the data structure in the order in which they were inserted.

In preliminary experiments, we also considered distributions other than uniform, and also intermingling insertions, queries and deletions. Apart from statistical fluctuations, the results led to the same assertions and thus we solely consider the experiment described above.

For each data structure, we run five iterations using a different random seed each (but the same seeds for all data structures and in the same order). We measure running times by the wall clock time difference between start and finish of an iteration, as well as the RAM usage using custom overridden versions of `malloc` and `free` and compute the averages

over the five iterations. Our code is written in C++17 and publicly available[1]; we compile using GCC version 9.3. For hash tables (Sections 4 and 5), we use a public[2] implementation of Robin Hood hashing [6] that is both faster and more memory efficient than the STL implementation (`std::unordered_map`). We conduct our experiments on Linux machines with an Intel Xeon E5-4640v4 processor (12 cores at 2.1 GHz, 12×32 kB L1, 12×256 kB L2, 30 MB L3 shared, line size 64 B) and 256 GB of RAM.

4 Dynamic Universe Sampling

A common technique used by predecessor data structures is known as *length reduction* [5], where we partition the universe into buckets of size $b \ll u$ and reduce the predecessor problem to the much smaller sub-universe $[b]$. For each bucket, we determine a representative, e.g., the minimum contained key, which is entered into a *top level* predecessor data structure. The buckets are maintained on the *bucket level*. When $\text{pred}(x)$ is queried for some $x \in U$, we first solve the predecessor problem on the top level to find the bucket that x belongs into, and then reduce the query to a smaller one answered on the bucket level. The van Emde Boas tree [24] applies this approach recursively. In this work, we develop a dynamic version the two-level data structure by Dinklage et al. [10] that achieved very good practical results for the *static* predecessor problem.

We partition U into buckets of size $b = 2^k$ for some $k > 0$. Let $i \in [u/b]$, then the i-th bucket can only contain keys from the interval $[bi, b(i+1) - 1]$. We call a bucket *active* if it contains at least one key from S. Since $b = 2^k$, the number i of the bucket that a key $x \in U$ belongs into is the number represented by the $\lceil \lg u \rceil - k$ highest bits of x. Hence, we only store the lowest k bits for each key to reduce space usage in the buckets.

Top level. The top level maintains the set of active buckets. Consider a query for $\text{pred}_S(x)$, then it reports the *rightmost* active bucket i such that $bi \leq x$. The predecessor of x is then contained in bucket i if x is greater than the bucket's current minimum. Otherwise, it is the current maximum key contained in the active bucket preceding i. Clearly, the top level requires a dynamic predecessor data structure on the set of active buckets, i.e., keys drawn from the universe $[u/b]$ represented by the keys' high bits. We explore two basic options. First, let $i_{\min}$ and $i_{\max}$ be the numbers of the leftmost and rightmost active buckets, respectively. We store $i_{\max} - i_{\min} = \mathcal{O}(u/b)$ pointers in an array such that the i-th entry points to the rightmost active bucket i' with $i' \leq i$. Predecessor queries can trivially be answered in time $\mathcal{O}(1)$ using a lookup, but updates may take time $\mathcal{O}(u/b)$ in the worst case as we may need to shift pointers and/or update pointers for succeeding non-active buckets. Furthermore, the array requires up to $\lceil (u/b) \lg(u/b) \rceil$ bits of space. Our alternative is a hash table H containing only pointers to active buckets, identified by their numbers. Let b' be the number of active buckets, then H requires $\mathcal{O}(b' \lg(u/b))$ bits of space. Updates can be done in $\mathcal{O}(1)$ expected time, but since the order of buckets in H is arbitrary, queries may require to perform up to b' lookups: when a key belongs in bucket i, we look up i in H; if that bucket is not active, we find no result and look up $i - 1$, and so on. This takes up to $\mathcal{O}(b')$ expected time.

[1] Our code is published at `https://github.com/pdinklag/tdc/tree/sea21-predecessor`. Make sure to check out the *sea21-predecessor* branch, which contains instructions in the readme.
[2] Robin Hood hashing by Martin Ankerl: `https://github.com/martinus/robin-hood-hashing`.

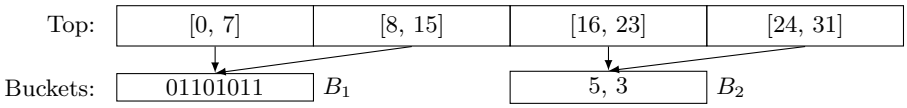


Figure 1 Hybrid universe sampling data structure for $S = \{1, 2, 4, 6, 7, 19, 21\}$ with $w = 5$, $b = 8$ and $\theta_{\min} = \theta_{\max} = 3$. The top level holds bucket pointers for the partitioned universe. Since there are no keys from the intervals $[8, 15]$ and $[24, 31]$ contained in S, their pointers point to the respective preceding buckets. Bucket B_1 is represented as a bit vector of length b such that each 1-bit corresponds to a key contained in S. Bucket B_2, on the other hand, only contains two keys that are represented as an unsorted list of keys relative to the left interval boundary.

Bucket level. On the bucket level, we first look at two basic data structures. We only store the lowest $k = \lg b$ bits of the contained keys, called *truncated keys* in the following, as the high bits are already defined by the bucket number. Let $S_i \subseteq S$ be the set of truncated keys contained in the i-th bucket. We can store them in a bit vector $B_i \in \{0, 1\}^b$ where $B_i[x] = 1$ if $x \in S_i$ and $B_i[x] = 0$ otherwise. Updates are then done in constant time by setting or clearing the respective bit in B_i. To compute $\text{pred}_{S_i}(x)$, we scan B_i linearly in time $\mathcal{O}(b/\lg u)$ using word packing. However, B_i always requires b bits of space. Let n' be the *current* number of keys contained in a bucket and consider the case where $n' < b/\lg b$. An alternative is storing an unsorted list of n' keys: this requires only $n' \lg b < b$ bits of space and retains $\mathcal{O}(1)$ time insertions, and predecessor queries and deletions take time $\mathcal{O}(n') = \mathcal{O}(b/\lg b) = \mathcal{O}(b/\lg u)$.

We now consider a hybrid of the two basic bucket structures. Let $\theta_{\min}$ and $\theta_{\max}$ be thresholds with $0 < \theta_{\min} \leq \theta_{\max} < b$. We maintain a bucket as an unsorted list of keys as long as $n' < \theta_{\max}$. If, after inserts, the bucket grows beyond $\theta_{\max}$ keys, we rebuild it to a bit vector. If, after deletions, the bucket size falls below $\theta_{\min}$ keys, we revert to an unsorted list. Rebuilding the bucket to a bit vector or unsorted list, respectively, takes time $\mathcal{O}(b)$. Let $\theta_{\min} := cb/\lg b$ and $\theta_{\max} := \theta_{\min} + c' \lg b$ for constants $c, c' > 0$. Then, $\Theta(\lg b)$ insertions need to occur before we switch to a bit vector representation, followed by $\Theta(\lg b)$ deletions before reverting to an unsorted list. We can thus amortize the time needed for one insertion and one deletion to $\mathcal{O}(b/\lg b)$. At all times, predecessor queries take at most $\mathcal{O}(b/\lg u)$ time and the bucket requires at most b bits of space. Figure 1 shows an example.

Experimental evaluation. Following our considerations in Appendix B,

(1) we set $b := 2^{24}$ for buckets backed by *bit vectors*,

(2) we set $b := 2^{10}$ for buckets backed by *unsorted lists* and

(3) for *hybrid* buckets, we set $\theta_{\min} := 2^9$ and $\theta_{\max} := 2^{10}$ and try different sizes b.

Our results are shown in Figure 2. We first discuss the results for 32-bit keys. In most configurations, we achieve throughputs higher than 2^{22} operations per second for both updates (insertions and deletions) and predecessor queries. Furthermore, we achieve compression in that we require less than 32 bits per key, because we only store trunacted keys within the buckets. The compression increases with larger n as for sufficiently large n, all buckets are active and more (truncated) keys are inserted into the same number of buckets. We observe that the top level organization, array versus hash table, barely appears to matter. The only difference is a slightly slower performance, but also lower memory consumption of the hash tables for smaller n, which was to be expected. However, as all buckets become active for larger n, these differences become negligible.

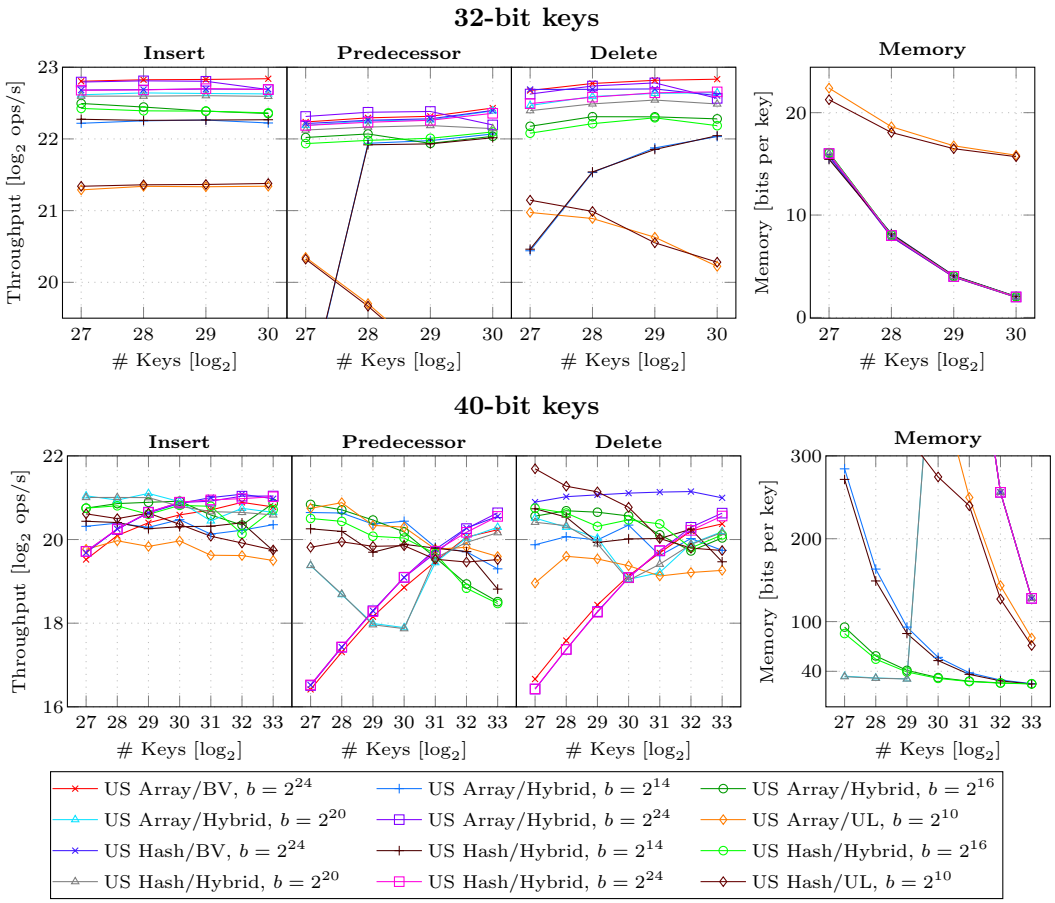


Figure 2 Throughputs for the insert, predecessor and delete operations, as well as memory usage of the universe sampling data structures for 32-bit (top) and 40-bit keys (bottom). Best viewed in colour. In the legend, *US* stands for universe sampling, *BV* stands for buckets backed by bit vectors, *UL* for unsorted lists. Missing points indicate throughputs lower than 2^{20} operations per second or exceeding of 300 bits consumed per key, respectively, and are omitted for clarity.

We now look at the three types of bucket organization. The clear outliers are where we implement buckets as unsorted lists of size at most 2^{10}: here, all operations are between 2–4 times slower than the rest, and the number of inserted keys visibly affects the performance of queries and deletions negatively, which is due to linear scans facing a higher bucket fill rate. Hybrid buckets and those backed by bit vectors appear to be on par especially for large n, as the hybrid representation eventually switches to bit vectors. As expected, the bit vector representation achieves higher compression than the unsorted list representation.

Now, we discuss the results for 40-bit keys. The memory consumption is obviously different: while we still achieve compression below 40 bits per key for large n, the top level now contains up to 2^{30} active buckets (for buckets of size 2^{10}), causing a big memory overhead that can only be compensated for sufficiently large n. Hybrid buckets may cause an explosion of memory consumption when they switch to bit vectors, as can be seen for bucket size 2^{24} at 2^{30} keys. For 2^{33} keys, we can see how it slowly starts to compensate. Regarding performance, similar observations as for 32-bit keys can be made, except that the top level organization now does matter: the smaller the buckets, the more linear scans weigh in, such that the hash table approach becomes faster for updates but slower for queries. Since this data structure is not suitable for large universes, we omit experiments for 64-bit keys.

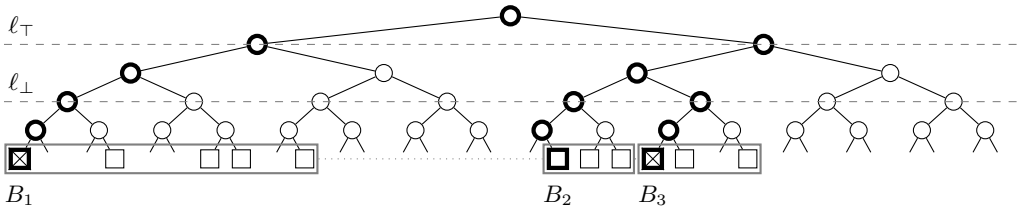


Figure 3 Our y-fast trie for $w = 5$ and $S = \{3, 6, 7, 9, 17, 18, 19, 21, 23\}$ with $t = 2$, $c = 2$ and $\gamma = 1$. Edge and leaf labels of the conceptual trie are omitted for the sake of clarity: left edges are labeled by 0, right edges by 1 and leaves are labeled by the corresponding keys. Keys contained in S are shown as squares, where representatives of buckets have a thick border. Representatives marked with an X are deleted: they are still representatives of buckets, but no longer contained in S themselves. Buckets are shown as rectangles around the contained keys. Nodes on paths that lead to representatives are contained in the x-fast trie's LSS and are drawn with a thick border; other nodes are not contained in the LSS. Levels $\ell_\top$ and $\ell_\bot$ are highlighted by dashed lines.

5 Y-Fast Tries

The *x-fast* trie by Willard [26] is conceptually a variation of the binary trie where
(1) the keys of S are doubly-linked in ascending order and
(2) if a node does not have a left (right) child, then the corresponding pointer is replaced by a *descendant* pointer that directly points to the smallest (largest) leaf descending from it.

The trie is stored in the *level-search data structure* (LSS). We say that a node of the trie is on level ℓ if it has depth ℓ. For each level ℓ of the trie, the LSS stores an entry for every node v that exists on level ℓ, which we identify by the bit sequence $B_v \in \{0,1\}^\ell$ that encodes the path in the trie from the root to v. Specifically, the LSS associates B_v to v's descendant pointers. We can find $\text{pred}_S(x)$ in expected time $\mathcal{O}(\lg \lg u)$ as follows: we first binary search the $\lceil \lg u \rceil$ levels of the trie to find the bottom-most node v on the path leading to x if it were contained in S. On each level ℓ that we inspect, we query the bit prefix $x\langle 0 \,..\, \ell - 1\rangle$ in the LSS in $\mathcal{O}(1)$ expected time to test if we are done. From v, by construction, we can take a descendant pointer to the predecessor or successor of x. Updates of the x-fast trie require $\mathcal{O}(\lg u)$ expected time as in the worst case, the LSS needs to be updated for every level following an insertion or deletion. The total memory consumption of the x-fast trie is $\mathcal{O}(n \lg u)$ words.

The *y-fast trie* improves this to $\mathcal{O}(n)$ words: we partition S into $\Theta(n/\lg u)$ buckets of $\Theta(\lg u)$ keys each and determine a *representative* for each bucket, e.g., the minimum contained key. Then, we build an x-fast trie over only the representatives, which occupies $\mathcal{O}(n)$ words of memory. For each bucket, we construct a binary search tree for the keys contained in it, consuming $\mathcal{O}((n/\lg u) \cdot \lg u) = \mathcal{O}(n)$ words. When looking for the predecessor of x, we can locate its bucket using the x-fast trie over the representatives in expected time $\mathcal{O}(\lg \lg u)$ and within the buckets, searching and updating can be done in time $\mathcal{O}(\lg \lg u)$. The sampling of representatives also improves the amortized expected update times to $\mathcal{O}(\lg \lg u)$.

Implementation. Let $t = \Theta(\lg u)$, $\gamma > 0$, and $c > 2\gamma$ be parameters. We partition S into buckets of size variable in $[\gamma t, ct]$. We name the minimum key contained in a bucket its representative and only representatives are contained in the x-fast trie, which has height $\lceil \lg u - \lg t \rceil$ as the lowest $\lg t$ bits of the keys are maintained in the buckets. Within a bucket, we store keys in a sorted or unsorted list rather than a binary search tree. This increases the asymptotic time needed for updates and predecessor queries, but the additional memory costs for structures such as binary trees, albeit asymptotically constant, would be too high in practice. Figure 3 shows an example of our y-fast trie that we describe in the following.

We try to keep t small such that buckets fit into few consecutive cache lines and can be searched quickly. Because this directly affects the height of the x-fast trie maintaining the representatives, we speed up searches as follows: let $\ell_\top$ be the bottommost level where *all* possible nodes exist in the x-fast trie and let $\ell_\bot$ be the topmost level where *no branching nodes* exist in the x-fast trie. Consider an operation involving a key $x \in U$: we locate its bucket by a vertical binary search in the x-fast trie's LSS. Because all levels above $\ell_\top$ contain all possible nodes and because all nodes on levels below $\ell_\bot$ point to the same buckets as their respective ancestors on level $\ell_\bot$, we limit the binary search to the levels between $\ell_\top$ and $\ell_\bot$ and maintain $\ell_\bot$ and $\ell_\top$ under updates with no asymptotic extra cost. With this strategy, we can also save space by avoiding storage of any nodes on levels below $\ell_\bot$; the corresponding hash tables in the LSS simply remain empty. Intuitively, this cuts off trailing unary paths in the x-fast trie. Note that due to the sampling mechanism, we always have $\ell_\bot \leq \lg(\gamma t)$, so the $\Theta(\lg \lg u)$ bottommost levels are never stored.

To speed up deletions, we allow the representative of a bucket to be no longer contained in the bucket by itself. When it is deleted, we mark it as such, but it remains the bucket's representative and also remains in the x-fast trie. This strategy avoids the need of finding a new representative and updating the x-fast trie every time a representative is deleted. However, we must consider a special case when answering predecessor queries. Let y_{rep} be the deleted representative of a bucket and $y_{\min} > y_{\text{rep}}$ the smallest key currently contained in the bucket, and consider the query $\text{pred}_S(x)$ with $y_{\text{rep}} \leq x < y_{\min}$. The x-fast trie will lead us to said bucket and y_{rep} would be the predecessor of x. When we detect y_{rep} as deleted, we follow a pointer to the preceding bucket, which must contain the predecessor of x.

Buckets are merged and split as in B-trees [7, chapter 18] to ensure their size stays within $[\gamma t, ct]$. To avoid the creation of a new bucket each time a new minimum is inserted into the data structure, we maintain a special bucket with representative $-\infty$ that we allow to become empty and will never be removed by a merge.

When using unsorted lists to maintain keys within a bucket, we can amortize insertion costs. Unless a split is required, inserting a key into a bucket simply means appending it in constant time. Thus, if we choose $c := \gamma + \Theta(t) > 2\gamma$, we can amortize the time needed for a split over $\Theta(t)$ constant-time insertions. This amortization leads to $\mathcal{O}((\lg u)/t) = \mathcal{O}(1)$ time needed for inserting a key into a bucket followed by a potential split, such that the amortized expected insertion time of the y-fast trie is $\mathcal{O}(\lg \lg u)$ as in the original. This cannot be achieved for deletions, as the key to be deleted needs to be located in time $\mathcal{O}(\lg u)$ first.

▶ **Example 1** (insertion)**.** Consider the y-fast trie in Figure 3 with $t = 2$ and $c = 2$. We insert the new key 8. The binary search in the x-fast trie's LSS is constrained only to the three levels between $\ell_\top$ and $\ell_\bot$ and leads to bucket B_1 with (deleted) representative 0. We insert x by appending it to the unsorted list of keys. However, we then have $|B_1| = 5 > 4 = ct$, thus we have to split B_1. We create a new bucket B_1' with representative 7 (the median) and move keys such that $B_1 := \{3, 6\}$ and $B_1' := \{7, 8, 9\}$. Even though 0 is no longer contained in B_1, it remains its representative. Finally, we enter key 7 into the x-fast trie, causing two new nodes to be added to the LSS. However, $\ell_\bot$ remains unchanged, as the newly added nodes form a unary path beginning at level $\ell_\bot$.

▶ **Example 2** (deletion)**.** Consider the y-fast trie in Figure 3 with $t = 2$ and $\gamma = 1$. We delete key 21, which we find in bucket B_3 as in Example 1. After deletion, we have $|B_3| = 1 < 2 = \gamma t$ (note how 20 is the representative, but is marked as deleted), thus we have to merge. As the only neighbour, we merge with bucket B_2 by moving key 23 such that $B_2 := \{17, 18, 19, 23\}$.

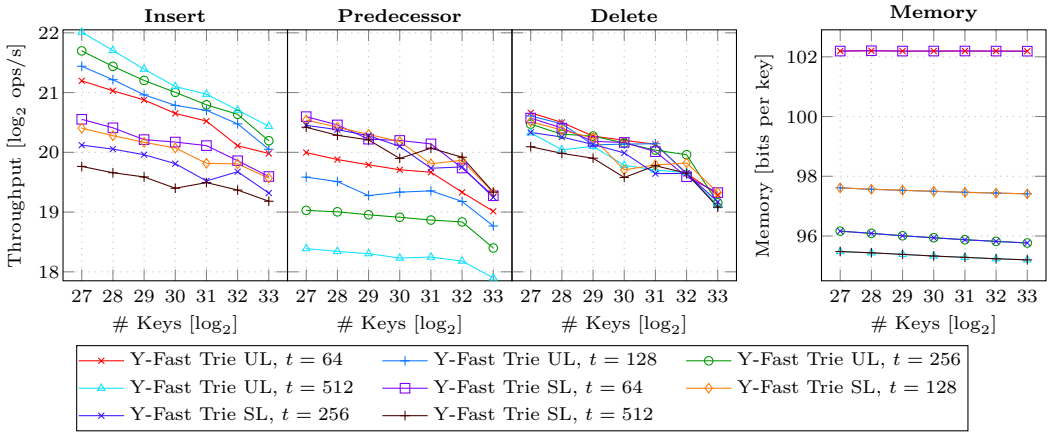


Figure 4 Throughputs for the insert, predecessor and delete operations, as well as memory usage of the y-fast trie for $U = [2^{64}]$. Best viewed in colour. *UL* stands for unsorted, *SL* for sorted lists.

The former representative of B_3, key 20, is now removed from the x-fast trie. Observe how the path of nodes leading to B_2 now becomes a unary path starting at level $\ell_\top$. Because all unary paths then start at level $\ell_\top$, we set $\ell_\bot := \ell_\top$.

Experimental evaluation. In our experiments, we set $c := 2$ and $\gamma := 1/4$ and choose t as powers of two to optimize memory alignments. Our results for $t := 64$ to 512 and 64-bit keys are shown in Figure 4. (Additional results are given in Figure 8 in Appendix A.)

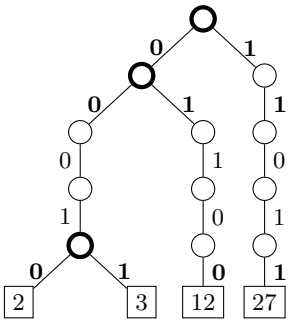
The fastest predecessor queries are achieved when buckets are organized as sorted lists and binary search is used to answer bucket-level queries. Conversely, insertions are are up to twice as fast in the unsorted list case, where new keys simply have to be appended without preserving any order. Regarding deletions, there is no substantial difference between using a sorted or unsorted list to organize the buckets: while we can find the item to be deleted using binary search when using a sorted list, we have to shift up to t keys afterwards. As we use simple arrays for storage in either case, there is also no difference in memory consumption.

As expected, the bucket size of t is a direct trade-off parameter for update versus query performance and memory usage, which is very visible when buckets are organized as unsorted lists. Here, insertions become faster as the bucket size grows since they are trivial on the bucket level and the LSS needs to be updated less often. However, larger buckets mean longer scans when answering predecessor queries. The bucket size is much less impactful on query performance when sorted lists are used, as the bucket-level query time is then only logarithmic in the bucket size. The memory consumption is also affected by the bucket size: larger buckets imply less levels in the LSS and thus less memory needed.

As a conclusion, unsorted lists may be preferable when fast insertions are required and the performance of predecessor queries is less important. For the general case, however, using sorted lists appears to be preferable, as all operations then have similar throughputs.

6 Fusion Trees

Pătraşcu and Thorup [22] introduce *dynamic fusion nodes* as a sorted list data structure for $|S| \leq k \leq \sqrt{w}$ keys that supports predecessor queries and updates in time $\mathcal{O}(1)$. It is based on the fusion node, originally described by Fredman and Willard [13], that simulates



(a) The binary trie for S. Branching nodes have thicker outlines and bits at distinguishing positions are written in bold.

$M = \texttt{11001}$

x	binary	$\hat{x}$	$\hat{x}^?$	BRANCH	FREE
2	00010	000	000	000	000
3	00011	001	001	001	000
12	01100	010	01?	010	001
27	11011	111	1??	100	011

(b) The keys $x \in S$, with binary representation and compressed versions $\hat{x}$ and $\hat{x}^?$ without and with don't cares according to [13] and [22], respectively. BRANCH and FREE encode the matrix given by column $\hat{x}^?$ as described in [22]. The mask M marks the distinguishing positions of S.

Figure 5 Binary trie and compressed keys for $S = \{2, 3, 12, 27\}$ and $w = 5$.

navigation in a compact binary trie of S represented by compressed keys. Given a key $x \in S$, we only consider those bits at *distinguishing* positions. A position $\ell < \lceil \lg u \rceil$ is a distinguishing position if there is at least one branch on level ℓ in the binary trie. For k keys, there can be at most k branches in the binary trie, and thus there can be at most k distinguishing positions. A *compressed key* $\hat{x}$ consists of only the bits of x at distinguishing positions moved to the k least significant positions. We maintain the set of distinguishing positions in a *mask* M of w bits where the k distinguishing bits are set and all other bits are clear. Figure 5b shows an example. From x, we can compute $\hat{x}$ in constant time by masking out unwanted bits using M, followed by multiplications to relocate the distinguishing bits. The k compressed keys of S can be stored in a $k \times k$ bit matrix $\hat{S}$ that fits into a single word. With this, we can compute $\text{pred}_S(x)$ in time $\mathcal{O}(1)$ as Fredman and Willard describe in [13]. However, updates may cause a new position to become distinguishing after an insertion, or a position to be no longer distinguishing after a deletion. In these cases, their data structure needs to be rebuilt from scratch. To resolve this, Pătraşcu and Thorup introduce *don't care* bits (written `?`) that indicate bits at distinguishing position that are, however, not used for branching in a specific compressed key. The data structure now contains a $k \times k$ matrix over the new alphabet $\{\texttt{0}, \texttt{1}, \texttt{?}\}$, which we encode using two $k \times k$ bit matrices that fit into one word each. Examples for this can be seen in Figure 5b. The notion of wildcards allows for updating the data structure in time $\mathcal{O}(1)$. We refer to Appendix C for a more detailed description and examples, including an elaboration of the deletion of keys not given in [22].

A *B-tree* is a self-balancing multiary tree data structure for representing a dynamic ordered set of items. With B the maximum degree of a node, it is guaranteed to maintain height $\log_B n$, such that lookup – including predecessor – queries and updates can be done in time $\mathcal{O}(\log_B n)$. We consider B-trees a well-known folklore data structure and refer to [7, chapter 18] for a comprehensive introduction. Embedding fusion nodes into a B-tree, using the keys contained in the nodes as splitters, are the typical ingredients of a *fusion tree*.

Implementation. As we deal with 64-bit architectures ($w = 64$), we choose $k := 8$, such that a $k \times k$ bit matrix can be stored in a single word represented row-wise by an array $\hat{X} = [\hat{x}_0, \ldots, \hat{x}_7]$ of compressed keys. We keep $\hat{X}$ in ascending order, i.e., $\hat{x}_0 < \hat{x}_1 < \cdots < \hat{x}_7$.

The most important operation is key compression, writing only the bits of x at distinguishing positions into a word $\hat{x}$. Instead of an approach based on sparse tables [23], we make use of the *parallel bits extract* (PEXT) instruction [16]. Let M be the w-bit mask identifying

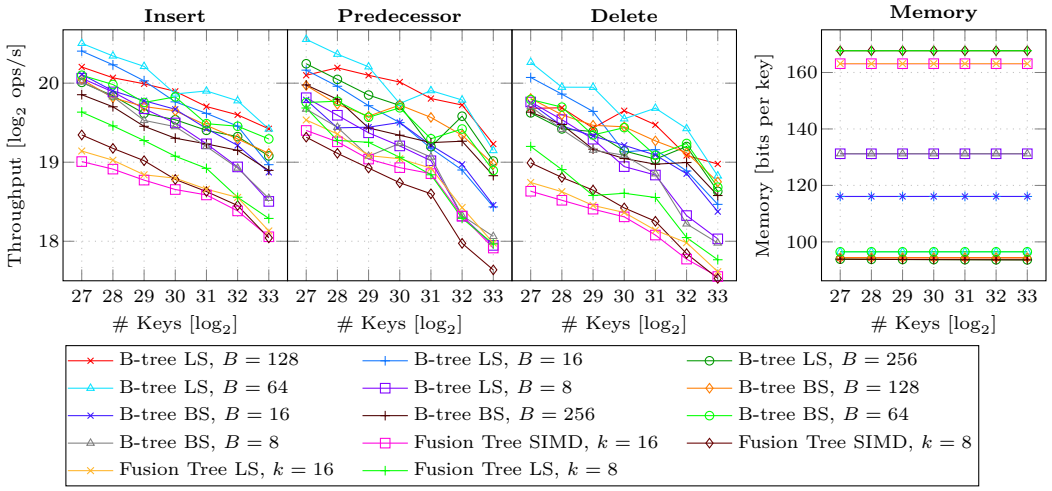


Figure 6 Throughputs for the insert, predecessor and delete operations, as well as memory usage of the fusion trees and B-trees for $U = [2^{64}]$. Best viewed in colour. In the legend, *LS* stands for linear searched nodes, *BS* for binary search and *SIMD* for use of SIMD instructions.

distinguishing positions. Then, conveniently, $\hat{x} = \text{PEXT}(x, M)$. Another core operation is finding the rank i of a compressed key $\hat{y}$ in $\hat{X}$. For this, we use the MMX SIMD instruction PCMPGTB [16], which performs a byte-wise greater-than comparison of two 64-bit words. First, we multiply $\hat{y}$ by the constant $(0^{k-1}1)^k$ to retrieve the word $\hat{y}^k$ containing k copies of $\hat{y}$. Let j be the smallest rank such that $\hat{x}_j > \hat{y}$. The instruction PCMPGTB$(\hat{X}, \hat{y}^k)$ returns the word $B_{\hat{X}>\hat{y}}$ where the kj lowest bits are zero and the remaining bits are set (because $\hat{X}$ is ordered). Therefore, $j = \lfloor \text{TZCNT}(B_{\hat{X}>\hat{y}})/k \rfloor$ and finally $i = j - 1$. Alternatively, because $\hat{X}$ easily fits into a cache line, we consider a naïve linear search.

We extend our implementation to support also $k = 16$ by simulating a 256-bit word using four 64-bit words. The special CPU instructions can be extended by executing them on each of the four 64-bit words and then combining the results. The processors to our disposal actually support a variant of PCMPGTB for the parallel comparison of sixteen 16-bit words contained in a 256-bit word (namely the Intel intrisic `_mm256_cmpgt_epi16`).

We implement B-trees largely following the description in [7, chapter 18]. Nodes have at most B children and thus contain up to $B - 1$ keys used as splitters. When plugging fusion nodes into B-trees of degree k, we have *fusion trees*.

Experimental evaluation. We use $B := 8$ and $B := 16$ for fusion trees with $k = 8$ and $k = 16$, respectively, comparing fusion nodes using the PCMPGTB instruction for rank queries against those using simple linear scans. Further, we compare fusion trees to straight B-trees, finding predecessors in a node using $\mathcal{O}(\lg B)$-time binary or $\mathcal{O}(B)$-time linear search. There, we also consider much larger B, as preliminary experiments suggested that the performance of all operations peaks at $B := 64$. Our results for 64-bit keys are presented in Figure 6. (More results for smaller universes are given in Figure 9 in Appendix A.)

To our surprise, fusion trees achieve the lowest throughputs for all operations: B-trees with large B are up to twice as fast, and even the B-trees with low degrees are visibly faster overall. Fusion trees also require more memory per key, which was expected, as each node needs to store three words (the compression mask and two matrices) in addition to the keys themselves. Interestingly, the fusion nodes using linear scans for ranking outperform those

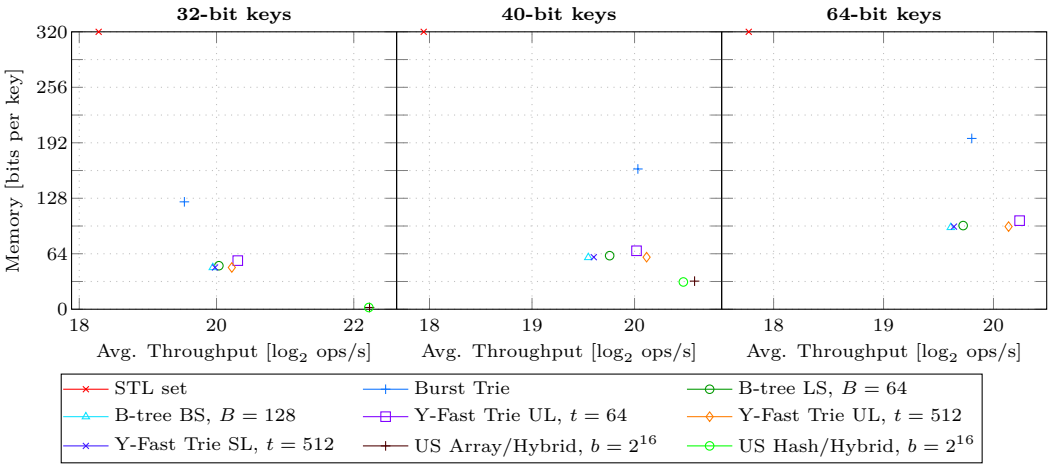


Figure 7 Comparing the average throughput of operations versus memory use of dynamic predecessor data structures for different universes and $n = 2^{30}$.

that use the SIMD instructions in nearly all instances. The reason is presumably that the corresponding MMX/AVX registers have to be filled prior to executing these instructions: in a direct comparison answering immediately consecutive random rank queries, the SIMD variant is about 28% faster than scanning. Fusion trees with $B = 16$ perform slower overall than those with $B = 8$ despite their lower height, which is due to overheads in our simulation of 256-bit words. It shall be interesting to redo these experiments with natively supported wide registers (e.g., AVX-512) and necessary instructions in the future.

We have a brief closer look at B-trees. Our preliminary experiments are largely confirmed in that B-trees with $B = 64$ perform best overall. For $B \leq 64$, nodes backed by linear search perform faster than those backed by binary search. The exact opposite is the case for $B > 64$, where binary search becomes faster. Concerning memory, unsurprisingly, the higher B is chosen, the less memory is required as the tree structure shrinks in height.

7 Comparison

In Figure 7, we plot the average throughput of insertions, predecessor queries and deletions against the memory usage of a subset of our data structures from Sections 4–6 for a fixed workload size of 2^{30}. For comparison, we also show the performance of the STL set (`std::set`, an implementation of red-black trees), and the burst trie of Nash and Gregg [19] – to the best of our knowledge the best practical dynamic predecessor data structure thus far. Note that burst tries are *associative* and store a value along with each key, so for a fair comparison, one should subtract w bits per key for each data point.

For all universes, at least one of our data structures is over four times faster than the STL set, the extreme being for 32-bit keys, where our sampling structures achieve an average throughput of approximately $2^{22.2}$ operations per second, whereas the set does about $2^{18.2}$. Furthermore, our data structures consume less than a third of the set's 320 bits per key. For 32-bit keys, we also outperform burst tries completely, where even our slowest data structure (B-trees with degree 128 and binary searched nodes) is about 33% faster. Our two sampling data structures with hybrid buckets of size 2^{16} are clearly the fastest. Their low space consumption of just about 2 bits per key should, however, be interpreted with care, as for $n = 2^{30}$, one quarter of all possible keys is contained in S, and hence they essentially

store S in a bit vector. This also shows up for $w = 40$ (but less pronounced), where they become the only data structure clearly faster than the burst tries. Our y-fast tries with unsorted buckets of size 2^9 are about 6% faster than burst tries, but still require considerably less memory even respecting that 40 bits per key in burst tries are for associated values. It appears that y-fast tries with unsorted buckets scale best with the size of the universe: for 64-bit keys, it is the fastest data structure with buckets of size either 2^6 or 2^9 and is approximately 30% faster than burst tries, again consuming significantly less memory. The y-fast tries with sorted buckets are about on par with our B-trees, which are overall between 14% and 56% slower than y-fast tries with unsorted buckets.

For 32-bit keys, we intended to include the stratified tree [9], but it failed to stay within the memory limits (256 GB) starting at 2^{30} keys. For 2^{29} keys, it consumed about 1,480 bits per key, ranking lowest with an average throughput of circa $2^{17.7}$ operations per second.

Conclusions. Our dynamic predecessor data structures are the most memory efficient of all tested. They clearly outperform the STL set and for all universes in question, at least one of our data structures is faster than burst tries, the previously fastest known to us. We confirm once more [4, 10, 11, 15] that naïve solutions can be more practical than sophisticated data structures on modern hardware and sufficiently small inputs. We also observed that SIMD instructions, while faster than sequences of *classic* (SISD) instructions when used in batches, may turn out less useful in more complex scenarios.

References

1 Advanced Micro Devices Inc. *AMD64 Architecture – Programmer's Manual Volume 3: General-Purpose and System Instructions*, September 2020. URL: `https://www.amd.com/system/files/TechDocs/24594.pdf`.

2 Miklós Ajtai, Michael L. Fredman, and János Komlós. Hash functions for priority queues. *Inf. Control.*, 63(3):217–225, 1984. `doi:10.1016/S0019-9958(84)80015-7`.

3 Arm Limited. *A64 Instruction Set Reference*, 2018. URL: `https://developer.arm.com/documentation/100076/0100/a64-instruction-set-reference`.

4 Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. Practical range minimum queries revisited. In *16th International Symposium on Experimental Algorithms (SEA)*, pages 12:1–12:16. Dagstuhl, 2017. `doi:10.4230/LIPIcs.SEA.2017.12`.

5 Djamal Belazzougui. Predecessor search, string algorithms and data structures. In *Encyclopedia of Algorithms*, pages 1605–1611. Springer, 2016. `doi:10.1007/978-1-4939-2864-4_632`.

6 Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin hood hashing (preliminary report). In *26th Symposium on Foundations of Computer Science (FOCS)*, pages 281–288. IEEE, 1985. `doi:10.1109/SFCS.1985.48`.

7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

8 Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–14. ACM, 1997. `doi:10.1145/263105.263133`.

9 Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list data structure for 32 bit key. In *6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 142–151. SIAM, 2004. URL: `https://web.archive.org/web/20201111145353/http://algo2.iti.kit.edu/dementiev/files/veb.pdf`.

10 Patrick Dinklage, Johannes Fischer, Alexander Herlez, Tomasz Kociumaka, and Florian Kurpicz. Practical performance of space efficient data structures for longest common extensions.

In *28th European Symposium on Algorithms (ESA)*, pages 39:1–39:20. Dagstuhl, 2020. doi:10.4230/LIPIcs.ESA.2020.39.

11 Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *J. Discrete Algorithms*, 43:72–80, 2017. doi:10.1016/j.jda.2016.09.002.

12 E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, 1960. doi:10.1145/367390.367400.

13 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

14 Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002. doi:10.1145/506309.506312.

15 Lucian Ilie and Liviu Tinta. Practical algorithms for the longest common extension problem. In *16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 302–309. Springer, 2009. doi:10.1007/978-3-642-03784-9_30.

16 Intel Corporation. *Intel (R) 64 and IA-32 Architectures – Software Developer's Manual – Volume 2: Instruction Set Reference, A-Z*, September 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

17 Maureen Korda and Rajeev Raman. An experimental evaluation of hybrid data structures for searching. In *3rd International Workshop on Algorithm Engineering (WAE)*, pages 214–228. Springer, 1999. doi:10.1007/3-540-48318-7_18.

18 Kurt Maly. Compressed tries. *Commun. ACM*, 19(7):409–415, 1976. doi:10.1145/360248.360258.

19 Nicholas Nash and David Gregg. Comparing integer data structures for 32- and 64-bit keys. *ACM J. Exp. Algorithmics*, 15, 2010. doi:10.1145/1671970.1671977.

20 Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5), 2020. doi:10.1145/3409371.

21 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *31st Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240. ACM, 2006. doi:10.1145/1132516.1132551.

22 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th Symposium on Foundations of Computer Science (FOCS)*, pages 166–175. IEEE, 2014. doi:10.1109/FOCS.2014.26.

23 Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979. doi:10.1145/359168.359175.

24 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Symposium on Foundations of Computer Science (FOCS)*, pages 75–84. IEEE, 1975. doi:10.1109/SFCS.1975.26.

25 M. Wenzel. Wörterbücher für ein beschränktes Universum (dictionaries for a bounded universe). *Master's thesis, Saarland University, Germany*, 1992.

26 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inform. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

A Additional Results

We give experimental results in addition to those presented in Sections 4 through 6, which have been omitted there for the sake of clarity as they lead to largely the same conclusions. Figure 8 shows results for our y-fast tries (Section 5) for 32-bit and 40-bit universes, respectively, Figure 9 for our fusion and B-trees (Section 6).

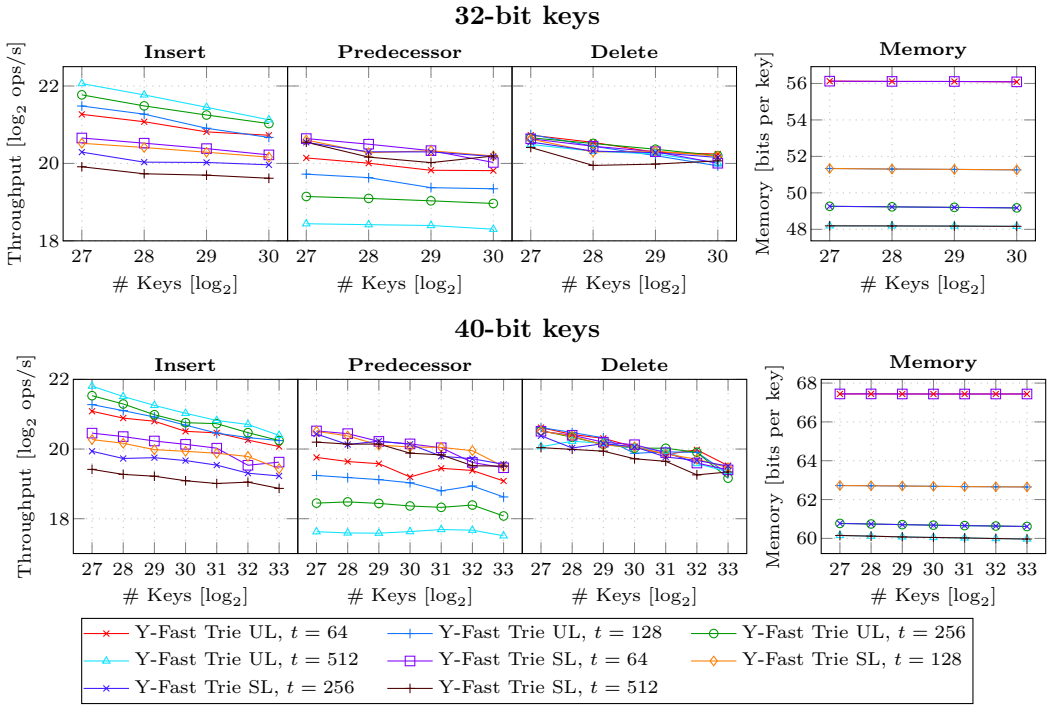


Figure 8 Throughputs for the insert, predecessor and delete operations, as well as memory usage of the y-fast trie for 32-bit (top) and 40-bit keys (bottom). Best viewed in colour.

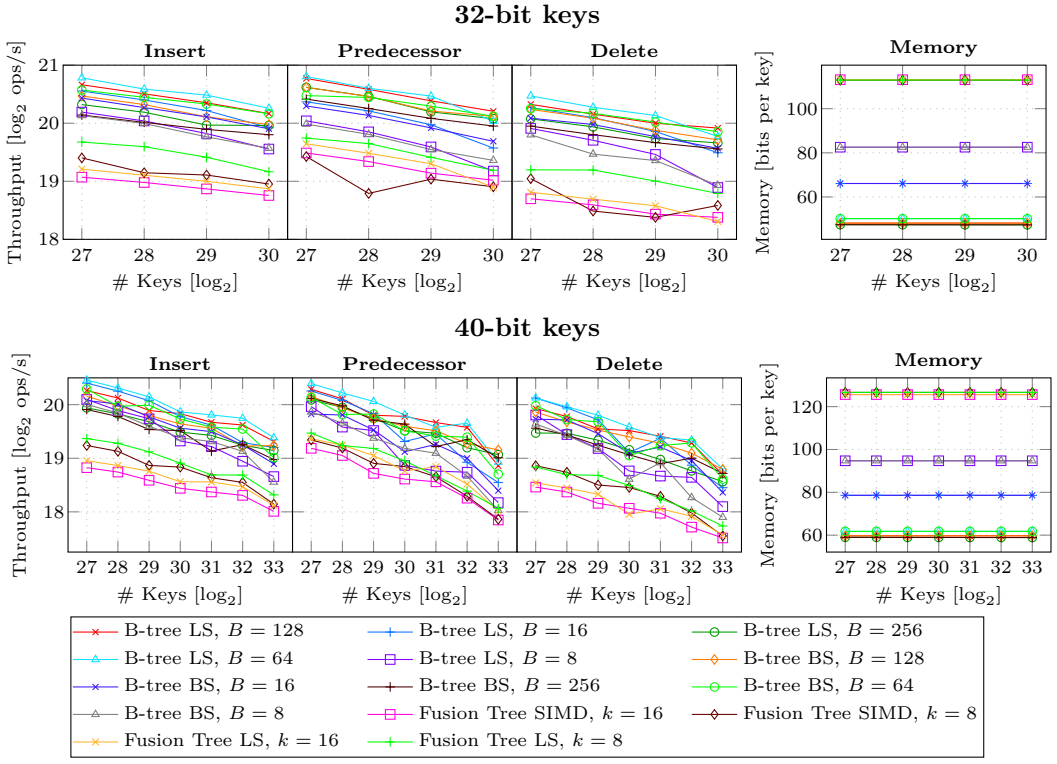


Figure 9 Throughputs for the insert, predecessor and delete operations, as well as memory usage of the fusion and B-trees for 32-bit (top) and 40-bit keys (bottom). Best viewed in colour.

B Choosing Parameters For Universe Sampling

The key question for preparing the experiments for our sampling data structure (Section 4) is how to configure the bucket size b, which is a direct trade-off parameter for the performance of the top level versus that of the bucket level: at the top level, we have worst-case costs of $\mathcal{O}(u/b)$ for updates or queries depending on the chosen data structure. At the bucket level, we have costs of $\mathcal{O}(b)$ for queries. To that end, we want to pick b large enough so that the top level does not end up with too many entries, and pick it small enough so that operations on the bucket level do not take too much time.

We first do some considerations on the top level. When we assume a uniform distribution of keys inserted into the data structure, we observe that the number of insertions required until every bucket is active is distributed geometrically. What follows is that long scans for active buckets on the top level occur more and more rarely as more keys are inserted into the data structure. The same conclusion can be drawn when inserted keys are skewed towards a range within U, because then it occurs rarely that active buckets far away from others need to be accessed. Therefore, assuming that a large enough number of keys is going to be inserted so that long top-level scans occur rarely, we focus on bucket-level performance.

On the bucket level, we have to consider our three strategies for maintaining the keys. First, when using an unsorted list, smaller buckets clearly result in faster query times. In preliminary experiments, the performance declined only marginally up to a bucket size of $b_{\text{list}} := 2^{10}$ keys, whereas buckets any larger caused a significant drop. When using a bit vector, we benefit from scanning through bits packed into words, resulting in large buckets of $b_{\text{bv}} := 2^{24}$ keys still performing very well. Here, choosing larger buckets caused insertions and deletions to become slower. Because these operations simply mean setting or clearing bits, this effect can be explained by a higher number of cache misses. In the hybrid case, the initial notion is that we want the unsorted list to never consume more memory than the bit vector and thus switch to when a bucket exceeds $b/\lg b$ keys. As we desire to maintain the sweet spot threshold of $\theta_{\max} := b_{\text{list}} = 2^{10}$ for unsorted lists, we seek b_{hybrid} such that $b_{\text{hybrid}}/\lg b_{\text{hybrid}} > 2^{10}$. This is the case for $b_{\text{hybrid}} \geq 2^{14}$, such that the bucket size of $b_{\text{bv}} = 2^{24}$ is again an option. However, consider the case where the bucket is switched from an unsorted list representation to a bit vector: we want to avoid a sudden explosion of memory occupied – and potentially wasted – by a bucket. Because a *perfect* choice of b_{hybrid} cannot be done without any prior knowledge about the input, we explore different configurations.

We add here that we also tried *sorted* lists for maintaining the keys in the buckets, enabling binary search to speed up queries. However, the performance of updates greatly suffered and for smaller buckets, the query speedup compared to linear scans became marginal. Sorted lists are therefore not considered in our experiments.

C Elaboration On Dynamic Fusion Nodes

We expand on the description of dynamic fusion nodes from Section 6. Let $\hat{x}^{?}$ indicate a compressed key that may contain don't cares. The $k \times k$ matrix $\hat{S}^{?}$ over the alphabet $\{0, 1, ?\}$ is represented by two $k \times k$ binary matrices Branch and Free defined as follows:

$$\textsc{Free}_{ij} = \begin{cases} 0 & \text{, if } \hat{S}^{?}_{ij} \neq ? \\ 1 & \text{, if } \hat{S}^{?}_{ij} = ? \end{cases} \qquad \textsc{Branch}_{ij} = \begin{cases} \hat{S}^{?}_{ij} & \text{, if } \textsc{Free}_{ij} = 0 \\ 0 & \text{, if } \textsc{Free}_{ij} = 1 \end{cases}$$

Intuitively, Free identifies the don't care bits in $\hat{S}^{?}$, and $\textsc{Branch}_{ij}$ is either equal to $\hat{S}^{?}_{ij}$ if a bit is used for branching, or zero if it is a don't care bit. The concatenation of the bits on the i-th *row* of $\hat{S}^{?}$ represent the compressed (with don't cares) i-th key contained in S.

We can find the predecessor of some key $x \in U$ by determining its rank $i < k$ among the keys in S. In the following, we reduce this rank query to compressed keys. To that end, we assume that $\hat{S}^?$ is maintained such that the rows are in ascending order. If this is not the case, we can afford to maintain an index as described in [2] without worsening the asymptotically constant query and update times. We now seek the number i' of the row in $\hat{S}^?$ that corresponds to the rank of $\hat{x}$ among the compressed keys. We say that $\hat{x} \in \{0,1\}^k$ *matches* a compressed key $\hat{y}^? \in \{0,1,?\}^k$ with don't cares if all non-don't care bits in $\hat{y}^?$ are equal to the corresponding bits in $\hat{x}$. Formally, this is the case if $\forall j < k : \hat{y}^?\langle j\rangle = ? \vee \hat{x}\langle j\rangle = \hat{y}^?\langle j\rangle$. We define the operation match(x) that, simultaneously for all $j < k$, tests whether $\hat{x}$ matches the compressed key encoded in the j-th row of $\hat{S}^?$ and reports the smallest j where this is not the case. Pătraşcu and Thorup [22] show how to perform this operation in constant time by

(1) computing $\hat{S}^{\hat{x}}$ by replacing the don't care bits in $\hat{S}^?$ by the corresponding bits of the $k \times k$ bit matrix $\hat{x}^k$ that contains k copies of $\hat{x}$ and

(2) performing a parallel row-wise greater-than comparison of $\hat{S}^{\hat{x}}$ against $\hat{x}^k$.

We then have $i' = \text{match}(x)$. If $x \in S$, then i' is also the rank of x within S as shown in [13]. Therefore, if $x = S[i']$, we already found the predecessor of x after one match operation. However, if $x \neq S[i']$, it is $x \notin S$. In the trie, consider the ancestor of the leaf of x – if x were contained in S – at level $j = \text{msb}(x \oplus S[i'])$. At this node, we branched off in a direction that does not necessarily lead us to the predecessor of x. To see examples of this, refer to Examples 3 and 4 below. To find the actual rank i of x within S and thus the predecessor $S[i]$ of x, we simulate the necessary trie navigation by performing another match operation. Consider the case where $x < S[i']$. In the trie, we navigate up to the lowest ancestor v that has two children and take the path to the rightmost leaf in the left subtrie of v. An equivalent approach is to take the path to the leftmost leaf in the right subtrie of v, and subtract one from that leaf's rank. The latter can be simulated by computing $i = \text{match}(x \wedge 1^{w-j}0^j) - 1$. In the case that $x > S[i']$, symmetrically, we simulate navigation to the rightmost leaf in the left subtrie of v by computing $i = \text{match}(x \vee 0^{w-j}1^j)$.

Pătraşcu and Thorup further show how to perform all the necessary manipulations of $\hat{S}^?$ in constant time in order to insert keys into the data structure. The intuition is always that $\hat{S}^?$ is stored in two words and all required word operations can be done in constant time.

▶ **Example 3.** We consider a predecessor search for $x = 25$ in the set S from Figure 5 with $w = 5$. The binary representation of x is `11001`, which we compress to $\hat{x} =$ `111`. We compute $i' = \text{match}(x) = 4$, corresponding to $S[i'] = 27$. This cannot be the predecessor of x because $x < S[i']$. The position at which we branched off in the wrong direction is $j = \text{msb}(x \oplus y) = 2$, at the node two levels above the leaf labeled by 27 in Figure 5a: a path leading to $x = 25$ would branch off to the left, whereas we branched off to the right. We simulate the necessary trie navigation by computing $i = \text{match}(x \wedge 1^{w-j}0^j) - 1 = \text{match}(\texttt{11000}) - 1 = 3$. Now, $S[i] = 12$ is the correct predecessor of x.

▶ **Example 4.** We consider a predecessor search for $x = 4$ in the set S from Figure 5 with $w = 5$. The binary representation of x is `00100`, which we compress to $\hat{x} =$ `000`. We compute $i' = \text{match}(x) = 1$, corresponding to $S[i'] = 2$. Since $x > S[i']$, we have the opposite case as in Example 3. The position at which we branched off in the wrong direction is $j = \text{msb}(x \oplus y) = 2$, at the node two levels above the leaf labeled by 2 in Figure 5a. We compute $i = \text{match}(x \vee 0^{w-j}1^j) = \text{match}(\texttt{00111}) = 2$, and $S[i] = 3$ is the predecessor of x.

Deleting keys

Pătraşcu and Thorup thoroughly describe the process of inserting a key into a dynamic fusion node in constant time [22]. They further claim that to delete a key, "we just have to invert the [...] process". However, some details require special attention, which is why we sketch the constant-time deletion of a key here. To that end, we use the same bag of tricks to perform the necessary $k \times k$ matrix manipulations in constant time using word operations and refer to [22] for the ideas.

Consider deleting a key $x \in U$ from our set S containing at most k keys from U. Recall that S is stored in a $k \times k$ matrix $\hat{S}^?$ of bits and don't cares represented by two words BRANCH and FREE, and that we maintain the k distinguishing positions by setting the corresponding bits in a mask M of w bits. We first compute the rank $i = \text{match}(x)$ of x within S in constant time. To verify that $x \in S$, we compare x against $S[i]$. If $x \notin S$, we abort the deletion. Otherwise, we require the position j of the least significant *distinguishing* bit at which x branches off in the trie. This position may no longer be distinguishing after the deletion of x and the, the corresponding bit must be removed from all remaining compressed keys in $\hat{S}^?$ to retain optimal compression. If j remains a distinguishing position, we need to replace the corresponding bits in all keys in the subtrie beneath node v by don't cares, where v is the ancestor of the leaf corresponding to x on level j. This is because due to the deletion of x, v loses a child and is no longer a branching node. We will not consider any distinguishing positions of significance higher than j, because for those, by construction, there must be at least one other key in S that branches off the corresponding trie node. The deletion of x works as follows:

1. Find the position j of the least significant distinguishing bit at which x branches off in the trie. This corresponds to the position of the least significant non-don't care bit in $\hat{x}^?$. Compute h by adding one to the number of trailing don't cares in $\hat{x}^?$, which equals the number of trailing ones in the i-th row of FREE. Then, $j = \text{select}_1(M, h)$.
2. Remove the i-th row, which contains $\hat{x}^?$, from $\hat{S}^?$.
3. Test if the deletion of x results in j no longer being a distinguishing position. This is the case if all non-don't care bits in the h-th column have the same value, indicating that there are no more branches at any node in the trie on level j. This can be done in constant time using a sequence of word operations; we refer to our code for details.
4. If that is the case, remove the h-th column from $\hat{S}^?$ and clear the j-th bit in M.
5. Otherwise, if j remains distinguishing, find the range i_0 to i_1 of keys in the subtree beneath j. This range must contain at least one key, because otherwise column h in row i would have been a don't care. For all compressed keys in the range, column h must be updated to a a don't care.

Compared to [22], we introduced two additional operations on words: counting trailing ones and a binary select operation. In Section 2, we already mentioned briefly how these can be performed in constant time both in theory and practice.

▶ **Example 5.** We consider the deletion of key $x = 12$ in *Figure* 5. It has rank $i = 3$ and occupies the third row in the matrix. We follow the steps of our sketched algorithm:

1. Observe that $\hat{x}^? = \texttt{01?}$ has one trailing don't care, so we have $h = 2$. The position of the least significant distinguishing bit at which x branches off is thus $j = \text{select}_1(M, h) = 2$. This corresponds to the second level in the trie shown in Figure 5a.
2. We remove the third row from $\hat{S}^?$, conceptually removing the leaf for $x = 12$ in the trie.

3. Observe how all non-don't care bits in column $h = 2$ of the matrix now have the same value 0. This corresponds to the fact that in the trie, on level $h = 2$, there are no longer any branches, which means that position j is no longer distinguishing.
4. Because j is no longer distinguishing, we remove the second column from the matrix completely and clear the corresponding bit in M.

The mask indicating distinguishing bits is now `10001`, and the matrix now consists of the compressed keys `00` (for key 2), `01` (for key 3) and `1?` (for key 27).