Freie Universität Berlin

Masterarbeit am Institut für Informatik an der Freien Universität Berlin

Arbeitsgruppe Theoretische Informatik

An In-Depth Analysis of Data Structures Derived from van-Emde-Boas-Trees

Marcel Ehrhardt

marehr@zedat.fu-berlin.de

Gutachter: Prof. Dr. Wolfgang Mulzer

October 12, 2015

Abstract

IP-packet routing is one of the most practical solved problems in computer science with millions of routed packets per second. The algorithmic problem behind it is the so-called predecessor problem. Assuming a universe U, in this problem one can query a set $S \subseteq U$ for the predecessor of an element q within the set S, i.e. $\max\{x \in S \mid x < q\}$. If the set S can be manipulated, the problem is called the dynamic predecessor problem, otherwise it is called the static predecessor problem.

In this thesis I give an overview of several different data structures for the w-bit word RAM on bounded integer universes, i.e. $U = \{0, \ldots, 2^w - 1\}$. Those data structures are based on the van-Emde-Boas-Tree [BKZ76]. For each data structure I work out the details of the algorithms by giving pseudo-code and for some of them I present new techniques, which give an easier and clearer presentation of the data structure and the algorithms. Last but not least, I answer an open problem concerning the space usage stated by Bose *et al* [Bos+13].

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder Ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 12.10.2015

Marcel Ehrhardt

Contents

> "Beware of bugs in the above code; I have only proved it correct, not tried it."
>
> – Donald Knuth,
>
> about his van-Emde-Boas tree implementation.

1. Introduction

In this thesis we will look at different solutions for the ***predecessor problem*** on a universe U. The predecessor problem is basically the problem to find the preceding element y of a given element $x \in U$ in a set $S \subseteq U$, i.e. $y = \max\{x \in S \mid x < q\}$.

The motivation behind studying the predecessor problem is that one of the most solved algorithmic problems every day, IP-packet routing, can be solved by finding the preceding element in a set of integers (i.e., the IP-adresses and IP-subnets encoded as integers) from a bounded universe. The routing algorithm decides for a given IP-packet on which port from the router the IP-packet should be routed by selecting the port with the most specific IP-address or IP-subnet.

In this thesis, we will focus on solutions for the predecessor problem of a set $S \subseteq U$ in a bounded universe U. We will assume that the bounded universe U is $\{0, \ldots, 2^w - 1\}$ for some fixed w and that w is a power of two. We will further assume that in the time bounds $\log(x)$ stands for $\log_2(x + 2)$. This ensures that multiple compositions of log's are always well defined.

This assumption will allow us to give faster algorithms compared to the more general comparison model, where the predecessor problem of a set S with $|S| = n$ elements can only be solved in $\Theta(\log n)$ time. For example, we will study the ***van-Emde-Boas tree*** which solves the predecessor problem in $\mathcal{O}(\log\log|U|)$ time, which is faster, if n is (much) larger than $\log|U|$.

The inventor of the van-Emde-Boas tree, Peter van Emde Boas, wrote a paper titled "Thirty nine years of stratified trees" [Boa13] which gives a historical overview of what was allowed at that time, why his solution was so complicated, how his idea evolved over time and what assumptions changed over time. I highly recommend reading it.

My thesis started from the blog posts of Mihai Pătraşcu, which can be found in the Appendix A, and evolved by reading and following various papers throughout the thesis. Unfortunately, Mihai Pătraşcu passed away in 2012.

1.1. The Predecessor Problem

The predecessor problem is commonly divided in the ***static*** predecessor problem, where the set S remains fixed, and the ***dynamic*** predecessor problem, where the set S can be manipulated.

The static predecessor problem has the following operations.

$search(q)$**:**	Given a query q, return q if $q \in S$ and $\perp$ otherwise.
$predecessor(q)$**:**	Given a query $q \in U$, return the predecessor element of q in the set S, i.e. $\max\{x \in S \mid x < q\}$. If the predecessor does not exist, return $\perp$.

successor(q): Given a query $q \in U$, return the successor element of q in the set S, i.e. $\min\{x \in S \mid q < x\}$. If the successor does not exist, return $\perp$.

The dynamic predecessor problem has the following additional operations.

insert(q): Insert the element $q \in U \setminus S$ into the set S, i.e. $S := S \cup \{q\}$.

remove(q): Delete the element $q \in S$ from the set S, i.e. $S := S \setminus \{q\}$.

We will further focus on solutions which are either space optimal, i.e. have linear space, or are later a part of a space optimal solution. If we allowed unlimited space, the static predecessor problem could be answered in $\mathcal{O}(1)$ time. Imagine a big array A where every field $A[i]$ has a pointer to its predecessor and successor element. Thus, for any $q \in S$, $A[q]$ answers the predecessor or successor query in constant time. Another example, where more space delivers better results, is given by Beame and Fich [BF99] as mentioned by Mihai and Thorup [PT07]. They improved the predecessor search time to $\mathcal{O}(\log\log|U|/\log\log\log|U|)$, but use $\mathcal{O}(n^4)$ space.

Furthermore, we will only consider solutions which follow the van-Emde-Boas Tree paradigm. That means that the runtime resembles a binary search on the ***word size*** w, which is the number of available bits in a memory cell. Thus, we will not cover solutions like the ***Fusion Tree*** [FW93].

Most presented solutions will use hash tables. For those solutions, we will assume perfect hash tables. The query time is always worst-case and only the update operation or preprocessing time needs amortized time with high probability. Throughout the thesis all running times of update operations are considered to be amortized with high probability. We will discuss different types of hash tables for the word RAM in Section 2.5.

1.2. Related Problems

We will give a quick overview of some problems which are similar or somehow related to the predecessor problem. Let $S \subseteq U = \{0, \ldots, 2^w - 1\}$ for any fixed w and $n = |S|$.

The first one is the classic rank problem. Given an element $q \in U$, retrieve the rank of the element q in S, i.e. $rank(q) = |\{x < q \mid x \in S\}|$. There is an easy reduction to the predecessor problem for the static case. In the preprocessing, assign each $x \in S$ its rank. Thus, the rank of a query $q \in U$ can be determined by a predecessor search of q in S. If no predecessor p exists, return rank 0, otherwise return the rank of p plus one. The opposite reduction is also easy. Store the set S in sorted order in an array A with size $n+1$ and with $A[-1] = \perp$. The predecessor search of q simply returns $A[rank(q) - 1]$. Pătraşcu and Thurop [PT14] gave a dynamic data structure, which answers rank queries as well as predecessor searches in $\mathcal{O}(\log_w n)$ time. They also mention that this is optimal for the rank problem and predecessor problem if the word size w is big. In this case, the runtime of a predecessor search is similar to the performance of the fusion tree.

Another problem is the order preserving perfect hash map [Bel+09], where a set $S \subseteq U$ of keys is mapped into the set $\{0, \ldots, m-1\}$ with $m \geq n$ and where the mapping has the same ordering as the set S. We will later revisit this data structure in Section 6.

Priority queues are also an example for a related problem. The priority queue supports the *deletemin*() and *min*() operations. It can also be reduced to the predecessor problem. Note, that *min*() is basically a successor search, i.e. $min() = search(0)$ or $successor(0)$. Therefore, *deletemin*() is the same as removing *min*() from S.

The last related problem is sorting. It is easy to see that sorting can be reduced to the predecessor problem. An interesting fact is that priority queues and sorting are basically the same problem. Han and Thorup [HT02] wrote "We note that the dynamic aspects are already pretty settled, [...] priority queues have once and for all been reduced to sorting.". This quote is based on the result by Thorup [Tho07].

1.3. Known Lower and Upper Bounds

There are different lower and upper bounds for different machine models and available space. Most lower bounds are for the static predecessor problem, because they also apply to the dynamic predecessor problem. An extended overview about lower and upper bounds and related work of the predecessor problem was given by Beame and Fich [BF02].

Comparison Model

In the comparison model, a height balanced binary tree trivially solves the predecessor problem in $\mathcal{O}(\log n)$ time. It is common knowledge that sorting requires $\Omega(n \log n)$ time in the comparison model. Thus, the predecessor problem needs $\Omega(\log n)$ time per operation, since sorting can be reduced to the predecessor problem.

Pointer Machine Model

Van Emde Boas [Boa77] showed that the predecessor problem can be solved in $\mathcal{O}(\log\log|U|)$ time and $\mathcal{O}(|U|)$ space. The lower bound for the time is $\Omega(\log\log|U|)$ [MNA88; Mul09] and $\Omega(|U|)$ for the space, as claimed without reference by Beame and Fich [BF02].

Van Emde Boas named his data structure ***stratified tree***. It is the predecessor of the van-Emde-Boas tree and initially used a non-recursive technique, avoided pointer address calculation and multiplications [Boa13]. We will not include this solution in this thesis, because it needs $\Theta(|U|)$ space and has tight bounds.

Word RAM

We will later explain what the word RAM is, but for now let us just look at the known bounds. In the word RAM the lower bound depends on the parameters n, the number of elements $|S|$, w, the word size (i.e. the number of available bits in a memory cell), and the usable space, i.e. how much space the data structure is allowed to use.

If we allow only linear space, most solutions in this thesis achieve the time bound $\mathcal{O}(\log w) = \mathcal{O}(\log\log|U|)$, e.g. the van-Emde-Boas tree in Section 3. Another approach is the fusion tree [FW93], which is based on an (a, b)-tree and yields $\mathcal{O}(\log_w n) = \mathcal{O}(\frac{\log n}{\log w})$ time per operation. When combining both solutions, the runtime for the predecessor search is $\mathcal{O}(\min\{\log w, \log_w n\}) = \mathcal{O}(\sqrt{\log n})$. Thus, if $w < 2^{\sqrt{\log n}}$ the van-Emde-Boas tree is used, otherwise the fusion tree is chosen.

Pătraşcu and Thurop [PT07] showed that the predecessor search needs $\Omega(\log w)$ time if $w = \Theta(\log n)$. The same holds even when allowing $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ space and randomization. Thus,

van-Emde-Boas Trees are optimal for these parameters. They also showed that the fusion tree is optimal for some different parameters.

If we allow polynomial space, i.e. $\mathcal{O}(n^k)$ for any $k > 1$, Beame and Fiche [BF99] improved the search time of the van-Emde-Boas tree to $\mathcal{O}(\frac{\log w}{\log \log w})$. This, combined with the fusion tree, yields a predecessor data structure which answers a predecessor search in $\mathcal{O}\left(\min\{\log_w n, \frac{\log w}{\log \log w}\}\right) = \mathcal{O}\left(\sqrt{\frac{\log n}{\log \log n}}\right)$ time. They also showed a matching lower bound.

1.4. Section Overview

In Section 2 we will introduce the w-bit word RAM, the $\boldsymbol{AC^0}$ instruction set, show some bit tricks which we will later use extensively, review the trie and compact trie under the aspect of later covered trie based solutions and discuss different types of hash tables.

In Section 3 we will present a van-Emde-Boas tree solution and show that it achieves linear space by using an ***indirection*** trick, which we will describe later in Section 5. A similar indirection trick was first used by Melhorn and Näher [MN90]. Furthermore, we will restate another argument for linear space by Pătraşcu [Păt10b].

In Section 4 we will review the X-fast trie by Willard [Wil83]. Even though it is not space optimal, it is the base idea for all following sections.

In Section 5 we will have a look at the Y-fast trie by Willard [Wil83]. We will give a detailed proof of the indirection and give a detailed amortized time analysis.

In Section 6 we will present Z-fast tries and describe a new technique for the dynamic problem. The name of the data structure was given by Belazzougui *et al* [Bel+09] and was independently invented by Ružić [Ruž09] and Belazzougui *et al* [Bel+09]. A dynamic version of the data structure was first given by Belazzougui *et al* [BBV10].

In Section 7 we will give the details of a proof sketch given by Pătraşcu in one of his blog posts [Păt10c]. According to the previous naming schema, we named the data structure μ-fast trie.

In Section 8 we will show a new data structure, called Δ-fast trie, which is based on a data structure given by Bose *et al* [Bos+13]. Their data structure needs for the static operations of the predecessor problem $\mathcal{O}(\log \log \Delta)$ worst-case time and for the dynamic operations $\mathcal{O}(\log \log \Delta)$ expected time, where Δ is the distance between the query q and its neighbor element in S. Since the space usage is $\mathcal{O}(n \log \log \log |U|)$, they left the problem open whether the same time bounds can be achieved by using linear space or not. We will show that Δ-fast tries have the same time bounds and uses linear space.

Table 1 gives a quick overview of the differences between the data structures.

1.5. Contributions

I assembled different data structures which solve the predecessor problem on a w-bit word RAM and gave an overview of them. For each data structure I presented proofs which were all developed by me, except where stated otherwise. For every data structure I gave algorithms as pseudo code.

Furthermore, the data structures (except the Y-fast trie) were implemented and tested as a prototype, which can be found on `https://www.github.com/marehr/veb-data-structures`.

While comparing the data structures, it became clear that the predecessor search for trie based approaches have two main problems. The first one is finding the ***lowest common ancestor***, the second one is finding the ***minimal*** or ***maximal element*** of a subtree. Most authors either considered the static case, where the second problem can be solved in constant time (for example in X-fast tries [Wil83], Y-fast tries [Wil83], μ-fast tries [Păt10c]), or used complex techniques such as jump pointers in Z-fast tries [BBV10] or a combination of Y-fast tries, X-fast tries and skip lists [Pug90] for Δ-fast tries [Bos+13]. Using the insight that minimal elements of subtrees decompose the trie into paths, I applied standard algorithms for path decomposition to solve the second problem. For example, regarding Z-fast tries I modified tango trees to manage the minimal and maximal elements in $\mathcal{O}(\log\log|U|)$ time and regarding Δ-fast tries I further modified the tango trees to manage those elements in $\mathcal{O}(\log\log\Delta)$ time.

Last but not least, I solved an open problem regarding the space usage stated by Bose *et al* [Bos+13] which resulted in the Δ-fast trie.

Table 1: Comparison of the different data structures in this thesis.

Data Structure	**Runtime per operation** Static	Dynamic	**Space usage**	**Preprocessing time**[‡]
non-compact trie	$\mathcal{O}(\log\|U\|)$	$\mathcal{O}(\log\|U\|)$	$\mathcal{O}(n\log\|U\|)$	$\mathcal{O}(n\log\|U\|)$
compact trie	$\mathcal{O}(\log\|U\|)$	$\mathcal{O}(\log\|U\|)$	$\Theta(n)$	$\mathcal{O}(n)^*$
van-Emde-Boas tree	$\mathcal{O}(\log\log\|U\|)$	$\mathcal{O}(\log\log\|U\|)$	$\Theta(n)$	-
X-fast trie	$\mathcal{O}(\log\log\|U\|)$	$\mathcal{O}(\log\|U\|)$	$\mathcal{O}(n\log\|U\|)$	$\mathcal{O}(n\log\|U\|)$
Y-fast trie	$\mathcal{O}(\log\log\|U\|)$	$\mathcal{O}(\log\log\|U\|)$ amortized	$\Theta(n)$	$\mathcal{O}(n)^*$
Z-fast trie	$\mathcal{O}(\log\log\|U\|)$	$\mathcal{O}(\log\log\|U\|)$	$\Theta(n)$	$\mathcal{O}(n)^*$
μ-fast trie	$\mathcal{O}(\log\log\|U\|)$	-	$\Theta(n)$	$\mathcal{O}(n\sqrt{\log\|U\|})$
Δ-fast trie	$\mathcal{O}(\log\log\Delta)^\dagger$	$\mathcal{O}(\log\log\Delta)^\dagger$ expected, $\mathcal{O}(\log\log\|U\|)$ worst-case	$\Theta(n)$	$\mathcal{O}(n\log\log\log\|U\|)^*$

[*] If the input S is sorted.
[†] Let $q^+ := \min\{s \in S \mid s \geq q\}$ and $q^- := \max\{s \in S \mid s \leq q\}$. $\Delta = \min\{|q - q^-|, |q - q^+|\}$.
[‡] For a given set S, how long does it take to build the static version of the data structure for S.

2. Preliminaries

2.1. w-bit Word RAM and the AC^0 Instruction Set

All the presented solutions in this thesis are based on the w-bit ***word RAM*** model. A word RAM is the same as a normal RAM with the exception that each memory cell has restricted space. To be precise, a memory cell can store a single w-bit word, which for example can be used to represent the integer set $\{0, \ldots, 2^w - 1\}$.

Furthermore, we will use the $\boldsymbol{AC^0}$ instruction set as much as possible. This set intuitively speaking only consists of "cheap" operations like in the C programming language, e.g. addition, subtraction, exclusive or $\oplus$, left shift $\ll$ and right shift $\gg$, but disallows multiplication [BH89], division [BCH86] and modulo[1] [Smo87] operations. Note that a division by 2^k or a multiplication by 2^k are respectively only a right shift by k and a left shift by k.

Definition 2.1 ([Tho03]) *A function $f : \{0,1\}^w \to \{0,1\}^w$ is in $\boldsymbol{AC^0}$, if it can be implemented by a constant depth, unbounded fan-in (AND, OR, NOT)-**circuit** of size $w^{\mathcal{O}(1)}$.*

The combination of a w-bit word RAM with the $\boldsymbol{AC^0}$ instruction set is called the $\boldsymbol{AC^0}$ ***RAM*** or sometimes ***Practical RAM*** [Ruž09].

Remark 2.2 *In this thesis, we want to use the $\boldsymbol{AC^0}$ instruction set as much as possible, but a fundamental result by Andersson et al [And+96] showed that a hash table using the $\boldsymbol{AC^0}$ instruction set has a circuit depth of at least $\Omega(\log w / \log\log w)$ for a constant query time. This depth is the same as it is needed to implement the multiplication and division operations. Thus, the algorithms in this thesis only work on the more general w-bit word RAM.*

Most significant bit

One of the most fundamental operation throughout this thesis is finding the most significant set bit.

Definition 2.3 *The **most significant set bit** operation of a w-bit word returns the index k of a w-**bit-sequence** $a_0 \ldots a_{w-1}$, where the prefix $a_0 \ldots a_{w-k}$ is $0^{w-k}1$. Denote the operation as $k = \mathrm{msb}(a)$.*

Remark 2.4 *The special case $\mathrm{msb}(a) = 0$ occurs if the sequence consists only of zeros.*

In the proof of fusion trees [FW93] Fredman and Willard show how to implement the most significant set bit operation in constant time with a constant number of multiplications. It seems to be common knowledge that the most significant bit operation is an $\boldsymbol{AC^0}$ operation and every paper that I read mentions this without giving a source. For example Thorup [Tho03] argues how to implement a fusion tree with only $\boldsymbol{AC^0}$ operations, but assumes that the most significant set bit operation is within the instruction set. Thus, we will give a proof.

Lemma 2.5 *The most significant set bit operation of a given bit-sequence $a = a_0 \ldots a_{w-1}$ is an $\boldsymbol{AC^0}$ operation.*

[1] This only holds for $x \mod p$, where $p \neq 2^m$ for some m, since $x \mod 2^m$ is an $\boldsymbol{AC^0}$ operation.

Proof. Assume that the most significant set bit is at position $w-k+1$, i.e. $a_0 \dots a_{w-k} = 0^{w-k}1$.

For each $j = 0, \dots, w$ let $bin_j : \{0,1\} \to \{0,1\}^w$ be a circuit which returns the number j as a binary number in a bit-sequence. For example, the circuit $bin_5(a_0)$ outputs the bit-sequence $0^{w-3}101$ for the number 5, while only using the circuits $0 = a_0 \wedge \neg a_0$ and $1 = a_0 \vee \neg a_0$.

Further let $pre_j : \{0,1\}^w \to \{0,1\}^w$ be a circuit for each $j = 0, \dots, w-1$ which returns the bit-sequence 1^w if the prefix of $a_0 \dots a_{w-1}$ is 0^j1 and 0^w otherwise. I.e.,

$$pre_j(a_0, \dots, a_{w-1}) = \left(\bigwedge_{0 \le i < j} (\neg a_i \wedge a_j), \dots, \bigwedge_{0 \le i < j} (\neg a_i \wedge a_j) \right) .$$

The circuit can compute $\bigwedge_{0 \le i < j} (\neg a_i \wedge a_j)$ once and use the same output wire w times.

Note that only one $pre_j(a_0, \dots, a_{w-1})$ will return the bit-sequence 1^w, because there is only one prefix of the form 0^j1. Thus,

$$\mathrm{msb}(a_0, \dots, a_{w-1}) = \bigvee_{0 \le j \le w-1} (pre_j(a_0, \dots, a_{w-1}) \wedge bin_{w-j}(a_0))$$

will select the term $pre_j(a_0, \dots, a_{w-1}) \wedge bin_{w-j}(a_0)$ for $j = w-k$ and will return $bin_{w-(w-k)}(a_0) = bin_k(a_0)$, which is the index k of the most significant set bit.

Since the circuit has constant depth and uses a polynomial number of (AND, OR, NOT)-circuits, the most significant set bit operation is an $\boldsymbol{AC^0}$ operation. □

Corollary 2.6 *The **least significant set bit** can be computed in constant time.*

Proof. As mentioned in [PT14], $\mathrm{lsb}(x) = \mathrm{msb}((x-1) \oplus x)$. □

Corollary 2.7 $\lfloor \log x \rfloor$ *and* $\lceil \log x \rceil$ *can be computed in constant time.*

Proof. There exists exactly one unique integer k which fulfills the inequality $x/2 < 2^k \le x$. Both $\mathrm{msb}(x) - 1$ and $\lfloor \log x \rfloor$ are integers and fulfill the inequality. Thus, $\mathrm{msb}(x) - 1 = \lfloor \log x \rfloor$. Furthermore, $\lceil \log x \rceil$ can be computed in $\mathcal{O}(1)$ time, too. Because $\lceil \log x \rceil = \lfloor \log x \rfloor + \delta$, where $\delta := 0$ if $2^{\lfloor \log x \rfloor} = x$ and $\delta := 1$ otherwise. □

2.2. Sequence and Bit Operations

Throughout the thesis, we will switch between the notion of a number and its representation as a bit string. Furthermore, we will interpret the bit string as a word over the formal language $\bigcup_{0 \le i \le w} \{0,1\}^i$ as defined in "Introduction to the Theory of Computation" [Sip96] and we want to have a representation of those strings.

Definition 2.8 *A bit string $a_0 \dots a_{m-1}$ is represented as a pair $a = (\mathcal{A}, m)$ of w-bit words. m is the length of the string and $\mathcal{A} = \sum_{i=1}^{m} a_{m-i} 2^{i-1}$ or $\mathcal{A} = 0^{w-m} a_0 \dots a_{m-1}$ (given as w-bit string) is the value of the string. The domain of $\mathcal{A}$ is $[0, 2^m)$, whereas the domain of m is $[0, w)$. A special string is the empty bit string $\varepsilon = (0, 0)$.*

It is easy to see that all operations on w-bit words can be applied to bit-sequences. Simply perform the operation on $\mathcal{A}$ (a w-bit word) and after that take the last m bits of the result by using a mask on those bits (or alternatively take the suffix of length m).

Lemma 2.9 *The following standard operations on strings (in formal languages) can be done in $\mathcal{O}(1)$ time (independent of k, m, m' and w).*

(1) The prefix operation $pre(a, k)$ returns the bit-sequence $a_0 \dots a_{k-1}$ with length k of a string $a = a_0 \dots a_{m-1}$,

(2) the suffix operation $suf(a, k)$ returns the bit-sequence $a_{m-k} \dots a_{m-1}$ with length k of a string $a = a_0 \dots a_{m-1}$,

(3) the split operation $b, c = split(a, k)$ splits a string $a = a_0 \dots a_{m-1}$ at the position k into two strings $b = a_0 \dots a_{k-1}$ and $c = a_k \dots a_{m-1}$,

(4) the concatenate operation $c = concat(a, b)$ concatenates two strings $a = a_0 \dots a_{m-1}$ and $b = b_0 \dots b_{m'-1}$ and returns $c = a_0 \dots a_{m-1} b_0 \dots b_{m'-1}$, assuming that $m + m' \leq w$, and

(5) the longest common prefix operation $\mathrm{lcp}(a, b)$ delivers the longest common prefix of two strings $a = a_0 \dots a_{m-1}$ and $b = b_0 \dots b_{m'-1}$.

Proof.

(1) Right-shift $\mathcal{A}$ by $m - k$ positions, s.t. $\mathcal{B} = \mathcal{A} \gg (m - k) = 0^{w-m} 0^{m-k} a_0 \dots a_{k-1}$. The final string is $(\mathcal{B}, m - k)$.

(2) Construct the bit-mask $0^{w-k} 1^k = (1 \ll k) - 1$ and mask out the last k bits of $\mathcal{A}$. The resulting string is $(\mathcal{A} \wedge 0^{w-k} 1^k, k)$.

(3) Simply return the prefix with length k and the suffix with length $m - k$.

(4) Left-shift $\mathcal{A}$ by m' positions and add it to $\mathcal{B}$. The string is $c = ((\mathcal{A} \ll m') \oplus \mathcal{B}, m + m')$.

(5) W.l.o.g assume that $m' \geq m$. The longest common prefix has at most length m. Therefore it suffices to determine the longest common prefix of the strings a and $b' = pre(b, m)$. $\mathcal{A} \oplus \mathcal{B}'$ yields a bit string $\mathcal{C} = 0^{w-n} 0^{n-k} 1 (0|1)^{k-1}$, where $w - k$ is the length of the longest common prefix on w-bits. k can be determined by the most significant set bit operation and therefore the longest common prefix is $pre(a, m - \mathrm{msb}(\mathcal{C}))$.

Each operation consists of a constant number of operations and each operation takes constant time. □

Definition 2.10 *Let $\lfloor\!\lfloor x \rfloor\!\rfloor$ denote $2^{\lfloor \log x \rfloor}$ and let $\lceil\!\lceil x \rceil\!\rceil$ denote $2^{\lceil \log x \rceil}$.*

Lemma 2.11 *$\lfloor\!\lfloor x \rfloor\!\rfloor$ and $\lceil\!\lceil x \rceil\!\rceil$ can be computed in constant time.*

Proof. Since $\lfloor\!\lfloor x \rfloor\!\rfloor = 2^{\lfloor \log x \rfloor} = 1 \ll (\lfloor \log x \rfloor)$ and $\lfloor \log x \rfloor = \mathrm{msb}(x) - 1$, both are constant time operations. □

Lemma 2.12 *Assuming w is a power of two, then $\lfloor\!\lfloor \sqrt{w} \rfloor\!\rfloor$ and $\lceil\!\lceil \sqrt{w} \rceil\!\rceil$ are within a constant factor of $\sqrt{w}$ and $\lfloor\!\lfloor \sqrt{w} \rfloor\!\rfloor \cdot \lceil\!\lceil \sqrt{w} \rceil\!\rceil = w$. Furthermore, $\lfloor\!\lfloor \sqrt{w} \rfloor\!\rfloor$ as well as $\lceil\!\lceil \sqrt{w} \rceil\!\rceil$ can be computed in $\mathcal{O}(1)$ time.*

Proof. If the exponent $w = 2^{2x+1}$ is odd, then $\lceil\!\lceil \sqrt{w} \rceil\!\rceil = 2^{x+1}$ and $\lfloor\!\lfloor \sqrt{w} \rfloor\!\rfloor = 2^x$ are within a constant factor of $\sqrt{w} = \sqrt{2} \cdot 2^x$ and $\lfloor\!\lfloor \sqrt{w} \rfloor\!\rfloor \cdot \lceil\!\lceil \sqrt{w} \rceil\!\rceil = 2^{2x+1} = w$. Otherwise, if the exponent in $w = 2^{2x}$ is even, then $\lceil\!\lceil \sqrt{w} \rceil\!\rceil = \lfloor\!\lfloor \sqrt{w} \rfloor\!\rfloor = 2^x$.

Therefore, to compute the square root, we first obtain the exponent $y = 2x + 1$ or $y = 2x$ of w by using the most significant set bit operation and then construct the square root $\lfloor\!\lfloor \sqrt{w} \rfloor\!\rfloor = 2^x =$

$1 \ll x$ with a shift by $x = \lfloor y/2 \rfloor = y \gg 1$. Furthermore to construct $\lceil\!\lceil \sqrt{w} \rceil\!\rceil$ we multiply $\lfloor\!\lfloor \sqrt{w} \rfloor\!\rfloor$ by two, if the exponent was odd. All of these are constant time operations. □

2.3. Binary Tries

Most solutions in this thesis are based on ***tries***. A trie $G = (V, E)$ normally stores strings and has to mark a node or needs a special symbol to indicate stored prefixes. In our setting the binary trie will store the elements $S \subseteq U$ in the bit-representation and a special symbol is not necessary, because the elements in S are always in the leaves. We will assume that each node $v \in V$ has a key attribute (denoted as $v.key$), which contains the whole prefix to that node (see Figure 1a for an example). We will abuse the notation between a node v and the key $v.key$ of that node. That means, a node can act as the address to that node (i.e. the key of that node), like in "the node 01001..." actually means "the node with the key 01001...", or a node v acts as a string, like in "the longest common prefix of q and v" actually means "the longest common prefix of q and $v.key$".

Definition 2.13 *We call a node in a trie a* ***branch node****, if it has two children and a node an* ***inner node****, if it has at least one child.*

Lemma 2.14 *A binary trie is a binary search tree with the following order*

$$a \leq b, \text{ iff } a^* < b^* \text{ or } (a^* = b^* \text{ and } |a| \leq |b|), \text{ where}$$

$$a^* = a_0 \dots a_{|a|-1} \underbrace{10\dots0}_{w-|a|} \text{ and } b^* = b_0 \dots b_{|b|-1} \underbrace{10\dots0}_{w-|b|} .$$

Proof. Notice that the ordering is correct for leaves. For non-leaves the suffix $10\dots0$ will be appended. This basically maps a non-leaf to its successor leaf (the *one* bit represents the right subtree and the following *zero* bits represent the left most child). Since the leaves are correctly ordered, the order of a^* and b^* is correct. The only special case is, if a node a is compared to the successor node b of a, because they both map to b. For example in Figure 1a the nodes $a = 001$ and $b = 0011$ are both mapped to 0011, but a is in the order of the binary search tree before b. Thus, the shorter key is further up in the tree and therefore earlier in the ordering. □

2.3.1. Lowest Common Ancestor

We will see that basically all operations of a trie $G = (u, V)$ can be reduced to the lowest common ancestor operation. This will have the effect, that we will later concentrate mostly on the implementation of the lowest common ancestor, while in most cases the data structures can be maintained with only a little extra effort.

Definition 2.15 *The* ***longest common prefix*** *of an element $q \in U$ is the longest common prefix among q and all nodes*

$$\text{lcp}(q) = \max\{\text{lcp}(q, v) \mid v \in V\} \quad .$$

Alternatively we could define $\text{lcp}(q)$ as $\max\{\text{lcp}(q, s) \mid s \in S\}$, but we want to make it clear that the following definition of the lowest common ancestor is similar to the longest common prefix.

Definition 2.16 *The* ***lowest common ancestor*** *of an element $q \in U$ is the node with the longest matching node-to-root path of q and all nodes. In short*

$$\mathrm{lca}(q) = \max\{v \in V \mid \mathrm{lcp}(q, v) = v\} \quad .$$

Note, that $\mathrm{lcp}(q)$ is the key of the node $\mathrm{lca}(q)$ (see Figure 1a). We will later see that for compact binary tries both definitions are different.

Lemma 2.17 *If the lowest common ancestor u of a query q is an inner node, then u has exactly one child and q belongs to the empty subtree rooted at the other child.*

Proof. u has at least one child, because u is an inner node with a key that has less than w-bits. Thus, there must be an element $s \in S$ with the prefix u, where the root-to-leaf path contains u. Assume that u has two children. In that case belongs q to a child of u, but then the child $u' = u_0 \dots u_{|u|-1} q_{|u|}$ of u has a longer common prefix with q than u, a contradiction. Thus, u has exactly one child and that q belongs to the empty subtree of the sibling. □

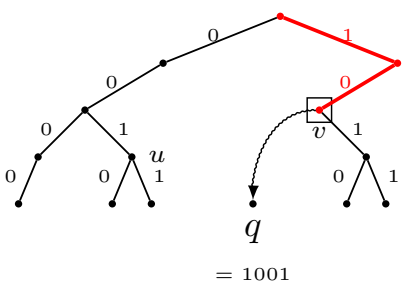
Corollary 2.18 *The lowest common ancestor u of a query q has at most one child.*

Proof. We have seen in the previous lemma that for an inner node u the claim holds. Thus, if u is not an inner node, then it is a leaf, but for this case the claim trivially holds. □

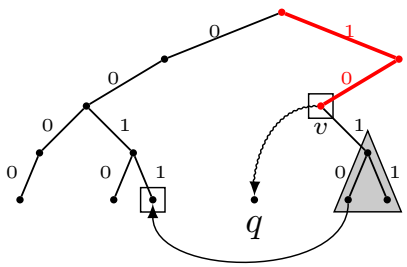
We will now show, that all operations can be reduced to the lowest common ancestor operation. Let $T_{\mathrm{lca}}(q)$ be the runtime to find the lowest common ancestor of q.

2.3.2. Search

Searching an element or doing a membership query is quite easy. If a query q belongs to S, the lowest common ancestor will return the leaf v of q, since v has the longest common prefix with q.



(a) Binary Tries - The trie stores the set $S = \{0, 2, 3, 10, 11\}$ and the keys of u and v are respectively 001 and 10. Let q be 1001. The node v is the lowest common ancestor of q and 10 is the longest common prefix of q.



(b) Binary Tries - Let q be 1001. The predecessor of q is clearly 0011. First find the minimal node of the subtree rooted at the right child of v (i.e., the successor of q) and then follow the link to the predecessor.

Figure 1: Binary Tries - The definition of lca and lcp and retrieving the predecessor.

Algorithm 1: Search an element in $\mathcal{O}(T_{\text{lca}}(q))$ time

```
search(T, q ∈ U):
  v = T.lca(q)

  if v is a leaf:
    return v
  return ⊥
```

2.3.3. Predecessor

The predecessor of an element q can be determined by looking at the lowest common ancestor lca. For now let us assume, that the lowest common ancestor is an inner node. If the right subtree of lca is empty, then according to the order of Lemma 2.14, the greatest element in the left subtree is the predecessor of q. Analogously, if the left subtree of lca is empty, the smallest element in that subtree is the successor of q. See Figure 1b for an example.

We can assume that the leaves are linked, so after finding the successor, the predecessor can be easily found by following the pointer. Furthermore we will assume that each inner node has a pointer to its minimal and maximal leaf. This is feasible, because we will see that updating needs $\Omega(w)$ time anyway. Thus an update operation can update all subtrees on the changed root-to-leaf path. Now let us consider the case where the lowest common ancestor is a leaf. But in this case the leaf has a pointer to its predecessor.

Thus, the runtime only depends on finding the lowest common ancestor.

Algorithm 2: The predecessor of an element in $\mathcal{O}(T_{\text{lca}}(q))$ time

```
predecessor(T, q ∈ U):
  v = T.lca(q)

  if v is a leaf:
    return v.previous_leaf

  if v.left_child ≠ ⊥:
    v_p = v.left_child.max_leaf
    return v_p

  v_s = v.right_child.min_leaf
  return v_s.previous_leaf
```

2.3.4. Successor

The successor operation is completely analogous to the predecessor operation.

Algorithm 3: The successor of an element in $\mathcal{O}(T_{\text{lca}}(q))$ time

```
successor(T, q ∈ U):
  v = T.lca(q)

  if v is a leaf:
    return v.next_leaf

  if v.right_child ≠ ⊥:
    v_s = v.right_child.min_leaf
    return v_s

  v_p = v.left_child.max_leaf
  return v_p.next_leaf
```

2.3.5. Insertion

The insertion of an element q has three main parts. In the first part, the missing nodes have to be inserted. We have seen in Lemma 2.17, that a query q goes into an empty subtree of a child

from the lowest common ancestor u. That means, that the prefixes $q_0 \dots q_{|u|}, \ \dots \ , q_0 \dots q_{|w|-1}$ are missing in the trie and have to be inserted starting from the lowest common ancestor.

In the second part, the minimal and maximal leaves of all nodes on the effected leaf-to-root path need to be updated. And in the last part, the leaves have to be chained together. This can be done by finding the predecessor and successor nodes.

Now, let us analyze the runtime. There will be $w - |lca(q)|$ new nodes added to the trie during an insertion, but since the lowest common ancestor can be any non-branch node, e.g. the root[1], the runtime is $\Theta(w)$.

Algorithm 4: Inserting an element in $\Theta(w)$ time

```
insert(T, q ∈ U \ S):
  u = T.lca(q)

  # add each missing prefix of q
  for p = q_0 ... q_|u|, ... , q_0 ... q_|w|-1:
    add new node with key p

  # update the min_leaf and max_leaf
  # pointers in O(w) time.

  v_p = T.predecessor(q)
  v_s = T.successor(q)

  # chain leaves
  insert q between v_p and v_s
```

2.3.6. Deletion

Deleting an element q is easier than inserting. First search the leaf of q, then remove it from the linked leaves. Afterwards remove all non-branch parents and update all minimal and maximal leaves. The runtime is $\Theta(w)$ and has the same worst-case example as the insertion.

Algorithm 5: Removing an element in $\Theta(w)$ time

```
remove(T, q ∈ S):
  u = T.search(q)

  remove u from the chained leaves

  # remove all dangling prefixes
  # of q

  repeat until the first branch node
    remove u
    u = u.parent

  # update the min_leaf and max_leaf
  # pointers starting from u.
```

2.3.7. Result

Theorem 2.1 *All static and dynamic predecessor operations can be reduced to the lowest common ancestor, where the static predecessor operations have a runtime of $\mathcal{O}(T_{\text{lca}}(q))$, while the update operations take $\Theta(w)$ time. The space usage is $\mathcal{O}(n \cdot w) = \mathcal{O}(n \log |U|)$.*

Proof. We have seen the runtime of the operations in the Algorithms 1 to 5. Let us consider the space usage. The trie has for each leaf at most w prefixes inserted, that means the trie has at most $n \cdot w$ nodes. Therefore, the space usage is $\mathcal{O}(n \log |U|)$. □

[1] A worst-case example: If all elements have the prefix 0 and the next element to be inserted has the prefix 1.

2.4. Compact Binary Tries

The space usage of a trie is comparatively high and the runtime of the update operation does not leave a lot of room for improvement. An interesting idea is to use compact binary tries, which are also called PATRICIA trees. See Figure 2a for an example of a compact binary trie.

Normally, compact tries only save space, while the runtime of the operations stays the same, because each edge label has to be compared character by character. But for the w-bit word RAM, we have seen that each bit-sequence can be compared in constant time (e.g. Lemma 2.9). That means, that highly compressed edges on a root-to-leaf path results in a faster runtime. Therefore, the worst-case runtime happens if no compression occurs.

Remark 2.19 *A branch node in a non-compact trie is also a branch node in the corresponding compact trie of the same set S, whereas a non-branch node in a non-compact trie is a part of an edge in the compact trie.*

Lemma 2.20 *Each inner node (except the root) is a branch node.*

Proof. Assuming an inner node has only one child, the edge label of the inner node could be added to the child and thus the trie was not compact, a contradiction. □

Definition 2.21 *Let $e = (u, v) \in E$ be an edge in the compact trie. We denote as $p \in e$ a string p that is on the edge e. Formally, $p \in e$ iff $u.key < p \leq v.key$ in lexicographical order and u is a prefix of p and p is a prefix of v.*

For the example let us take the Figure 2a as reference. Let u be the root and v be the node with the key 101 and $e = (u, v)$ the edge of the nodes. In this case are $1, 10, 101 \in e$, because they are on the edge e.

2.4.1. Lowest Common Ancestor

The following definition of the lowest common ancestor is the same as for non-compact tries.

Definition 2.22 *The **lowest common ancestor** of an element $q \in U$ is the node with the longest matching node-to-root path of q and all nodes. In short*

$$\mathrm{lca}(q) = \max\{v \in V \mid \mathrm{lcp}(q, v) = v\} \quad .$$

The difference is that the lowest common ancestor and the longest common prefix were the same, but now they are different (see Figure 2a).

Corollary 2.23 (of Lemma 2.17) *The longest common prefix*

$$\mathrm{lcp}(q) = \max\{\mathrm{lcp}(q, v) \mid v \in V\}$$

is not a node in the trie, except when the prefix is empty or q is in S.

Proof. We know, that each inner node in the compact trie is a branch node and therefore also a branch node in the corresponding non-compact trie and vice versa. Lemma 2.17 states, that the lowest common ancestor (i.e. the longest common prefix) in a non-compact trie is a non-branch

node. Thus, the longest common prefix is part of an edge in the compact trie and not a node in the compact trie. □

So when inserting an element, there will be two additional nodes. One is the node for the longest common prefix, because it will become a branch node in the non-compact trie according to Algorithm 4 and the second inserted node is a leaf, which represents q.

As in the non-compact trie case, all operations can be reduced to the lowest common ancestor. This is not surprising, since both tries correspond to each other. In contrast to the non-compact trie, we will not assume that the minimal and maximal leaves can be found in constant time. We will discuss in the later sections, which runtime the management of the minimal and maximal leaves have.

2.4.2. Search

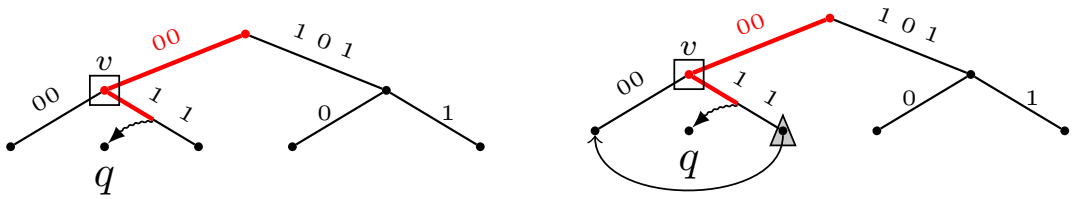
The searching algorithm is the same as for non-compact tries.

Algorithm 6: Search an element in $\mathcal{O}(T_{\text{lca}}(q))$ time

```
search(T, q ∈ U):
  v = T.lca(q)

  if v is a leaf:
    return v
  return ⊥
```

2.4.3. Predecessor and Successor

We will imitate the behavior of the predecessor search in the non-compact trie. The predecessor search in non-compact tries (Algorithm 2) determined the predecessor by checking which child of the longest common prefix is present, i.e. $\neq \perp$. We know, that the node of the longest common prefix in the non-compact trie has only one child[1], so in the compact trie the longest



(a) Compact Binary Tries - Let q be 0010. The lowest common ancestor of q is v and the longest common prefix of q is 001.

(b) Compact Binary Tries - Let q be 0010. The predecessor of q is clearly 0000. First find the minimal node of the subtree rooted at the right child of v (since the $|\operatorname{lcp}(q)| + 1 = 4$-th bit is *one*) and then go to the predecessor.

Figure 2: Compact Binary Tries - The definition of lca and lcp and retrieving the predecessor.

[1] This is only true if it is an inner node in the non-compact trie (Lemma 2.17). We will shortly after that consider what happens if it is not an inner node.

common prefix is part of an edge $e = (u, v)$. We further know, that u is the lowest common ancestor.

Therefore, we can emulate which child of the longest common prefix p is present, by looking at the $(|lcp| + 1)$-th bit of $v.key$. The value of the $(|lcp| + 1)$-th bit of $v.key$ is the corresponding child of the longest common prefix in the non-compact trie, i.e. if the bit value is *zero* the left child is present in the non-compact trie and if the value is *one* the right child is present. See Figure 2b for an example.

The search has one special case when the longest common prefix is not part of an edge, i.e. is not an inner node in the non-compact trie. This is the case, when either the query q is contained in S or the longest common prefix is ε. The case $q \in S$ is trivial, because then the lowest common ancestor is a leaf and the predecessor is the preceding leaf. But, the case $lcp = \varepsilon$ needs more attention. In this case, q is not part of an edge and therefore the $(|lcp| + 1) = 1$-th bit can not be accessed. Thus, we will need to directly determine the predecessor from the root.

As earlier mentioned, we will not assume a pointer from a subtree to the minimal and maximal leaf. Let $T_{\min}(q)$ denote the time to find those leaves.

Algorithm 7: The predecessor of an element in $\mathcal{O}(T_{\text{lca}}(q) + T_{\min}(q))$ time

```
predecessor(T, q ∈ U):
  u = T.lca(q)

  if u is a leaf:
    return u.previous_leaf

  v = u's child in direction of q
  lcp = lcp(q, v)

  # lcp = ε and u is the root
  if v = ⊥ and u.left_child ≠ ⊥:
    return u.left_child.max_leaf
  if v = ⊥ and u.left_child = ⊥
    return ⊥

  let v_0 ... v_{|v|-1} be the bit sequence
      of v

  # (lcp+1)-th bit of v is zero and
  # thus the left child of the lcp
  # is present.
  if v_{|lcp|} bit is zero:
    pred = v.left_child.max_leaf
    return pred

  succ = v.right_child.min_leaf
  return succ.previous_leaf
```

And for completeness' sake, the successor search.

Algorithm 8: The successor of an element in $\mathcal{O}(T_{\text{lca}}(q) + T_{\min}(q))$ time

```
successor(T, q ∈ U):
  u = T.lca(q)

  if u is a leaf:
    return u.next_leaf

  v = u's child in direction of q
  lcp = lcp(q, v)

  if v = ⊥ and u.right_child ≠ ⊥:
    return u.right_child.min_leaf
  if v = ⊥ and u.right_child = ⊥
    return ⊥

  let v_0 ... v_{|v|-1} be the bit sequence
      of v

  # (lcp+1)-th bit of v is zero and
  # thus the left child of the lcp
  # is present.
  if v_{|lcp|} bit is one:
    v_s = v.right_child.min_leaf
    return v_s

  v_p = v.left_child.max_leaf
  return v_p.next_leaf
```

2.4.4. Insertion

We have already seen that inserting an element q only adds two new nodes, i.e. when splitting the edge containing the longest common prefix, one node for the longest common prefix and one leaf node for the element q. That means that the trie can be maintained in constant time after finding the lowest common ancestor. Therefore, the runtime depends on finding the lowest common ancestor $T_{\text{lca}}(q)$ and maintaining the minimal and maximal elements $T_{\text{update-min}}(q)$.

Like in the predecessor case, there is a special case when the longest common prefix is ε. But in this case, one has to split the only available edge of the root.

Algorithm 9: Inserting an element in $\mathcal{O}(T_{\text{lca}}(q) + T_{\text{update-min}}(q))$ time

```
insert(T, q ∈ U \ S):
  u = T.lca(q)

  if S is empty:
    add a node for q below u
    return

  if u is the root:
    v = root's only child
  else:
    v = u's child in direction of q

  lcp = lcp(v, q)

  split the edge (u, v) at lcp into
      the edges (u, lcp) and (lcp, v)

  add a new node for q at node lcp
      with the edge label q_{|lcp|} ... q_{w-1}

  # update the min_leaf and max_leaf
  # pointers in T_update-min(q) time.

  v_p = T.predecessor(q)
  v_s = T.successor(q)

  # chain leaves
  insert q between v_p and v_s
```

2.4.5. Deletion

For removing an element q, one has to remove the leaf of q, its parent and after that merge the edge of the parent. This can be done in constant time, whereas the whole runtime depends again on the lowest common ancestor search $T_{\text{lca}}(q)$ and the time to update the minimal and maximal elements $T_{\text{update-min}}(q)$.

Algorithm 10: Removing an element in $\mathcal{O}(T_{\text{lca}}(q) + T_{\text{update-min}}(q))$ time

```
remove(T, q ∈ S):
  u = T.search(q)

  remove u from the chained leaves
  remove u from the trie

  if |S| > 1:
    merge the edge of u's parent

  # update the min_leaves and
  # max_leaves starting from
  # u's parent.
```

2.4.6. Linear space

We will show that a compact trie has a linear number of nodes. It follows directly that a compact trie only uses linear space.

Lemma 2.24 *A compact binary trie $G = (V, E)$ of the set S has $|V| = 2|S|$ nodes and $|E| = 2|S| - 1$ edges for $|S| \geq 1$.*

Proof. Since a compact trie is also a binary search tree (i.e., Lemma 2.14) and trees satisfy the given correlation between the number of edges and nodes, we only have to show that the number of nodes is $2|S|$.

The proof is by induction over the number of elements. The lemma is true for $|S| = 1$. Now assume $|S| > 1$. Take an arbitrary element $q \in S$ and construct the compact trie for $S \setminus \{q\}$. Per induction, the trie has $2(|S| - 1)$ nodes. Now insert q into the trie. We know that two new nodes will be inserted, thus $|V|$ is $2|S|$. □

2.4.7. Linear Time Construction

Andersson *et al.* [And+98] gave a general purpose construction for any arbitrary fixed alphabet by building a ***Cartesian tree*** [GBT84] and transforming it into a compact trie. We will give a simpler algorithm, which in fact uses the same arguments as in the linear time construction of the Cartesian tree.

Lemma 2.25 *A compact binary trie of the set S can be constructed in linear time, if the set S is sorted.*

Proof. The algorithm builds the compact trie incrementally. Insert each element $s_1, \ldots, s_n$ in sorted order into the trie. We call the root-to-leaf path of the last inserted element the ***right spine***. Note that when inserting the next element s_k the longest common prefix is somewhere on the right spine, otherwise the inserted element would not be the current maximum element. Thus, the longest common prefix $\mathrm{lcp}(q)$ among all nodes is the longest common prefix $\mathrm{lcp}(s_{k-1}, s_k)$ between s_k and s_{k-1}. As seen in Lemma 2.9.(5), $\mathrm{lcp}(s_{k-1}, s_k)$ can be computed in constant time. After knowing the position on the right spine, where the new leaf of s_k has to be attached to, we can walk up from the leaf of s_{k-1} until we find the edge e, which contains $\mathrm{lcp}(s_{k-1}, s_k)$ and insert the element s_k.

The runtime is overall linear, because after inserting the element, the right spine changes s.t. the path on the old right spine from the leaf to the longest common prefix is not part of the new right spine. Therefore, walking up the edges of the right spine will visit each of those edges only once during the construction. Since the compact trie has only a linear number of edges altogether, the construction takes linear time. □

2.4.8. Result

Theorem 2.2 *The static and dynamic predecessor problem can be solved by a compact trie, where each static operation takes at most $\mathcal{O}(T_{\mathrm{lca}}(q) + T_{\min}(q))$ time and each update operation takes $\mathcal{O}(T_{\mathrm{lca}}(q) + T_{\text{update-min}}(q))$ time, while the space usage is linear. The preprocessing time in the static predecessor case is linear on sorted input.*

Proof. The runtime is as described in the Algorithms 6 to 10 and the space usage follows from Lemma 2.24.

In the static predecessor case, the preprocessing time is linear, because according to Lemma 2.25 the compact trie can be constructed in linear time on sorted input. Furthermore, in this case the minimal and maximal leaf of a subtree can be a direct pointer, like in the non-compact trie. Thus, chaining the leaves and assigning each subtree a pointer to the minimal and maximal leaf can be done in the preprocessing time, since it only involves a pre-order depth-first traversal. □

2.5. Hash Tables

In the following sections, all algorithms use hash tables to achieve better running times. The hash tables map a set S with size $|S| = m$ from the universe $[2^w]$ to $[\mathcal{O}(m)]$, where $[i] := \{0, \ldots, i-1\}$ for any $i \geq 0$.

In Remark 2.2 we already mentioned that there is no hash table for the $\boldsymbol{AC^0}$ RAM with constant query time.

Let us restate the following types of operations used in dictionaries categorized by Demaine *et al* [Dem+06]:

Membership: Given an element x, is it in the set S?

Retrieval: Given an element x in the set S, retrieve r bits of data associated with x. (The outcome is undefined if x is not in S.) The associated data can be set upon insertion or with another update operation.

Perfect hashing: Given an element x in the set S, return the hashcode of x. The data structure assigns to each element x in S a **unique** hashcode in $[n + t]$, for a specified parameter t (e.g., $t = 0$ or $t = n$). Hashcodes are stable: the hashcode of x must remain fixed for the duration that x is in S. (Again the outcome is undefined if x is not in S.)

Membership and Retrieval Hash Tables

In Sections 3 to 6 we will need hash tables with constant query time and supporting membership and retrieval, where the associated data are the pointer addresses.

The commonly known hash family given by Carter and Wegman [CW79]

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m \quad ,$$

with a standard collision resolution, e.g. chaining or probing, support membership and retrieval in constant time. This approach will use $\mathcal{O}(m)$ space to store the data.

There exists a hash table solution for the w-bit word RAM by Demaine *et al* [Dem+06]. They solve the dynamic dictionary problem and support insertions and deletions as well as perfect hashing queries and/or membership queries. Each operation has constant time with high probability[1] and the data structure uses optimal space. For the construction of the hash table they need a 2-independent hash function, where the one given by Thorup and Zhang [TZ12] could be used.

There also exists $\boldsymbol{AC^0}$ RAM hash tables, like the one given by Pătraşcu and Thorup [PT10] and Thorup and Zhang [TZ12], which supports $\mathcal{O}(c)$ query time and uses $\mathcal{O}(c\ |U|^{1/c})$ space. But they have a super linear space usage if the query time is constant.

Retrieval Hash Tables

In Sections 7 and 8 we will need special hash tables, which only support retrieval queries. In this setting, we will assign data to each element in S, which is exactly r bits long.

[1] For any constant $c > 0$, the probability is $\geq 1 - \frac{1}{n^c}$.

There is a static dictionary solution by Pătraşcu in the Appendix A.3, which has constant query time and uses optimal $\mathcal{O}(nr)$ bits, but has a high preprocessing time. But, since the solution by Pătraşcu is only a static dictionary, we will use the following result by Demaine *et al* in Sections 7 and 8.

Theorem 2.3 ([Dem+06], Theorem 3) *For any $t \geq 0$, there is a dynamic dictionary that supports updates and perfect hashing with hash-codes in $[n + t]$ (and therefore also retrieval queries) in constant time per operation, using $\mathcal{O}(n \log\log \frac{|U|}{n} + n \log \frac{n}{t+1})$ bits of space. The query and space complexities are worst-case, while updates are handled in constant time with high probability.*

Theorem 2.4 *There is a dynamic dictionary that supports updates and retrieval queries with r bit associated data in constant time per operation, using $\mathcal{O}(n \log\log \frac{|U|}{n} + nr)$ bits of space. The query and space complexities are worst-case, while updates are handled in constant time with high probability.*

Proof. With $t = n$ the term $n \log \frac{n}{t+1}$ in the space usage disappears. Allocating an array A for $2n$ entries and r bits per entry needs additional $\mathcal{O}(nr)$ bits. The associated data can be managed by using the unique hash- codes $g : S \mapsto [2n]$ (i.e., update the data in the array $A[g(x)] := data$).□

3. Van-Emde-Boas tree

In this section, we will study the data structure called van-Emde-Boas tree and present a implementation, which is based on a implementation given in the lecture "Advanced Data Structures" of Erik Demaine [Dem12]. The data structure processes all operations in $\mathcal{O}(\log\log|U|)$ time and uses linear space. The general technique of van-Emde-Boas trees is called ***cluster galaxy decomposition*** [Boa13].

3.1. Data Structure

The data structure is a tree with high degree and is defined recursively. Each van-Emde-Boas node v stores a set $S \subseteq U$ of items over a shrinking universe U with size $|U| = u$ and has the following attributes:

clusters: There are $\sqrt{u}$ clusters $C_0, \ldots, C_{\sqrt{u}-1}$, where each cluster C_c is a van-Emde-Boas node and stores a set $S'_c \subseteq [\sqrt{u}]$ over the reduced universe $[\sqrt{u}]$. The clusters will be managed by a hash table. Let $x = c\sqrt{u} + i \in S$ and $x \cong (c, i) \in [\sqrt{u}] \times [\sqrt{u}]$ be the unique representation of x. The number x would be stored recursively in the cluster C_c represented by i.

summary: The summary stores the clusters C_c with at least one element by the identifier c, i.e. $c \in \mathit{summary}$ iff $|C_c| \geq 1$. The summary itself is a van-Emde-Boas node over the reduced universe $[\sqrt{u}]$.

min/max: min and max are respectively the minimum and maximum element of the set S. They will never be stored in any of the clusters below the node u and are the base case for the recursion. As long as $|S| \leq 2$ the node will not initialize the clusters or the summary.

Let $S_c = \{x \in S \setminus \{\min S, \max S\} \mid x \cong (c, i)\}$ be the set of elements that are in the same cluster c. Clearly, $\dot{\bigcup}_{c\in[\sqrt{u}]} S_c = S \setminus \{\min S, \max S\}$.

Since the universe U has size $|U| = 2^w$, the reduced universe has size $\sqrt{|U|} = 2^{w/2}$. Therefore, the number of bits is halved and we can compute

$$x \cong (c, i) = \left(\frac{x}{\sqrt{u}}, x \mod \sqrt{u}\right) = \left(\frac{x}{2^{w/2}}, x \mod 2^{w/2}\right)$$

in constant time by a split at position $w/2 = w \gg 1$, as seen in Lemma 2.9.(3). It is even easier to recompute the value $x = c2^{w/2} + i$ in constant time, because $x = (c \ll w/2) + i$.

We will see that all operations have the running time given by the recursion

$$T(|U|) = T(\sqrt{|U|}) + \mathcal{O}(1) = T(2^{w/2}) + \mathcal{O}(1) = \mathcal{O}(\log\log|U|) \ .$$

Since the bit size is halved in each step, the runtime is $\mathcal{O}(\log w) = \mathcal{O}(\log\log|U|)$.

As a last remark, note that the size of S can be differentiated by the following four states:

$$|S| = \begin{cases} 2 + \sum_{c\in[\sqrt{u}]} S_c, & \text{if } \mathit{summary} \neq \bot \\ 2, & \text{if } \mathit{summary} = \bot \text{ and } \mathit{min} \neq \mathit{max} \\ 1, & \text{if } \mathit{summary} = \bot,\ \mathit{min} = \mathit{max} \text{ and } \mathit{min} \neq \bot \\ 0, & \text{if } \mathit{summary} = \bot \text{ and } \mathit{min} = \mathit{max} = \bot \end{cases}$$

3.2. Static Predecessor Problem

3.2.1. Search

The search for an element q is easy. If either min or max is the searched element, we know that the van-Emde-Boas node contains q. Otherwise let $q \cong (c, i)$ and recurse into C_c by searching i in that cluster. The search procedure is correct, since we know that i is in the cluster C_c, iff x is in the data structure. The runtime can be defined recursively by $T(w) \leq T(w/2) + \mathcal{O}(1)$ and is $\mathcal{O}(\log\log|U|)$.

Algorithm 11: Search an element in $\mathcal{O}(\log\log|U|)$ time

```
search(T, q ∈ U, w):
  c, i = split(q, w/2)

  if T.min = q or T.max = q:
    return True

  cluster = T.clusters[c]
  if cluster = ⊥:
    return False

  return cluster.search(i, w/2)
```

3.2.2. Predecessor and Successor

The data structure has to deal with some special cases, before it recurses deeper. Those cases are as follows.

If S is empty it contains no predecessor of an element q. Therefore, assume $|S| > 0$. Furthermore, if $q < \min S$, then q has no predecessor in the set S and if $\max S < q$, the predecessor of q in S is $\max S$. Therefore, let $S \geq 2$ and $\min S < q \leq \max S$. We have seen that $|S| \leq 2$, iff $summary = \perp$. Hence, if $summary = \perp$ the predecessor is $\min S$.

Finally assume $|S| > 2$ and $\min S < q \leq \max S$, let $q \cong (c, i)$ and let $p \cong (c_p, i_p)$ be the predecessor of q in S. First let us consider the case, where the predecessor can be found in $S_c \neq \emptyset$. That means the minimal element in S_c is less than or equal to the predecessor, or equally $\min S_c \cong (c, \min C_c) \leq (c, i_p) < (c, i)$. Thus, the predecessor can be found by searching the predecessor i_p of i in C_c.

Otherwise, assume $p < \min S_c$. Since p is in a different cluster than c, the first non-empty cluster c_p before c contains the predecessor. This is a predecessor search on the cluster identifiers, i.e. a search in the summary. Thus, the maximal element in S_{c_p} is the predecessor of q. There is a special case when $\min S = p$. In this case, the predecessor query in the summary will fail, because the cluster identifier c_p of $\min S$ is not in the summary, since $\min S$ is not recursively stored in the clusters. But since we know that a predecessor exists ($\min S < q \leq \max S$), the search will return $\min S$.

The runtime has the recursion $T(w) \leq T(w/2) + \mathcal{O}(1)$, because in every case, there is at most one recursive call to the predecessor.

Algorithm 12: Predecessor of an element in $\mathcal{O}(\log\log|U|)$ time

```
predecessor(T, q ∈ U, w):
  # |S| = 0
  if T.min = T.max = ⊥:
    return ⊥

  if q ≤ T.min
    return ⊥

  if q > T.max:
    return T.max

  # min S < q ≤ max S and 2 ≤ |S| ≤ 2
  if T.summary = ⊥:
    return T.min

  c, i = split(q, w/2)
  cluster = T.clusters[c]

  # predecessor in the same cluster
  if cluster ≠ ⊥ and i >cluster.min:
    i_p = cluster.predecessor(i)
    p = concat(c, i_p)
    return p

  # predecessor in a diff. cluster
  c_p = T.summary.predecessor(c)
  if c_p = ⊥:
    return T.min

  i = T.clusters[c_p].max
  p = concat(c_p, i)
  return p
```

The successor search is completely analogous.

Algorithm 13: Successor of an element in $\mathcal{O}(\log\log|U|)$ time

```
successor(T, q ∈ U, w):
  # |S| = 0
  if T.min = T.max = ⊥:
    return ⊥

  if q ≥ T.max:
    return ⊥

  if q < T.min:
    return T.min

  # min S < q ≤ max S and 2 ≤ |S| ≤ 2
  if T.summary = ⊥:
    return T.max

  c, i = split(q, w/2)
  cluster = T.clusters[c]

  # successor is in the same cluster
  if cluster ≠ ⊥ and i <cluster.max:
    i_s = cluster.successor(i)
    s = concat(c, i_s)
    return s

  # successor is in a diff. cluster
  c_s = T.summary.successor(c)
  if c_s = ⊥:
    return T.max

  i = T.clusters[c_s].min
  s = concat(c_s, i)
  return s
```

3.3. Dynamic Predecessor Problem

3.3.1. Insert

When inserting an element $q \cong (c, i)$ into the data structure, two main cases have to be considered. The first one is when $|S| < 2$, because in this case no element will be stored recursively and the clusters and summary will not be initialized. The other case is when $|S| \geq 2$.

Assume $|S| < 2$. In this case the only thing to consider is to keep the order of the attributes *min* and *max* consistent with the definition. If $|S| = 0$, both *min* and *max* have to point to the same element. This case needs constant time.

Now assume $|S| \geq 2$. The first step is to ensure that, when $\min S \cup \{q\}$ or $\max S \cup \{q\}$ differ from $\min S$ or $\max S$, to update the *min* and *max* elements accordingly to the new extreme values and to recursively store the old extreme value in the *clusters*. The rest is straightforward. If

$|S| = 2$, create the *summary*, after that insert i recursively in the cluster C_c. If the cluster C_c is not in the hash table *clusters*, create it and store the cluster identifier c in the *summary*.

The runtime is recursively defined by $T(w) \leq T(w/2) + \mathcal{O}(1)$. There are two cases where both of them have one recursive call each.

Case 1: If C_c was newly created, the insertion of i into the cluster C_c takes constant time (i.e. the first case of the insertion with $|S_c| \leq 2$) and the insertion of the cluster identifier c into the summary takes at most $T(w/2)$ time.

Case 2: If the cluster identifier was already inserted into the summary (i.e. $|S_c| > 1$), there is only one recursive call for the insertion of i into the cluster C_c, which takes at most $T(w/2)$ time.

Algorithm 14: Insert an element in $\mathcal{O}(\log\log|U|)$ time

```
insert(T, q ∈ U \ S, w):
  # |S| = 0
  if T.min = ⊥:
    T.min = T.max = q
    return

  one_element = T.min == T.max

  if q < T.min:
    swap q and T.min

  if q > T.max:
    swap q and T.max

  # |S| = 1
  if one_element:
    return

  # |S| = 2
  if T.summary = ⊥:
    T.summary = new node

  # |S| ≥ 2
  c, i = split(q, w/2)
  cluster = T
      .create_or_use_cluster(c)

  # newly created cluster
  if cluster.min = ⊥:
    T.summary.insert(c, w/2)

  cluster.insert(i, w/2)
```

Note that `create_or_use_cluster(c)` either returns the van-Emde-Boas-Node $T[c]$ of the cluster c or if $T[c] = \perp$ creates a new node, stores it in $T[c]$ and finally returns it.

3.3.2. Delete

The delete procedure of an element $q \cong (c, i) \in S$ will be split into two cases. One for $|S| \leq 2$, whereas the other case is for $|S| > 2$.

Algorithm 15: Remove an element in $\mathcal{O}(\log\log|U|)$ time.

```
delete(T, q ∈ S, w):
  c, i = split(q, w/2)

  # at most two elements
  if T.summary = ⊥:
    T.delete_{|S|≤2}(q, w)
    return

  T.delete_{|S|>2}(q, w)
```

When $|S| \leq 2$, the summary is not initialized and the data structure only has to guarantee the order of $\min S \setminus \{q\}$ and $\max S \setminus \{q\}$, which can be done in constant time.

Algorithm 16: Remove procedure for $|S| \leq 2$ in $\mathcal{O}(1)$ time.

```
delete_{|S|≤2}(T, q ∈ S, w):
  # |S| = 1
  if T.min = T.max and q = T.max:
    T.min = T.max = ⊥
    return

  # |S| = 2
  if T.min = q:
    T.min = T.max
    return

  # |S| = 2
  if T.max = q:
    T.max = T.min
    return
```

Let $|S| > 2$. If either $\min S \setminus \{q\}$ or $\max S \setminus \{q\}$ differ from $\min S$ or $\max S$, then set the extreme value accordingly with the succeeding or preceding element in S. The new extreme value (now named $q \cong (c, i)$) must be removed recursively from the data structure. The rest is again straightforward. First remove i recursively in the cluster C_c. If the cluster C_c becomes empty, remove it from the summary and if the summary becomes empty, remove the summary, which indicates that S stores at most two elements.

The runtime has again the recursion $T(w) \leq T(w/2) + \mathcal{O}(1)$, because either the cluster C_c remains non-empty and the summary does not have to delete the cluster from it and therefore only has to recurse once with at most $T(w/2)$ time, or the summary has to delete the cluster identifier c, which takes at most $T(w/2)$ time, but the deletion of i in cluster C_c took constant time, because C_c only contained the element i (i.e., the first case $|S_c| \leq 2$ of the deletion).

Algorithm 17: Remove procedure for $|S| > 2$ in $\mathcal{O}(\log\log|U|$ time

```
delete_{|S|>2}(T, q ∈ S, w):
  c, i = split(q, w/2)

  # min S \ {q} ≠ min S
  if q = T.min:
    c = T.summary.min
    i = T.clusters[c].min
    q = T.min = concat(c, i)

  # max S \ {q} ≠ max S
  if q = T.max:
    c = T.summary.max
    i = T.clusters[c].max
    q = T.max = concat(c, i)

  cluster = T.clusters[c]
  cluster.delete(i, w/2)

  # cluster became empty
  if cluster.min = ⊥:
    T.clusters[c] = ⊥
    T.summary.delete(c, w/2)

  # summary became empty
  if T.summary.min = ⊥:
    T.summary = ⊥
```

3.4. Linear Space

First observe, that if the clusters would be stored within an array instead of a hash table, that the space usage would have the following recursion

$$S(|U|) = (\sqrt{|U|} + 1)S(\sqrt{U}) + \mathcal{O}(1) \quad ,$$

where $\sqrt{|U|}$ clusters and the summary store elements in the reduced universe $\sqrt{|U|}$. The recursion has the closed form $S(|U|) = |U|$. That is not surprising, since already three elements would initialize an array for the clusters with size $\sqrt{|U|}$.

Mehlhorn and Näher [MN90] first showed that the actual space is $\mathcal{O}(n \log\log|U|)$ and used a standard ***indirection*** trick to reduce the space to $\mathcal{O}(n)$.

Lemma 3.1 *The space usage is* $\mathcal{O}(n \log\log |U|)$.

Proof. Let $x \in S$ be arbitrary, but fixed. At each level, the element stores the first $w/2$ bits recursively in the summary and the remaining $w/2$ bits in its cluster, where w is the current word size of the level. Furthermore on each level it uses $\mathcal{O}(1)$ space, either for the hash entry $T[c]$ for its cluster c, or its value in the min or max attributes. Thus, $S(w) = 2S(w/2) + \mathcal{O}(1) = \mathcal{O}(\log |U|)$ space per element.

But remark, that this approach heavily overcharges, because when a lot of elements share the same cluster and therefore pay for the summary, in reality only one of the elements needs to pay for it. Since, $\min S_c$ and $\max S_c$ are not stored recursively in the cluster C_c, we will charge the recursive cost for the summary entry only for those elements. Therefore, all other elements in S_c pay for the cluster C_c recursively. Thus, each element stores $w/2$ bits recursively in either the summary or the cluster, resulting in the following space recursion for each element $x \in S$

$$S(w) = S(w/2) + \mathcal{O}(1),$$

which has the closed form $S(w) = \mathcal{O}(\log w) = \mathcal{O}(\log\log |U|)$. Hence, the whole space requirement is $\mathcal{O}(n \log\log |U|)$. □

We will see in Section 5 a technique called indirection, which also works for the van-Emde-Boas tree. In fact, the X-fast trie has stronger requirements for the indirection as the van-Emde-Boas tree. For example the X-fast trie has a space usage of $\mathcal{O}(n \log |U|)$ and an update time of $\Theta(\log |U|)$. The update time is also the reason, why the resulting Y-fast trie has only amortized $\mathcal{O}(\log\log |U|)$ time per update operation. Whereas, the van-Emde-Boas tree retains the worst-case $\mathcal{O}(\log\log |U|)$ update time.

Lemma 3.2 *The space can be reduced to* $\mathcal{O}(n)$ *by using a technique called indirection.*

Proof. See the proof in Section 5 or the one of Mehlhorn and Näher [MN90]. □

Another technique to show linear space is by Pătraşcu, which can be found in Appendix A.1.2. The main idea is that multiple items can be stored in one word. An element on the first level needs $(1/2)w$ bits, on the second level $(1/4)w$ bits and so forth. That means that each level gets geometrical cheaper and that it takes $\mathcal{O}(w)$ bits to store one element. Thus, $\mathcal{O}(nw)$ bits for all elements, or stated differently $\mathcal{O}(n)$ words. But not only the data has to be stored, but also the pointers. Since the pointer need to address smaller universes u, the pointer size $\log u$ gets geometrical cheaper. So theoretically speaking, it is possible to store a whole van-Emde-Boas tree within a contiguous bit-array, which consist of $\mathcal{O}(n)$ words.

But practically speaking, it seems to be hard to implement it in that way. Because on each level the number of elements in the clusters can greatly differ, there seems to be no easy way to subdivide the words, s.t. the van-Emde-Boas nodes reside within the word and can be easily accessed or updated. Therefore, we will prefer the indirection technique used by Mehlhorn and Näher and review the use cases of this compression idea in the later Sections 7 and 8.

3.5. Result and Remarks

Theorem 3.1 *The van-Emde-Boas tree solves the dynamic predecessor problem with linear space and takes* $\mathcal{O}(\log\log |U|)$ *time for each operation.*

Proof. Algorithms 11 to 15 and Lemma 3.2. □

Remarks

Even though this tree is named after Peter van Emde Boas, Donald Knuth was the first, who fully formalized the algorithms and proved[1] them correct [Boa13]. In 2009, a new chapter in "Introductions to algorithms" [Cor+09] for the van-Emde-Boas-Tree was added. The solution itself is presented in a step-by-step way with several intermediate solutions and leaves the linear space analysis as an exercise.

[1] The famous quote "Beware of bugs in the above code; I have only proved it correct, not tried it." by D. Knuth came from this work.

4. X-fast Trie

The X-fast trie, introduced by Willard [Wil83], is a different approach to the static predecessor problem and similar to van-Emde-Boas trees in that it supports all static operations in $\mathcal{O}(\log\log|U|)$ time, by using $\mathcal{O}(n\log|U|)$ space. Even if the space usage is not optimal, it is remarkable, that this data structure is the scaffold of a family of different solutions, e.g. Y-fast, Z-fast, μ-fast and Δ-fast tries, which will be presented later in this thesis.

4.1. Data Structure

A X-fast trie consists of a non-compact binary trie $G = (V, E)$ and a hash table T. All nodes $u \in V$ will be stored in the hash table T by using the key of the node, which is the bit-sequence of the root-to-node path, in short $T[u.key] = u$.

4.2. Finding the Lowest Common Ancestor

Finding the lowest common ancestor of an element q is quite easy. Remember the definition of the lowest common ancestor

$$\mathrm{lca}(q) = \max\{v \in V \mid \mathrm{lcp}(q, v) = v\} \ .$$

Therefore, the lowest common ancestor is the longest prefix of q which has a corresponding node in the trie. Since the hash table maps all keys to their corresponding nodes, we can check whether a node with a prefix of q is present in the trie by looking it up in the hash table. Thus, a binary search on the prefixes will find the lowest common ancestor and takes $\mathcal{O}(\log w) = \mathcal{O}(\log\log|U|)$ time. The special case $\mathrm{lca}(q) = q \in S$ can be handled by looking at $T[q]$ before the binary search.

Algorithm 18: The lowest common ancestor search in $\mathcal{O}(\log\log|U|)$ time

```
lca(T, q ∈ U \ S, lcp = ε, w):
  if w ≤ 1:
    return T[lcp]

  c, i = split(q, w/2)
  k = concat(lcp, c)

  if T[k] ≠ ⊥:
    return lca(T, i, k, w/2)
  else:
    return lca(T, c, lcp, w/2)
```

4.3. Static and Dynamic Predecessor Problem

Theorem 4.1 *The X-fast trie solves the static predecessor problem, where the operations take $\mathcal{O}(\log\log|U|)$ time and the structure uses $\mathcal{O}(n\log|U|)$ space. The structure can be build in $\mathcal{O}(n\log|U|)$ time. Furthermore, it allows update operations in $\Theta(\log|U|)$ time.*

Proof. We have seen in Theorem 2.1 that all static predecessor operations only depend in their runtime on finding the lowest common ancestor. Furthermore, we have seen that the non-compact trie can be built in $\mathcal{O}(n\log|U|)$ time, by inserting the elements according to Algorithm 4. The space usage is $\mathcal{O}(n\log|U|)$, since both the non-compact trie and the hash map store at most $\mathcal{O}(n\log|U|)$ prefixes. The update time of $\Theta(\log|U|)$ follows from the non-compact trie. $\square$

5. Y-fast Trie

The Y-fast trie, introduced by Willard [Wil83], is a different approach for the static predecessor problem and it supports all static operations in $\mathcal{O}(\log\log|U|)$ time, by only using linear space and $\mathcal{O}(n\log\log|U|)$ preprocessing time. The dynamic version has $\mathcal{O}(\log\log|U|)$ amortized time, while maintaining the linear space. The general technique used in Y-fast tries is called ***indirection***.

5.1. Data Structure

The Y-fast trie improves the space requirement of the X-fast trie by using indirection and uses a top and a bottom structure to achieve this. The bottom structure partitions the set S in contiguous[1] buckets $B_1, \ldots, B_m \subseteq S$ with $m = \frac{n}{\log|U|}$. Each bucket B_i for $i = 1 \ldots m$ has the size $\mathcal{O}(\log|U|)$ and is managed by a balanced binary tree (e.g., AVL-tree, red- black-tree). Furthermore, each bucket B_i chooses one ***representing element*** and promotes it into the top structure which is managed by a X-fast trie.

Lemma 5.1 *The data structure uses $\mathcal{O}(n)$ space.*

Proof. The top structure, i.e. the X-fast trie, manages $\mathcal{O}(\frac{n}{\log|U|})$ items, which by Theorem 4.1 takes $\mathcal{O}(\frac{n}{\log|U|} \cdot \log|U|) = \mathcal{O}(n)$ space. The bottom structures needs $\mathcal{O}(\log|U|)$ space for each bucket. Hence, all binary search trees together take linear space. □

5.2. Static Predecessor Problem

To answer the predecessor queries, one has to resolve the indirection, i.e. find the bucket containing q. Let $U_1 = (s_0, s_1], \ldots, (s_{i-1}, s_i], \ldots, (s_{m-1}, s_m] = U_m$ be a partition of U with the splitter elements $s_0 = -1$, $s_i = \max B_i$ (for $1 \leq i < m$) and $s_m = |U| - 1$. Note that this essentially extends the range of a bucket to the left, except for the last bucket where it also extends to the right (see Figure 3). Furthermore, let $r_1 \in B_1, \ldots, r_m \in B_m$ be representing elements of each bucket.

5.2.1. Searching for the Correct Bucket

Finding the bucket B_i of q is the same as finding the set U_i, which contains q. If q is smaller than r_1 the set is U_1. Otherwise the set can be found by a predecessor search on the representing elements. The search delivers an element r_i with $r_i \leq q < r_{i+1}$. Thus, q is either in the set U_i or U_{i+1}. This can be determined in constant time by looking at the splitter $s_i = \max U_i$. The bucket search takes $\mathcal{O}(\log\log|U|)$ time, because there is only one predecessor query in the X-fast trie.

[1] All elements in the partition are naturally ordered, that means for $i < j$, $x \in B_i$ and $y \in B_j$ we have $x \leq y$.

Algorithm 19: Find the bucket in $\mathcal{O}(\log\log|U|)$ time

```
search_tree(X, q ∈ U):
  # T_i with max T_i ≤ q < max T_{i+1}
  T = X.predecessor_or_equal(q)

  # q is smaller than r_1
  if T = ⊥:
    T = X.min_tree

  # the last bucket covers all
  if T.next_tree = ⊥:
    return T

  if T.splitter < q:
    return T.next_tree

  return T
```

5.2.2. Searching for an Element

After resolving the indirection, it is easy to find an element within the Y-fast trie. First find the correct bucket and then search in the bucket for the element. Both searches take $\mathcal{O}(\log\log|U|)$ time[1].

Algorithm 20: Search an element in $\mathcal{O}(\log\log|U|)$ time

```
search(X, q ∈ U):
  T = X.search_tree(q)
  return T.search(q)
```

5.2.3. Finding the Predecessor and Successor

Before finding the predecessor of q, find the bucket which contains q. Assume that $q \in U_i$. The predecessor of q is either in B_i or in B_{i-1}, because the interval $(\max U_{i-1}, \min B_i) \subsetneq U_i$ contains no elements of S. Thus, the predecessor is either $\perp$ (for $q < \min B_1$), $\max B_{i-1}$ (for $q < \min B_i$) or the predecessor in the set B_i.

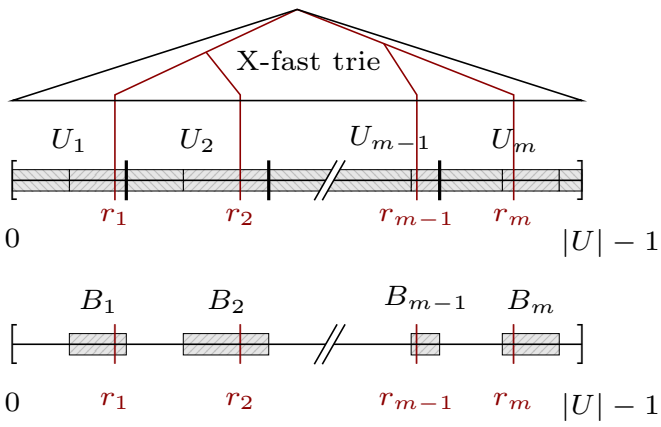


Figure 3: Y-fast Trie - At the bottom is an illustration of the buckets $B_1, \ldots, B_m$. The top depicts the partition $U_1, \ldots, U_m$ of U and the representing elements $r_1, \ldots, r_m$ which were promoted into the top structure.

[1] The balanced binary tree manages $\mathcal{O}(\log|U|)$ items. Hence, all operations have a runtime of $\mathcal{O}(\log\log|U|)$.

Algorithm 21: Find the predecessor in $\mathcal{O}(\log\log|U|)$ time

```
predecessor(X, q ∈ U):
  # return T_i with max T_{i-1} < q ≤ max T_i
  T_i = X.search_tree(q)
  p = T_i.predecessor(q)

  if p ≠ ⊥:
    return p

  T_{i-1} = T_i.prev_tree
  if T_{i-1} = ⊥
    return ⊥

  return T_{i-1}.max
```

And for the sake of completeness, the successor search.

Algorithm 22: Find the successor in $\mathcal{O}(\log\log|U|)$ time

```
successor(X, q ∈ U):
  # return T_i with max T_{i-1} < q ≤ max T_i
  T_i = X.search_tree(q)
  s = T_i.successor(q)

  if s ≠ ⊥:
    return s

  T_{i+1} = T_i.next_tree
  if T_{i+1} = ⊥
    return ⊥

  return T_{i+1}.min
```

5.2.4. Result

Theorem 5.1 *The static Y-fast trie solves the static predecessor problem in $\mathcal{O}(\log\log|U|)$ per operation and linear space. The preprocessing time is linear on sorted input.*

Proof. Starting from a sorted input, subdividing the n elements in $\frac{n}{\log|U|}$ buckets of size $\log|U|$ and constructing the binary search tree for each bucket is possible in linear time (repeatedly take the median as root). The construction time of the X-fast trie is $\mathcal{O}(n'\log|U|)$ (Theorem 4.1). Thus, the X-fast trie can be build in linear time, because there are $n' = \frac{n}{\log|U|}$ representing elements. □

5.3. Dynamic Predecessor Problem

If $\mathcal{O}(\log|U|)$ elements per bucket can be maintained, then all operations will have the same time bound. To ensure this, a bucket *underflows*, if the bucket has size $\frac{1}{2}\log|U|$ or smaller and is *full*, if the size is $2\log|U|$ or larger.

Additionally, let the notation of the partition $U_1,\ldots,U_m$ correspond dynamically to the current state of the data structure. There is one remaining problem with the definition. We assumed, that r_i is in B_i and therefore in U_i, but all elements greater or equal to r_i could be deleted from B_i, s.t. r_i is not in U_i anymore. That would break the notation, that r_i is the representing element of U_i. Therefore we redefine the splitter element s_i as $s_i = \max(B_i \cup \{r_i\})$ for $1 \le i < m$.

5.3.1. Inserting an Element

Insertion is straightforward. First find the set $q \in U_i$ and then insert q into the tree, which manages B_i. If the bucket B_i is full, split it in two buckets with nearly equal size, delete r_i

from the X-fast trie and insert two new representing elements for the two buckets into the X-fast trie.

Algorithm 23: Insert an element in $\mathcal{O}(\log\log|U|)$ amortized time

```
1  insert(X, q ∈ U):
2    if X is empty:
3      T = create new tree
4      T.insert(q)
5      X.insert(T)
6      return
7
8    T = X.search_tree(q)
9    T.insert(q)
10
11   if T is full:
12     X.remove(T)
13
14     T1,T2 = T.split_at_median()
15
16     T1.splitter = T1.median()
17     T2.splitter = T2.median()
18
19     X.insert(T1)
20     X.insert(T2)
```

The operations in the binary search tree and the search for the bucket takes $\mathcal{O}(\log\log|U|)$ time. Thus, the insert operation of the Y-fast trie takes $\mathcal{O}(\log\log|U|)$ time, if no split occurs. The update operations for the X-fast trie take $\Theta(\log|U|)$ time. We will later see, that the time can be amortized to $\mathcal{O}(\log\log|U|)$ per operation.

There remains one problem, when inserting the first element q (see Algorithm 23, Line 2ff). In this case, a new representing element r_1 will be promoted into the X-fast trie which takes $\Omega(\log|U|)$ time. When alternatively deleting and inserting q, the amortized time would be $\Omega(\log|U|)$ and not the targeted $\mathcal{O}(\log\log|U|)$. Thus, we have to defer the insertion of the representing element r_1 after the first $\Omega(\log|U|)$ insertions when the second representing element r_2 gets promoted into the top structure. In the meantime before inserting r_2, answer the queries (i.e. search, predecessor and successor) according to the separately stored r_1.

5.3.2. Deleting an Element

Like insert, first search the set $U_i \ni q$ and delete q from B_i. Afterwards, if the bucket underflows, merge the bucket B_i with one of the neighbor buckets. Should the merged bucket be full, split it again.

Algorithm 24: Remove an element in $\mathcal{O}(\log\log|U|)$ amortized time

```
1  remove(X, q ∈ S):
2    T1 = X.search_tree(q)
3    T1.remove(q)
4
5    # the last bucket becomes empty
6    if T1 is empty:
7      X.remove(T1)
8      return
9
10   # the last bucket can't be merged
11   if T1 underflows and m ≥ 2:
12     T2 = T1.next_tree or T1.prev_tree
13     X.remove(T1)
14     X.remove(T2)
15     T = T1.merge(T2)
16
17     if T is full:
18       T1,T2 = T.split_at_median()
19
20       T1.splitter = T1.median()
21       T2.splitter = T2.median()
22
23       X.insert(T1)
24       X.insert(T2)
25     else:
26       T.splitter = T.median()
27
28       X.insert(T)
```

We have to be careful about the deferred insert. Thus, if the X-fast trie has only two elements and one of them will be deleted, delete both elements and store the other element separately.

This ensures that when deleting the last element the remove operation will only need $\mathcal{O}(1)$ time.

5.3.3. Amortized Analysis

Lemma 5.2 *The operations remove and insert have an amortized runtime of* $\mathcal{O}(\log\log|U|)$.

Proof. If a bucket does not underflow and is also not full during an update operation, the runtime is clearly $\mathcal{O}(\log\log|U|)$. There are three events where $\mathcal{O}(\log|U|)$ time is needed for an update (see Figure 4). The first one is during an insert operation where a bucket overflows, the second one is during a remove operation where a bucket underflows, but the resulting merged bucket is not full and the last one where the merged bucket is full and needs a split. We will show that the update cost at those events can be paid by the accumulated payment.

Let c be the maximal constant of all possible $\mathcal{O}(\log|U|)$ time operations (e.g., for the remove and insert operations of the X-fast trie) in the insert and delete operations of the Y-fast trie and let $w = \log|U|$. Define the fund $\phi(B)$ of a bucket B as

$$\phi(B) = \begin{cases} c(|B| - w) & \text{if } |B| \geq w \\ 5c(w - |B|) & \text{if } |B| < w \end{cases} .$$

This fund function adds only constant overhead per update operation, because when inserting an element one has to pay $1c$ and when deleting one has to pay $5c$. The fund function $\phi(B)$ indicates how much accumulated payment a bucket B of the current size should have.

The main idea in the following amortized analysis is that at those events the accumulated $fund$ covers the *costs* of the event, i.e. $costs \leq fund$. As *costs* we mean the costs of the event and the expected accumulated payment of the bucket(s) after that event, since a new bucket must have the appropriate reserves for its size, otherwise it may not be able to pay future events. Whereas the $fund$ is the accumulated money to pay the costs.

For example in the first case of a full bucket B_s, the costs are cw, which reflects the runtime of the insertion operation for $w = \log|U|$, and $\phi(B_1)$ and $\phi(B_2)$, which is the expected accumulated payment of the buckets B_1 and B_2 after the insertion. In this case is $\phi(B_1) = \phi(B_2) = 0$, because the size of B_1 and B_2 after the split is w. The $fund$ of saved money is $\phi(B_s) = c(2w - w)$, because there were at least w insertions before the split.

Inserting an Element:

Let B_s be the bucket after the insert operation, but before the resulting buckets of the split B_1, B_2. The size of the buckets are $|B_s| = 2w$ and $|B_1| = |B_2| = w$. See Figure 4a for an illustration.

$$costs \leq cw + \phi(B_1) + \phi(B_2) = cw + 2c(w - w) \tag{1}$$

$$fund = \phi(B_s) = c(2w - w) \tag{2}$$

$$costs - fund \leq cw - cw \leq 0 \tag{3}$$

Removing an Element:

Let in the rest of the proof B_1 be the bucket, which contained q, B_2 be the neighbor bucket and $B_m = B_1 \cup B_2$ be the merged bucket. The size of B_1 is $\frac{1}{2}w$, because it caused the underflow event and the size of B_m is $|B_m| = |B_1| + |B_2| = \frac{1}{2}w + |B_2|$.

Merge the Buckets, but no split: $(\frac{1}{2}w \leq |B_2| < \frac{3}{2}w)$

In this case the size of B_m is in the range $w \leq |B_m| < 2w$. Thus, the fund $\phi(B_m)$ and $\phi(B_1)$ remains the same during this case. Therefore, we have to proof in two cases the inequality $costs \leq fund$, depending on the available fund of B_2. Figure 4b depicts this case.

Before showing the two cases, let us calculate the funds:

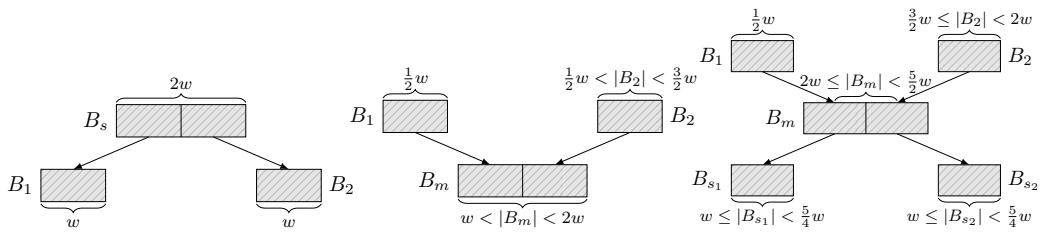
$$\phi(B_1) = 5c(w - \tfrac{1}{2}w) = \tfrac{5}{2}cw \qquad \text{(since } |B_1| < w) \tag{4}$$
$$\phi(B_2) = 5c(w - |B_2|) = 5cw - 5c|B_2| \qquad \text{(if } |B_2| < w) \tag{5}$$
$$\phi(B_2) = c(|B_2| - w) = c|B_2| - cw \qquad \text{(if } |B_2| \geq w) \tag{6}$$
$$\phi(B_m) = c(|B_m| - w) = c(\tfrac{1}{2}w + |B_2| - w) = c|B_2| - \tfrac{1}{2}cw \qquad \text{(since } |B_m| \geq w) \tag{7}$$

Case 1: $(\frac{1}{2}w < |B_2| < w)$

$$costs - fund \leq cw + \phi(B_m) - (\phi(B_1) + \phi(B_2)) \tag{8}$$
$$= cw + c|B_2| - \tfrac{1}{2}cw - \tfrac{5}{2}cw - 5cw + 5c|B_2| \qquad (|B_2| \leq w) \tag{9}$$
$$\leq cw + cw - 8cw + 5cw \leq 0 \tag{10}$$

Case 2: $(w \leq |B_2| < \frac{3}{2}w)$

$$costs - fund \leq cw + \phi(B_m) - (\phi(B_1) + \phi(B_2)) \tag{11}$$
$$= cw + c|B_2| - \tfrac{1}{2}cw - \tfrac{5}{2}cw - c|B_2| + cw \leq 0 \tag{12}$$



(a) After inserting an element, the bucket B_s is full and needs to be split.

(b) After removing an element, the bucket B_1 underflows and needs to be merged with a neighbor bucket B_2.

(c) The merge produces a full bucket, which again needs to be split into two buckets.

Figure 4: Y-fast Trie - Events which need to be considered in the amortized analysis, because they take $\mathcal{O}(\log |U|)$ time.

Merge the Buckets with a Split afterwards: $(\frac{3}{2}w \leq |B_2| < 2w)$

An illustration of this case can be found in Figure 4c. In this case we have to split B_m again into two buckets B_{s_1} and B_{s_2}. Clearly, the size of B_{s_1} and B_{s_2} is $|B_{s_1}| = |B_{s_2}| = \frac{1}{2}|B_m|$ and is in the range $w \leq |B_{s_1}| < \frac{5}{4}w$, because the size of B_m is in the range $2w \leq |B_m| < \frac{5}{2}w$.

Let us calculate the funds like before:

$$\phi(B_1) = 5c(w - \tfrac{1}{2}w) = \tfrac{5}{2}cw \qquad \text{(since } |B_1| < w) \tag{13}$$
$$\phi(B_2) = c(|B_2| - w) = c|B_2| - cw \qquad \text{(since } |B_2| \geq w) \tag{14}$$
$$\phi(B_{s_1}) = \phi(B_{s_2}) = c(|B_{s_1}| - w) \qquad \text{(since } |B_{s_1}| \geq w) \tag{15}$$
$$= c\left(\tfrac{1}{2}\left(\tfrac{1}{2}w + |B_2|\right) - w\right) = \tfrac{1}{2}c|B_2| - \tfrac{3}{4}cw \tag{16}$$

In this case are the costs, cw for the merge and split operations and the expected reserves of the buckets B_{s_1} and B_{s_2}.

$$costs - fund \leq cw + \phi(B_{s_1}) + \phi(B_{s_2}) - (\phi(B_1) + \phi(B_2)) \tag{17}$$
$$= cw + 2\left(\tfrac{1}{2}c|B_2| - \tfrac{3}{4}cw\right) - \tfrac{5}{2}cw - c|B_2| + cw \tag{18}$$
$$= cw + c|B_2| - \tfrac{3}{2}cw - \tfrac{5}{2}cw - c|B_2| + cw = -2cw \leq 0 \tag{19}$$
$$\tag{20}$$

□

5.3.4. Result

Theorem 5.2 *The dynamic Y-fast trie solves the dynamic predecessor problem in $\mathcal{O}(\log\log|U|)$ time per static operation and $\mathcal{O}(\log\log|U|)$ amortized time per dynamic operation. The data structure uses linear space.*

Proof. We have seen that all static operations require $\mathcal{O}(\log\log|U|)$ time, while the dynamic operations require $\mathcal{O}(\log\log|U|)$ amortized time (Algorithms 19 to 24 and Lemma 5.2). The needed space is linear (Lemma 5.1). □

6. Z-fast Trie

In this section, we will look at a space efficient solution for the static and dynamic predecessor problem, called Z-fast trie. The name was coined by [Bel+09] and was independently discovered by Ružić [Ruž09] and Belazzougui *et al* [Bel+09] in 2009. They both solved the static predecessor problem, whereas Belazzougui *et al* later gave in [BBV10] a solution for the dynamic predecessor problem. We will present a new technique for the dynamic predecessor problem.

6.1. Data Structure

The data structure uses a compact trie and a hash table, which builds an index into the compact trie and is similar to the X-fast trie. In fact we will – similar to X-fast tries – assign an edge $e = (u, v)$ a key, which we call an ***associated key*** denoted by α_e and store the upper endpoint u of the edge e in the hash table with the key α_e, in short $T[\alpha_e] = u$. Furthermore, store $T[v.key] = v$ for each leaf v in the trie and $T[\varepsilon] = root$.

The associated key α_e for an edge $e = (u, v)$ will be on the edge e, i.e $\alpha_e \in e$ (as denoted in Definition 2.21) and is unique among all associated keys (see Figure 5a for an example). We will later see which string on the edge should be chosen, s.t. the binary search will work.

Lemma 6.1 *The data structure uses linear space.*

Proof. We have seen in Theorem 2.2 that the compact trie needs linear space. The additional space by the hash map is also linear, because the number of edges and nodes in a compact trie is linear (Lemma 2.24). □

6.2. Finding the Lowest Common Ancestor

Finding the lowest common ancestor of a query q will be done in two phases. First find a ancestor node of q with a binary search on the edges. This node might not be the lowest common ancestor, so the second phase will find it.

Second Phase

Let us assume that the search in phase one returns a key from the hash map which has the longest common prefix with the query q, i.e.

$$search_I(q) = \max\{\alpha \in pre(q) \mid T[\alpha] \neq \bot\} \ ,$$

where $pre(q)$ is the set of all prefixes of q. Note that $search_I(q) = q$ iff $q \in S$.

We defined the lowest common ancestor of an element $q \in U$ as a node of the compact trie (Definition 2.22). Since the hash map stores all nodes, the goal is to find the key p of the lowest common ancestor in the hash map, i.e. $T[p] = \text{lca}(q)$. The next lemma shows that after finding $\alpha = search_I(q)$, we almost found the lowest common ancestor of q, i.e. that either $p = \alpha$ and $T[\alpha] = \text{lca}(q)$ or $T[\alpha].child = \text{lca}(q)$.

Lemma 6.2 *Let $q \in U$ and $search_I(q) = \alpha$. The lowest common ancestor of q in the compact trie is either $T[\alpha]$ or one of the children of $T[\alpha]$.*

Proof. First let us deal with the special cases. If $q \in S$ then $search_I(q)$ will return q, because q is the key of the leaf $v = T[q]$ and by definition is v the lowest common ancestor of q. Now assume $\alpha = \varepsilon$, but then is the longest common prefix of q among the elements in S lexicographical smaller than any associated key α_e and in particular lexicographical smaller than the children of the root. Thus, $T[\varepsilon] = root$ is the lowest common ancestor.

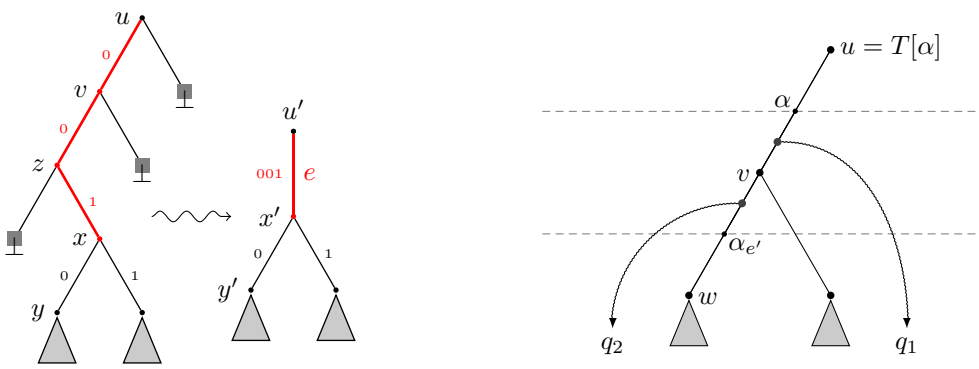
Now let us consider the case for $q \notin S$ and $\alpha \neq \varepsilon$, which is depicted in Figure 5b. In this case, α is an associated key of an edge $e = (u, v)$ with $\alpha \in e$. We know that u is at least an ancestor of q, because $u < \alpha \leq q$ and $search_I(q) = \alpha$ (the precondition of the lemma).

If $q < v$, then $u < \alpha \leq q < v$ and by definition u is the lowest common ancestor of q.

Thus, the only case left is $v < q$, since the case $\text{lcp}(q) = v$ can not happen according to Corollary 2.23 and the current assumption of $q \notin S$ and $\alpha \neq \varepsilon$. Let $e' = (v, w)$ be the edge in direction of q. Since α is the longest common prefix of q among the associated keys, q is lexicographical smaller than $\alpha_{e'}$. Thus, $u < \alpha \leq v < q < \alpha_{e'} \leq w$ and by definition v is the lowest common ancestor of q.

We have shown that in each case either $T[\alpha]$ or one of the children of $T[\alpha]$ is the lowest common ancestor of q. □

Thus, the lowest common ancestor can be found in constant time when given the result of the first phase and the runtime is only dependent on the time needed for the first phase, which we will see is $\mathcal{O}(\log \log |U|)$.



(a) The compact edge e consist of the nodes u, v and z from the non-compact trie representation and the associated key α_e could be any string on the edge e, i.e. $\alpha_e \in \{0, 00, 001\}$.

(b) $search_I$ will return for both queries q_1 and q_2 the associated key α, whereas the lowest common ancestor of q_1 and q_2 is respectively u and v.

Figure 5: Z-fast Trie - Choices for an associated key of a edge in a compact trie and an illustration of the second phase of the lowest common ancestor search.

Algorithm 25: The lowest common ancestor search in $\mathcal{O}(search_I)$ time

```
1  lca(T, q):
2    # special case q ∈ S
3    if T[q] ≠ ⊥:
4      return T[q]
5
6    α = search_I(q)
7    u = T[α]
8
9    # special case α = ε
10   if α = ε:
11     return u
12
13   let (u, v) be the edge of α
14
15   # lcp(q) is on the edge
16   if α ≤ q ≤ v:
17     return u
18
19   return v
```

The definition of an associated key and how to compute it.

Before we present the short definition of the associated key α_e, which was given by Belazzougui *et al* [Bel+09], we want to show that the short definition comes from a binary search. This will help us later to understand why the definition of the associated key works in the binary search of $search_I(q)$.

We know that $\alpha_e \in e$ must be on the edge $e = (u, v)$. Let v^* be $v_0 \dots v_{|v|-1} 0^{w-|v|}$ and let p be the current prefix in a binary search on all prefixes of v^*. The binary search works as follows. If $p \leq u$ (in lexicographical order) proceed with a longer prefix, if $p > v$ proceed with a shorter prefix and if $p \in e = (u, v)$ stop the search. This binary search takes $\mathcal{O}(\log w) = \mathcal{O}(\log \log |U|)$ time, because there are w prefixes of v^*.

Definition 6.3 *Define the* ***associated key*** α_e *as the prefix* p*, where the above described binary search stops.*

Remark 6.4 *Note that the padding of* v^* *could have been any bit-string of length* $w - |v|$ *with the same resulting* α_e*.*

Since all p in the binary search are prefixes of v^* (even u and v), the binary search can be characterized by the length of u, v and p. The search stops, if $|u| < |p| \leq |v|$. If $|u| < |p| \leq |v|$, then $v_0 \dots v_{|p|-1} \in e$ is the associated key α_e with $|\alpha_e| = |p|$. The following algorithm describes the binary search to find the length of α_e.

Algorithm 26: Finding the length of the associated key in $\mathcal{O}(\log \log |U|)$ time

```
1  len_α(l = |u|, r = |v|, w):
2    m = w/2
3
4    if l < m ≤ r:
5      return m
6
7    if l < r < m:
8      return 0 · w/2 + len_α(l, r, m)
9
10   if m ≤ l < r:
11     l = l − m
12     r = r − m
13     return 1 · w/2 + len_α(l, r, m)
```

We will now show how to transform Algorithm 26 into the one given by Belazzougui *et al*.

Lemma 6.5 *Let* $m_0 \dots m_{\log w - 1}$ *be the binary expansion of the length of* p*, i.e.* $|p| = \sum_{i=0}^{\log w - 1} m_{\log w - i - 1} 2^i$*, and let* k *be the index where* m_k *is the last bit with value* one *in the given sequence, i.e.* $m_k \dots m_{\log w - 1} = 10^{\log w - k - 1}$*.*

The value of m_i for $0 \leq i < k$ corresponds to the $(i+1)$-th choice in the binary search of Algorithm 26. That means that in the $(i+1)$-th iteration, the search recurses into the case $l < r < m$ when $m_i = 0$ and into the case $m \leq l < r$ when $m_i = 1$. The search stops in the $(k+1)$-th iteration.

Proof. Let $l_0 \dots l_{\log w - 1}$ and $r_0 \dots r_{\log w - 1}$ be the binary expansion of the lengths $|u|$ and $|v|$. Note that m is $w/2$ in the first iteration, $w/4$ in the second and generally, $w/2^k$ in the k-th iteration and be aware that in each iteration one bit of the bit-sequence $m_0 \dots m_{\log w - 1}$ will be set.

In the first iteration where $m = w/2$, the cases $l < r < w/2$, $w/2 < l < r$ and $l < w/2 < r$ correspond to the values of the most significant bits l_0 and r_0 and in this cases are the values respectively $l_0 = r_0 = 0$, $l_0 = r_0 = 1$ and $0 = l_0 < r_0 = 1$. In the case $l_0 = r_0 = 0$, the bit m_0 will be set to *zero* by adding $0 \cdot w/2$, in the case $l_0 = r_0 = 1$ the bit m_0 will be set to *one* by adding $1 \cdot w/2$ and in the case $0 = l_0 < r_0 = 1$ the bit m_0 will be set to *one*, because $m = w/2$ will be returned. Therefore in the cases $l < r < m$ and $m \leq l < r$ holds $l_0 = r_0 = m_0$ and in the case $l < m \leq r$ holds $0 = l_0 < m_0 = r_0 = 1$.

Now notice that before recursing in either of the case, the bit-sequences will be normalized to $l_0 l_1 \dots l_{\log w - 2} := l_1 \dots l_{\log w - 1}$, $r_0 \dots r_{\log w - 2} := r_1 \dots r_{\log w - 1}$ and $m_0 \dots m_{\log w - 2} := m_1 \dots m_{\log w - 1}$ and further $w := w/2$. Therefore, the same argumentation can be repeated until the case $l < m \leq r$ happens.

Thus, $m_0 \dots m_k$ corresponds to the choices of the binary search and the values of $m_{k+1}, \dots, m_{\log w - 1}$ are *zero*, because no more terms of the sum follows after the $(k+1)$-th iteration. We further showed, that $l_0 \dots l_{k-1} = r_0 \dots r_{k-1} = m_0 \dots m_{k-1}$ are the same prefixes. □

Corollary 6.6 *The binary search in Algorithm 26 finds the first position k where the bit-sequences $l_0 \dots l_{\log w - 1}$ and $r_0 \dots r_{\log w - 1}$ differ, i.e. $l_i = r_i$ for $0 \leq i < k$ and $l_k \neq r_k$, and returns the length of α_e as binary expansion $r_0 \dots r_k 0^{\log w - k - 1}$.*

Proof. We showed that $l_0 \dots l_{k-1} = r_0 \dots r_{k-1} = m_0 \dots m_{k-1}$, that $0 = l_k \leq m_k = r_k = 1$ and that $m_{k+1} \dots m_{\log w - 1} = 0^{\log w - k - 1}$. Thus, $|\alpha_e| = r_0 \dots r_k 0^{\log w - k - 1}$. □

The first differing bit $l_k \neq r_k$ can be computed in constant time by finding the index of the most significant set bit of the binary expansion of $l \oplus r$ (Lemma 2.5). Thus, m can be constructed by zeroing the last $\log w - k - 1$ bits of r.

Algorithm 27: Finding the length of the associated key in $\mathcal{O}(1)$ time

```
1 lenα(l = |u|, r = |v|):
2   d = l ⊕ r
3   k = msb(d) − 1
4   return (r ≫ k) ≪ k
```

Thus, the associated key can be computed in $\mathcal{O}(1)$ time.

Algorithm 28: Finding the associated key in $\mathcal{O}(1)$ time

```
1 associated-key(u, v):
2   m = lenα(|u|, |v|)
3   v0 ... vm−1 = split(v, m)
4   return v0 ... vm−1
```

First Phase

We assumed this specification of the first search phase:

$$search_I(q) = \max\{\alpha \in pre(q) \mid T[\alpha] \neq \bot\} \ .$$

Remember that not all return values of $search_I(q)$ are associated keys, because we additionally defined $T[\varepsilon] = root$ and $T[s] = leaf\ of\ s$ for $s \in S$.

In the following, we will assume that $search_I(q)$ returns associated keys, because $search_I(q) = q$ can be checked by $T[q] \neq \bot$ before invoking $search_I(q)$ and $search_I(q) = \varepsilon$ is the base case when no associated key with the properties was found.

Lemma 6.7 *When $search_I(q)$ returns an associated key, it is the maximal associated key which is a prefix of q and is smaller than or equal to the key of the longest common prefix of q, i.e. $search_I(q) \leq \mathrm{lcp}(q)$. $\mathrm{lcp}(q)$ was defined in Definition 2.15.*

Proof. That $\alpha = search_I(q)$ is a prefix of q is clear, therefore let us show that it is not longer than $\mathrm{lcp}(q)$. Since $\alpha \in e = (u, v)$ for some edge e and $\alpha \in pre(v) \subseteq pre(q)$, α is one of the checked prefixes in the Definition 2.15 of $\mathrm{lcp}(q)$. □

We will argue that the following algorithm implements the specification of $search_I(q)$ for an element $q \in U \setminus S$.

Algorithm 29: $search_I$ in $\mathcal{O}(\log\log|U|)$ time.

```
1  searchI(T, q, lcp = ε, lca = root, w = |q|):
2    if w = 1:
3      return lca
4
5    c, i = split(q, w/2)
6    p = concat(lcp, c)
7
8    if T[p] ≠ ⊥:
9      return searchI(T, i, p, T[p], w/2)
10   else if p is on the edge of the lca
         and one of its children:
11     return searchI(T, i, p, lca, w/2)
12   else:
13     return searchI(T, c, lcp, lca, w/2)
```

Definition 6.8 *We say that an edge $e = (u, v)$ will be* ***accessed*** *by p, if $p \in e$ (see Definition 2.21)*

For now let us assume, that we do a binary search on the prefixes of q. Let p be the current prefix of q. If $p \in e$ for an edge e, proceed with a longer prefix, otherwise proceed with a shorter prefix. The search stops after $\log w$ iterations, since there are only w prefixes of q.

It is easy to see, that this binary search finds the longest common prefix of q. But note, that this search can collect all associated keys of the accessed edges and return the maximum associated key which is smaller than or equal to $\mathrm{lcp}(q)$. Thus, implementing $search_I(q)$ according to Lemma 6.7.

We will show that this binary search and the one in Algorithm 29 correspond to each other. For that, we will show that both searches enumerate the same associated keys.

The following lemma shows, that the first access p to an edge e in the above described binary search is the associated key α_e.

Lemma 6.9 *The first time an edge $e = (u, v)$ is accessed by a prefix p, the condition $T[p] \neq \bot$ holds. Assuming that $p \leq \mathrm{lca}(q)$.*

Proof. The first access of $p \in pre(q)$ to the edge e means that for the first time in the binary search $p \in e = (u, v)$ holds. Note that u, v and p are prefixes of q. Let $q_0 \dots q_{k-1}$ be the binary expansion of p with $k = |p|$. Since p is a prefix of v (by the Definition 2.21 of $p \in e$), the binary expansion of p is $q_0 \dots q_{k-1} = v_0 \dots v_{k-1}$.

First remember Definition 6.3, where we defined α_e as the first prefix in a binary search on all prefixes of $v^* = v_0 \dots v_{|v|-1} 0^{w-v}$, which fulfilled $\alpha_e \in e = (u, v)$. We observed that any bit-sequence can be used as padding for v^* (Remark 6.4). Thus, we chose $v^* = v_0 \dots v_{|v|-1} q_v \dots q_{w-1} = q_0 \dots q_{w-1} = q$. But that means that both binary searches of p and α_e proceeded the same steps. Thus, $p = \alpha_e$ and $T[p]$ is defined. □

The following lemma considers the remaining case of the above lemma, i.e. $\mathrm{lca}(q) < p$. This is a special case, because it describes the last part of the path of the query q which leaves the trie, for example when the query q_1 in Figure 5b diverges from the trie. It also reflects the reason, why the lowest common ancestor search is either in $T[\alpha_e]$ or in one of the children of $T[\alpha_e]$.

Lemma 6.10 *Assume* $\mathrm{lca}(q) < p \leq \mathrm{lcp}(q)$*, i.e. the last accessible edge* $e = (u, v)$*. Either the condition* $T[p] \neq \perp$ *holds for the first access to* e*, or it holds not, but then will the condition* $T[p'] \neq \perp$ *be false for all possible accesses* p' *to* e*. That means for* $p' \in e$ *and* $p' \in pre(q)$*, which is essentially the path between* $\mathrm{lca}(q)$ *and* $\mathrm{lcp}(q)$*.*

Proof. There are two cases, either $\alpha_e \in pre(\mathrm{lcp}(q))$ or $\alpha_e \notin pre(\mathrm{lcp}(q))$, because $\mathrm{lcp}(q) \in e$ and $\alpha_e \in e$ are prefixes of v.

First assume $\alpha_e \in pre(\mathrm{lcp}(q))$, but then $\alpha_e \in pre(\mathrm{lcp}(q)) \subseteq pre(q)$. This case is depicted in Figure 5b for the query q_1. Let $m = |\mathrm{lcp}(q)|$. Look at $v^* = v_0 \dots v_{m-1} v_m \dots v_{|v|-1} q_{|v|} \dots q_{w-1}$. Since $\alpha_e \in pre(\mathrm{lcp}(q))$, the prefixes $v_0 \dots v_{m-1} v_m, v_0 \dots v_m v_{m+1}, \dots, v_0 \dots v_{|v|-2} v_{|v|-1} \in e$ were not accessed first in the binary search. Thus, the part $v_m \dots v_{|v|-1}$ in v^* can be exchanged by $q_m \dots q_{|v|-1}$ without changing the result of α_e. But then $v^* = q_0 \dots q_{w-1} = q$, since $\mathrm{lcp}(q) = v_0 \dots v_{m-1} = q_0 \dots q_{m-1}$. Thus, both binary searches did the same steps again, resulting in $p = \alpha_e$.

Now assume $\alpha_e \notin pre(\mathrm{lcp}(q))$. But, then is $T[p']$ undefined for all $p' \in e$ with $p' \in pre(q)$, because there is only one associated key per edge and all p' are shorter than α_e. □

Before reading the next lemma, be aware that not every queried prefix k produces an access to an edge. For example, $k = q \notin S$ has no leaf in the trie and has therefore no edge it belongs to, which could produce an access.

Lemma 6.11 *Let* $e'_1, \dots, e'_m$ *be the access sequence of edges. An edge* e *appears at most in one contiguous subsequence* $e'_i, e'_{i+1}, \dots, e'_j$ *of that sequence, i.e.* $e = e'_l$ *for each* $l \in [i, j]$*.*

Proof. Let $p_1, \dots, p_m$ be the prefixes accessing the edges $e'_1, \dots, e'_m$. It is clear that each edge access will let the binary search proceed with a longer prefix. Thus, $p_1 < p_2 < \dots < p_m$.

Let us assume, that the edge $e = (u, v)$ has at least one access and let $e'_i, \dots, e'_j$ be the first accesses of e. The claim is, that e will not be accessed any time later, i.e. $e \neq e'_l$ for any $j < l \leq m$. Lets look at $e'_{j+1} = (u', v') \neq e$. It holds

$$u < p_j \leq v \leq u' < p_{k+1} \leq v' \ ,$$

because the nodes of different edges do not overlap. But then all prefixes of q have at least the prefix u', which means that $p_l \notin e$ for $j < l \leq m$. Thus, e will not be accessed later. □

Lemma 6.12 *The combination of the Algorithms 25 and 29, finds the lowest common ancestor of $q \in U$ in $\mathcal{O}(\log\log|U|)$ time.*

Proof. First let us proof the correctness of the first phase of the search.

Before the first access to the edge e'_1, the above described binary search might try prefixes p which are not on edges. In this case, the binary search will proceed with shorter prefixes. The algorithm does the same, because neither the condition $T[p] \neq \perp$ nor the condition, that p is on one of the edges from the root, will be true.

The first access to an edge e will be handled by $T[p] \neq \perp$, due to Lemma 6.9. The subsequent accesses are either to the same edge, to a non-existing edge or to a different edge.

- The access to the same edge is handled by remembering the last found edge e_{last} (this is done in the algorithm by storing lca). It suffices to store the last found edge, because the accesses to the same edge are contiguous (Lemma 6.11).
- The access to a non-existing edge shortens the prefix in the binary search. This also happens in the algorithm, because neither the condition $T[p] \neq \perp$ nor the condition $p \in e_{last}$ will be true.
- The last case, where a different edge will be accessed, is in fact an access to a new edge e' (Lemma 6.11). Thus, repeating the argumentation for $e := e'$.

A note on the exception when $\text{lca}(q) < p \leq \text{lcp}(q)$ (Lemma 6.10). This clearly happens only for the last accessed edge e_l in the accessed edges. We have seen, that either the first access of e_l is covered by $T[p] \neq \perp$ or not. If not, it means that the associated key α_{e_l} of e_l will not be enumerated by the binary search. Since the algorithm also does not enumerate it, they behave the same.

Thus, the the algorithm for the first phase works correct and takes $\mathcal{O}(\log\log|U|)$ time. We have seen in Lemma 6.2 that after the first phase, the lowest common ancestor can be found in constant time, completing the proof. □

6.3. Static Predecessor Problem

Theorem 6.1 *The Z-fast trie solves the static predecessor problem in $\mathcal{O}(\log\log|U|)$ time for each static operation, $\mathcal{O}(n)$ space and with a linear preprocessing time on sorted input.*

Proof. Using Theorem 2.2 and having a pointer from each subtree to its minimal and maximal leaf, the most claims follow directly. The hash table can be constructed in linear time by traversing the edges and computing the associated key for each edge in constant time (Algorithm 28). The data structure uses linear space, as seen in Lemma 6.1. □

Before explaining the dynamic predecessor problem, we will introduce tango trees. In Theorem 2.2 we have seen that the runtime of the operations is $\mathcal{O}(T_{\text{lca}}(q) + T_{\min}(q) + T_{\text{update-min}}(q))$, where $T_{\text{lca}}(q)$ is the time to find the lowest common ancestor of q, $T_{\min}(q)$ to find the minimal and maximal element in a subtree and $T_{\text{update-min}}(q)$ to update the minimal and maximal element. The current strategy of the static version to have a pointer to the minimal and maximal

leaves for each subtree is insufficient for $T_{\text{update-min}}(q)$, since each update operation can change $\Omega(\log |U|)$ pointers in the subtrees (e.g. when the global minimum changes).

The main observation is that all nodes which share the same minimal (or maximal) element form paths in the trie and that those paths decompose the trie (see Figure 6). The tango tree can update this path decomposition when the trie changes. Since the tango tree accomplishes that by managing height balanced binary search trees of at most $w = \log |U|$ elements, the time $T_{\text{update-min}}(q)$ to maintain the decomposition is $\mathcal{O}(\log \log |U|)$. Furthermore, the minimal (or maximal) element in a binary search tree can be found $\mathcal{O}(\log \log |U|)$ time.

6.4. Tango Trees

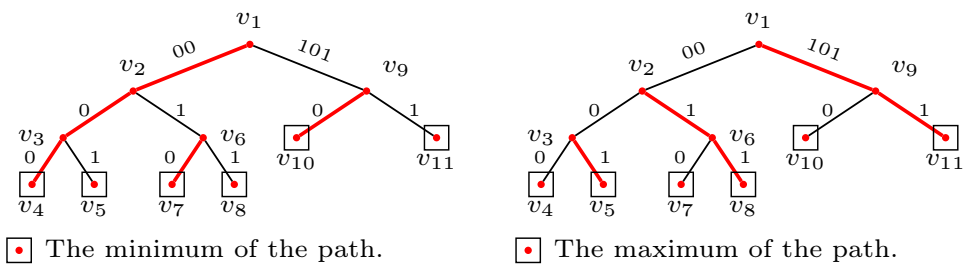
Tango trees were introduced by Demaine *et al* [Dem+07] to show that there exists a binary search tree with an $\mathcal{O}(\log \log n)$-competitive ratio. This result improved the previously know trivial bound of $\mathcal{O}(\log n)$. A data structure is c-competitive, if that data structure belongs to the family of binary search trees and executes any access sequence X in time $\leq c \cdot OPT(X)$, where $OPT(X)$ denotes the best possible runtime of any (offline) dynamic binary search tree, that can execute the sequence X by only using normal tree operations. Tango trees are similar to Sleator and Tarjan's link-cut trees [ST85], which allow efficient handling of subtrees. We will us tango trees to manage the path decomposition of the trie.

6.4.1. Data Structure

A Tango Tree manages the following model efficiently:

> Let P be the perfect binary tree of the keys $\{1, \ldots, n\}$. Each inner node v of P has a ***preferred child***, which is either the left or right child of v and which can be changed to its sibling.

Notice, that the preferred children form sub-paths of root-to-leaf-paths in P, which we call ***preferred paths***. When changing a preferred child, it can be viewed as a cut of the preferred path at some depth d into two paths p_1 and p_2 and joining the path p_1 with the path of the sibling, see Figure 7a.



(a) For example v_1, v_2, v_3 and v_4 share the same minimum v_4.

(b) For example v_1, v_9 and v_{11} share the same maximum v_{11}.

Figure 6: Z-fast Trie - The path decomposition of the trie.

A tango tree stores each preferred path in an ***auxiliary tree***, which is a height-balanced binary search tree (e.g., red-black tree or AVL tree), where the nodes of the preferred path are ordered by the key value. Each auxiliary tree stores at most $\mathcal{O}(\log n)$ nodes and therefore the height is at most $\mathcal{O}(\log \log n)$.

The Tango Tree given by Demaine *et al* [Dem+07] is a single binary search tree, where each auxiliary tree is a subtree of the tango tree. But, since this property is not needed in our case, we will not go into the details for that.

Furthermore [Dem+07] requires, that each node v in the auxiliary trees has the following additional attributes:

depth: The depth of the node v in the corresponding perfect binary search tree P, which remains unchanged during update operations.

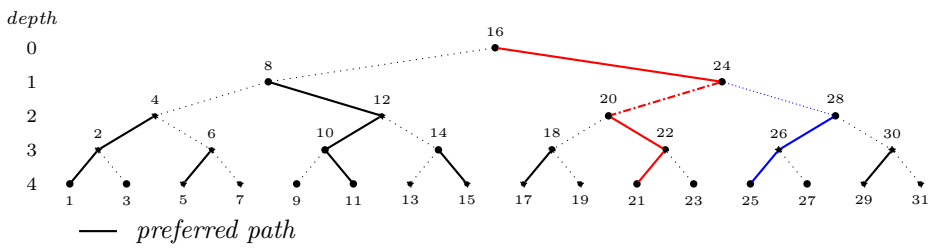
min-depth: The minimal *depth* in the perfect binary tree P of all nodes in the auxiliary subtree rooted at v, i.e. $\textit{min-depth} = \min\{u.\textit{depth} \mid u \text{ is a descendant of } v \text{ in the auxiliary tree}\}$.

max-depth: $\textit{max-depth} = \max\{u.\textit{depth} \mid u \text{ is a descendant of } v \text{ in the auxiliary tree}\}$.

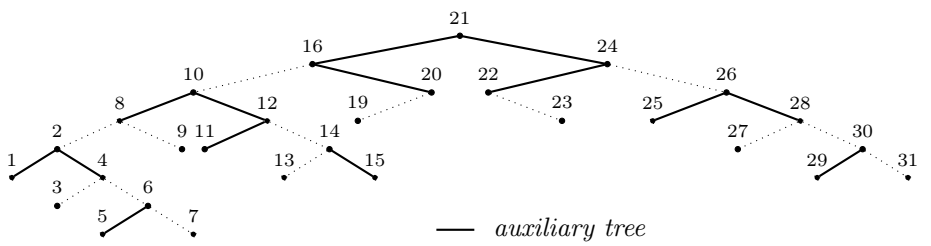
The augmented data *min-depth* and *max-depth* can be maintained during update operations with constant time overhead. Simply update the attributes after each rotation in the binary search tree locally. This is the same procedure as attributes like height, size or rank would be maintained.

6.4.2. Auxiliary Tree Operations

The auxiliary trees support the following operations:



(a) Changing the preferred child of 24 is the same as cutting the red path at depth 2 and joining the first halve with the blue path.



(b) The tango tree of the same preferred path decomposition, using a AVL tree as auxiliary tree.

Figure 7: Tango Trees - The reference tree P ($n = 31$) and a corresponding tango tree.

- $cut(T, d)$, cuts the preferred path represented by the auxiliary tree T at depth d into two trees, which represent the preferred paths with depth $< d$ and depth $\geq d$, and
- $join(T_1, T_2)$, joins the two preferred paths represented by the auxiliary trees T_1 and T_2 into one tree, representing the joined preferred path. The bottom node in the preferred path of T_1 must be the parent of the top node in the preferred path of T_2.
- $root(u)$, delivers the root of the auxiliary tree, where u belongs to.

The implementation of *cut* and *join* uses the attributes *depth*, *min-depth* and *max-depth*. Both operations can be implemented in $\mathcal{O}(h)$ time, where $h = \mathcal{O}(\log \log n)$ is the (maximum) height of the auxiliary tree. For the actual implementation details, see [Dem+07].

6.4.3. Changing the Preferred Child

We already noticed, that changing a preferred child corresponds to cutting and joining paths together. The following algorithm summarizes that.

Algorithm 30: Changing the preferred child in $\mathcal{O}(h)$ time.

```
1 change-pref-child(u, sibling(u)):
2   P2 = sibling(u)
3   T = root(u)
4   d = u.depth
5
6   P1, T1 = cut(T, d)
7   T2 = join(P1, P2)
8
9   return (T1, T2)
```

sibling(u) is the sibling node of u in the perfect binary tree P. The auxiliary trees P_1, P_2, T_1 and T_2 represent respectively the preferred path of u with depth $< d$, the preferred path of the sibling of u, the preferred path of u after the changed preferred child and the preferred path of the sibling after the changed preferred child.

join can be executed, because the depth of u and the depth of the sibling is the same. The preferred child can be changed in $\mathcal{O}(h)$ time, because *root*, *cut* and *join* take at most $\mathcal{O}(h)$ time.

6.5. Managing the Minimal and Maximal Leaves

In this section we will show, how we use tango trees to manage the minimal and maximal leaves. We will use two tango trees T_{min} and T_{max}, which respectively manage the minimal and maximal leaves separately. Since both trees are kind of symmetric, we will mainly argue how T_{min} manages the minimal leaves.

The main idea is that a minimal leaf of a subtree rooted at a node u in the compact trie T is shared by at most by w nodes in T. To be more precise, the minimal leaf of a node u is recursively defined as the minimal leaf of the left subtree of u. That means, that the preferred child of u is always the left child and that the preferred path formed by this definition contains all nodes which share the same minimal leaf. Of course, the maximal element defines the preferred child as the right child for each node.

Furthermore notice, that the tree P does not have to be a perfect binary tree and in fact can be any static[1] binary search tree. The only reason why P was chosen as a perfect binary search

[1] Static in the sense, that the depths of the nodes do not change.

tree is that it guarantees that the longest root-to- leaf path is always at most $\mathcal{O}(\log n)$ long and that therefore the heights of the auxiliary trees are bounded by $\mathcal{O}(\log \log n)$.

Therefore, we will chose P as the compact trie of the Z-fast trie, which according to Lemma 2.14 is a binary search tree. We have to show that the requirement of the static depth is given. Let u be a node in the compact trie and u_{min} and u_{max} the respective nodes of u in the tango trees T_{min} and T_{max}. Define $u_{min}.depth$ and $u_{max}.depth$ as $|u.key|$. This definition works, because as long as u is not deleted, the key of u remains the same.

Each node in the compact trie has two additional pointers:

> ***min-node***: points to the corresponding node in the auxiliary tree of the tango tree T_{min} which manages the minimal leaves.
>
> ***max-node***: points to the corresponding node in the auxiliary tree of the tango tree T_{max} which manages the maximal leaves.

And each node in T_{min} and T_{max} has the following additional pointer:

> ***trie-node***: points to the corresponding node in the compact trie.

The last remark is an implementation detail. Since all nodes in each auxiliary tree of the tango trees T_{min} and T_{max} are prefixes of the minimal or maximal leaf, it suffices to use the depth of each node as key. Instead of using the actual prefix as key. Thus, the ordering will be the same.

6.5.1. Finding the Minimal and Maximal Leaf

Finding the minimal leaf in the compact trie is easy, because it is also the minimal element in the corresponding auxiliary tree. This is a direct result from the fact, that all nodes that share the same minimal leaf are prefixes of the minimal leaf.

The algorithm switches into the corresponding auxiliary tree, which contains u', goes up to the root of auxiliary tree, finds the minimal element u'_{min} and finally switches back to the corresponding node in the compact trie. This procedure takes at most $\mathcal{O}(\log \log |U|)$ time, because in the auxiliary tree are at most $\mathcal{O}(w) = \mathcal{O}(\log |U|)$ nodes.

Algorithm 31: The minimal and maximal leaf of a subtree u in $\mathcal{O}(\log \log |U|)$ time.

```
min-leaf(u):
  u' = u.min-node
  u'_root = root(u')
  u'_min = min(u'_root)
  return u'_min.trie-node
```

```
max-leaf(u):
  u' = u.max-node
  u'_root = root(u')
  u'_max = max(u'_root)
  return u'_max.trie-node
```

6.5.2. Inserting an Element

Let b be the inserted branch node, u be the parent of b, q be the leaf, which belongs to the inserted element q and v be the sibling of q. For an illustration see Figure 8. After inserting the element q, the minimal and maximal elements below u may change. Like before, we will only consider T_{min}.

After inserting the branch node b, the preferred child of b must be assigned to either v or q. We assign the preferred child of b to v by inserting $b.min\text{-}node$ into the auxiliary tree of v. If the preferred child of b is not the left child of b, we will change it to the left child. Note that this procedure solves the four different possible cases depicted in Figure 8. For example if v was a left child of u, inserting $b.min\text{-}node$ into the auxiliary tree of v also assigns the preferred child of u, because b was inserted into the same preferred path as u is.

In the following algorithm, finding the root of an auxiliary tree, inserting a node into an auxiliary tree and changing a preferred child takes at most $\mathcal{O}(\log\log|U|)$ time.

Algorithm 32: Insertion; Update min-/maximal leaves in $\mathcal{O}(\log\log|U|)$ time.

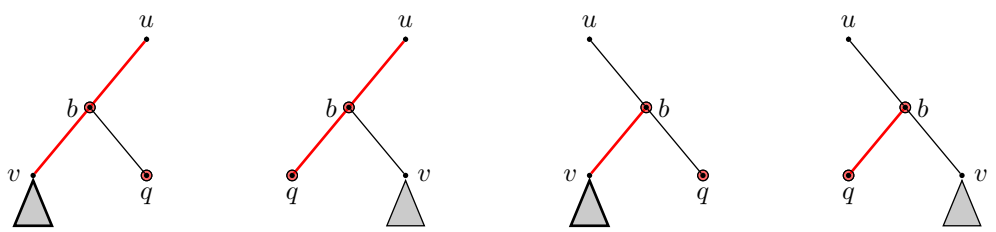
```
insert-min-leaves(b, v, q):
  v' = v.min-node
  b' = b.min-node # newly created
  q' = q.min-node # newly created

  T_min = root(v')
  T_min.insert(b')

  if q is left child of b:
    change-pref-child(v', q')
```

```
insert-max-leaves(b, v, q):
  v' = v.max-node
  b' = b.max-node # newly created
  q' = q.max-node # newly created

  T_max = root(v')
  T_max.insert(b')

  if q is right child of b:
    change-pref-child(v', q')
```

6.5.3. Removing an Element

Removing an element q is in some sense the reverse of the insertion. Therefore, the following is before the actual deletion of q. Let b be the branch node, which is the parent of q and let v be the sibling of q. We will only consider the update of T_{min}.

If q is the preferred child of b (i.e. the left child of b), change it to the sibling v. Afterwards remove b from its auxiliary tree. Due to the preferred child change, q is the only node in its auxiliary tree and does not need to be removed explicitly.



(a) When v was the left child of u. (b) When v was the right child of u.

Figure 8: Z-fast Trie - The different possibilities of the path decomposition of the minimal leaves after an insertion.

Algorithm 33: Deletion; Update min-/maximal leaves in $\mathcal{O}(\log\log|U|)$ time.

```
remove-min-leaves(b, v, q):
  v' = v.min-node
  b' = b.min-node
  q' = q.min-node

  if q is left child of b:
    change-pref-child(q', v')

  T_min = root(b')
  T_min.remove(b')

  destroy b' and q'
```

```
remove-max-leaves(v, b, q):
  v' = v.max-node
  b' = b.max-node
  q' = q.max-node

  if q is right child of b:
    change-pref-child(q', v')

  T_max = root(b')
  T_max.remove(b')

  destroy b' and q'
```

6.6. Dynamic Predecessor Problem

Theorem 6.2 *The Z-fast trie solves the dynamic predecessor problem with $\mathcal{O}(\log\log|U|)$ time per operation and linear space.*

Proof. According to Theorem 2.2 all operations have the runtime $\mathcal{O}(\log\log|U|)$, if finding the lowest common ancestor and managing the minimal and maximal leaves have the given runtime, which is fulfilled by Lemma 6.2 and Algorithms 31 to 33. The tango trees T_{min} and T_{max} need additional linear space, because they have as many nodes as there are nodes within the compact trie. Concluding with Lemma 6.1, that the total space usage is linear. □

7. μ-fast Trie

In this section, we will look at the μ-fast trie, which was introduced by Pătraşcu [Pät10c]. The μ-fast trie solves the static predecessor problem in $\mathcal{O}(\log\log|U|)$ time and linear space. The preprocessing time is $\mathcal{O}(n\log\log|U|)$. This approach is unlike the previous techniques not easily adaptable to the dynamic predecessor problem.

7.1. Data Structure

The data structure is similar to the X-fast trie. That means we will store a subset $\mathcal{V} \subseteq V$ of nodes of a non-compact trie $G = (V, E)$ in the hash map. We will map a node $v \in \mathcal{V}$ to the lowest branch node of the node v, i.e.

$$T[v.key] = \max\{u \in pre(v) \mid u \text{ has two children or is the root } \} .$$

Note that the μ-fast trie does not need to store a non-compact trie explicitly. Since the hash map points only to branch nodes, it suffices to store the corresponding compact trie. The static predecessor operations will be reduced to the compact trie operations. For that we have to find the lowest common ancestor for an element q of the compact trie (Theorem 2.2). We have seen in Lemma 2.24 that a compact trie only needs linear space.

We assume that each subtree in the compact trie has a pointer to its minimal and maximal leaf and that the leaves are linked together. We can assume this, because we will only consider a data structure for the static predecessor problem.

The technique to achieve linear space in the μ-fast trie is to reduce the number of nodes stored in the hash map to $|\mathcal{V}| \in \mathcal{O}(n\sqrt{w})$[1] and to allocate at most $\log w$ bits per node in a hash map. This results in $\mathcal{O}(n\sqrt{w} \cdot \log w) = o(nw)$ allocated bits, which can be packed into $\mathcal{O}(n)$ words[2], thus linear space.

7.2. Node Reduction

As we already mentioned, the search for the lowest common ancestor uses the hash map T which contains nodes of a non-compact trie, but we need the definition of the lowest common ancestor definition of a compact trie to implement the static predecessor operations. During the search, a more proper name would be lowest common branch ancestor, but we will stick to the term lowest common ancestor. In this section, the definition of $\mathrm{lca}(q)$ will be

$$\mathrm{lca}(q) = \max\{u \in pre(q) \mid u \text{ has two children or is the root } \} .$$

Split the Binary Search

The reduction of the number of nodes is accomplished by splitting the lowest common ancestor search into two binary search phases, where both preserve the $\mathcal{O}(\log\log|U|)$ search time. The first phase is to find the longest common prefix of every $\sqrt{w}$-th node

$$k = \max\left\{k \in \{0, \ldots, \lfloor\!\lfloor\sqrt{w}\rfloor\!\rfloor\} \mid p = q_0 \ldots q_{k\lceil\!\lceil\sqrt{w}\rceil\!\rceil} \text{ and } T[p] \neq \bot\right\} .$$

[1] A X-fast trie stores $\mathcal{O}(nw)$ nodes.
[2] A word has space for w bits.

Whereas, the second phase is to find the longest common prefix in the remaining range of $\sqrt{w}$ nodes

$$\mathrm{lcp}(q) = \max\left\{p = q_0 \dots q_i \mid i \in \{k\lceil\sqrt{w}\rceil, k\lceil\sqrt{w}\rceil + 1 \dots, (k+1)\lceil\sqrt{w}\rceil - 1\} \text{ and } T[p] \neq \bot\right\}$$

One could think that $k\lceil\sqrt{w}\rceil$ is an expensive operation, but we have seen in Lemma 2.12 that it can be computed in $\mathcal{O}(1)$ time and that it is just a shift operation. Furthermore, we have seen that $\lfloor\sqrt{w}\rfloor, \lceil\sqrt{w}\rceil$ is bounded by $\Theta(\sqrt{w})$.

Reduce Nodes

Let $\mathcal{V}_1, \mathcal{V}_2 \subseteq V$ be the set of nodes such that the first phase and second phase of the lowest common ancestor search works.

It is clear that $\mathcal{V}_1$ contains for each $s \in S$ all prefixes on $\sqrt{w}$ levels, i.e. $s_0 \dots s_{k\lceil\sqrt{w}\rceil} \in \mathcal{V}_1$ for all $k \in \{0, \dots, \lfloor\sqrt{w}\rfloor\}$. Otherwise the first phase would not work. Note that this are $|\mathcal{V}_1| = \mathcal{O}(n\sqrt{w})$ nodes.

In the following lemma, we will show that for the second phase it suffices that $\mathcal{V}_2$ contains for each branch node v of the non-compact trie, v and $\lceil\sqrt{w}\rceil$ ancestor nodes directly above[1] v. Note that this are $|\mathcal{V}_2| = \mathcal{O}(n\sqrt{w})$ nodes.

Thus, the hash map needs to store the set $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$ of $\mathcal{O}(n\sqrt{w})$ elements.

Lemma 7.1 *The first and second search phase yields the lowest common ancestor of q, while only using $|\mathcal{V}| = \mathcal{O}(n\sqrt{w})$ nodes.*

Proof. The first search phase delivers a k, where all nodes below[2] the depth $(k+1)\lceil\sqrt{w}\rceil$ have no common prefix with the search query q. That means that the longest common prefix $\mathrm{lcp}(q)$ is within the interval $k\lceil\sqrt{w}\rceil$ and $(k+1)\lceil\sqrt{w}\rceil$. Note that the first search phase can be done by a binary search.

Let us shortly describe how the second binary search on the set $I_k = \{k\lceil\sqrt{w}\rceil, k\lceil\sqrt{w}\rceil + 1 \dots, (k+1)\lceil\sqrt{w}\rceil - 1\}$ for the second phase works. Let $p \in pre(q)$ be the current prefix in the binary search with $|p| \in I_k$. The binary search proceeds with a shorter prefix if $T[p] = \bot$ and with a longer prefix when $T[p] \neq \bot$.

Let v be the prefix found by the second binary search. The search guarantees $T[v] \neq \bot$. Clearly, $k\lceil\sqrt{w}\rceil \leq |v| \leq |\mathrm{lcp}(q)| < (k+1)\lceil\sqrt{w}\rceil$. We will show that $T[v]$ is the lowest common ancestor of q. For example, if the binary search finds the longest common prefix $\mathrm{lcp}(q)$, then $T[v]$ is obviously $\mathrm{lca}(q)$.

Note the following fact. Let u be an arbitrarily branch node within the interval. Since we added $\lceil\sqrt{w}\rceil$ nodes above u to the hash map and the interval is smaller than the number of added prefixes, all prefixes of the branch node reach above the depth $k\lceil\sqrt{w}\rceil$.

There are three cases that can happen during the second binary search. But each case results in $T[v] = \mathrm{lca}(q)$.

In the first case, there is a branch node u within the interval, which has $\mathrm{lcp}(q)$ as ancestor. Since all prefixes of u reach at least up to the depth $k\lceil\sqrt{w}\rceil$, all important nodes for the second binary

[1] Above in the sense, that the nodes have a lower depth in the trie.
[2] Below in the sense, that the nodes have a higher depth in the trie.

search are present and the second binary search will find the longest common prefix $\text{lcp}(q) = v$. Thus, $T[v] = \text{lca}(q)$. This is illustrated in Figure 9a.

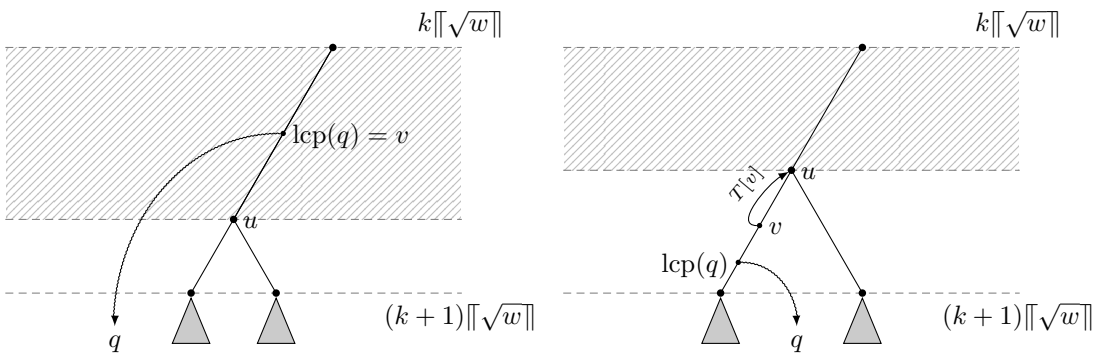
In the second case, there is a branch node within the interval which is an ancestor of $\text{lcp}(q)$, but there is no branch node below $\text{lcp}(q)$. Let u be the lowest such branch node. That branch node u is by definition the lowest common ancestor of q. The second binary search will return a node v with $u \leq v \leq \text{lcp}(q)$, because the hash map contains at least all nodes between $k\lceil\sqrt{w}\rceil$ and u. But notice that there is no branch node between u and v. Therefore, $T[v]$ points to its lowest ancestor branch node, which is u. Thus, $T[v] = u = \text{lca}(q)$. This case is depicted in Figure 9b.

In the lase case, there is no branch node within the interval below and above $\text{lcp}(q)$. Since $v \leq \text{lcp}(q)$, there is no branch node between v and the node at depth $k\lceil\sqrt{w}\rceil$ and no branch node between v and $\text{lcp}(q)$. Thus, v and $\text{lcp}(q)$ have the same lowest branch node. Since we know that $T[\text{lcp}(q)] = \text{lca}(q)$, $T[v] = \text{lca}(q)$ points to the lowest common ancestor of q. □

7.3. Search Modification

Right now the hash map needs $\mathcal{O}(n\sqrt{w})$ words space. To reduce the space, we modify the hash access $T[p] = u$. For that, we assume that T stores the depth of the lowest branch node, i.e. $T[p] = |u|$ and that a second hash map T_b stores all nodes from the compact trie, i.e all branch nodes from a non-compact trie.

Instead of directly accessing $T[p] = u$, we will first find $d = T[p]$, then $u = T_b[p_0 \dots p_{d-1}]$, after that verify that u is the lowest branch node[1] of p and finally return u. If any of these steps fail, return $\perp$.



(a) The first case, where all prefixes from u until $k\lceil\sqrt{w}\rceil$ are inserted and therefore, $\text{lcp}(q)$ will be found by the binary search. Thus, $T[\text{lcp}(q)] = \text{lca}(q)$.

(b) The second case, where the binary search finds v, but, since no branch nodes are between $\text{lcp}(q)$ and u, the nodes in that range points to u. Thus, $T[v] = \text{lca}(q)$.

Figure 9: μ-fast Trie - The fist two cases in the proof, where the hatched area depicts the nodes in the hash table, which are definitely inserted.

[1] A node u is the lowest branch node of p, iff $u = p$ or $v \neq p \in (u, v)$ is on the edge (Definition 2.21), where v is one of the children of u.

To avoid confusion, T' is the ***original definition*** of T. that means it stores the lowest branch node.

Lemma 7.2 *Assuming that $T[p]$ returns an arbitrarily depth, if $T'[p] = \perp$. The above described procedure is a replacement of the statement $T'[p] \neq \perp$.*

Proof. If $u = T'[p] \neq \perp$, the hash table T will return $|u| = d = T[p]$. Since $u.key$ is a prefix of p, $u = T_b[p_0 \dots p_{d-1}]$ can be retrieved using p. And since u is the lowest branch node of p, the final check whether u is the lowest branch node of p will be true.

Now assume $T'[p] = \perp$. The hash table T will return an arbitrary depth $d = T[p]$. Let $u.key$ be $p_0 \dots p_{d-1}$ and let us assume that $u = T_b[u.key]$ is the lowest branch node of p, since each failing step would result in the desired returned $\perp$.

But notice that this will not break the binary search within the interval $[k\lceil\sqrt{w}\rceil, (k+1)\lceil\sqrt{w}\rceil)$, because we already accounted in the binary search that in the second and third case longer prefixes could be found, but that they do not change the final result of the binary search. Thus, the procedure is indeed a replacement for the statement $T'[p] \neq \perp$. □

Algorithm 34: Modified $T[p] \neq \perp$ in $\mathcal{O}(1)$ time.

```
d = T[p]
# branch node must be a prefix of p
if d > |p|:
    return false

# the node u is not a branch node
u = p_0 ... p_{d-1}
if T_b[u] ≠ p:
    return false

# p is the branch node
if p = u:
    return true

# u is the lowest branch node of p
if p ≠ v and p is on the edge
    between u and one of its
    children v:
    return true
return false
```

Lemma 7.3 *The lowest common ancestor operation of a query q can be found in $\mathcal{O}(\log\log|U|)$ time.*

Proof. The first and second phase are binary searches on $\mathcal{O}(\sqrt{w})$ prefixes. Each step of the binary search takes $\mathcal{O}(1)$ time (Algorithm 34). Thus, the runtime is $\mathcal{O}(\log\sqrt{w}) = \mathcal{O}(\log\log|U|)$. □

7.4. Linear Space

Lemma 7.4 *The μ-fast trie has linear space.*

Proof. The hash map T_b needs linear space, because there are only linear many branch nodes (Lemma 2.24). For the hash map T_Δ, we will use the retrieval only hash map in Theorem 2.4. The space requirement for this hash map is $\mathcal{O}(n'\log\log\frac{|U|}{n'} + n'r)$ bits for n' items and r bits associated data per item. Since $n' = \mathcal{O}(n\sqrt{w})$ and $r = \lceil\log w\rceil$, i.e. the possible depths $\{0, \dots, \log w - 1\}$ encoded as block code, the space needed is

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

where $w = \log |U|$. Or stated differently $\mathcal{O}(n)$ words. As already mentioned, the compact trie needs linear space. Thus, the μ-fast trie has linear space. □

7.5. Static Predecessor Problem

Theorem 7.1 *The μ-fast trie solves the static predecessor problem in $\mathcal{O}(\log\log|U|)$ time per operation, while only using linear space. The preprocessing time is $\mathcal{O}(n\sqrt{w})$.*

Proof. We have seen in Lemma 7.4 that the space usage is linear. Further we have seen that it suffices to store the compact trie. Since we use the definition of the lowest common ancestor from the compact trie, the predecessor, successor and search operation are the same as for the compact trie and their runtime is $\mathcal{O}(\log\log|U|)$ (Lemma 7.3).

The preprocessing time is at least $\mathcal{O}(n\sqrt{w})$, because $\mathcal{O}(n\sqrt{w})$ nodes have to be inserted into the hash table T. That means there is enough time to sort the set S. Using the deterministic sort algorithm by Han [Han02], the set S can be sorted in $\mathcal{O}(n\log\log n) \leq \mathcal{O}(n\log\log|U|) = \mathcal{O}(n\log w) = \mathcal{O}(n\sqrt{w})$ time. Thus, the other data structures like the compact trie (Lemma 2.25) and the hash map T_b can be constructed in linear time. □

7.6. Dynamic Predecessor Problem

Unfortunately, this data structure is not easily adaptable to the dynamic case with $\mathcal{O}(\log\log|U|)$ update time, because each update could add / update at least $\Omega(\sqrt{w})$ nodes to / in the hash map T. Thus, some update operation take at least $\omega(\log\log|U|)$ time.

8. Δ-fast Trie

The Δ-fast trie is derived from a data structure by Bose *et al* [Bos+13]. Let $q^+ := \min\{s \in S \mid s \geq q\}$ and $q^- := \max\{s \in S \mid s \leq q\}$. They used X-fast tries, Y-fast tries and a skip list [Pug90] to build a data structure, which can answer predecessor and successor queries in $\mathcal{O}(\log\log\Delta)$ time with $\Delta = \min\{|q - q^-|, |q - q^+|\}$. The new data structure developed in this section solves an open problem stated in [Bos+13] regarding the space usage. Their solution needed $\mathcal{O}(n \log\log\log|U|)$ words space, which is not optimal. Here, we will combine the Z-fast trie with the compression idea of the Δ-fast trie to achieve an optimal space usage of $\mathcal{O}(n)$ words.

8.1. Data Structure

We use the data structure of the Z-fast trie as the starting point and augment it. We call the hash map of the Z-fast trie T_z and we have an additional hash map called T_Δ. This hash map will store – like in the X-fast trie – nodes at certain depths of the leaf-to-root path for each item $s \in S$. This will enable the improved search time of $\mathcal{O}(\log\log\min\{|q - q^-|, |q - q^+|\})$.

8.2. The Predecessor Search

We first show how to get $\mathcal{O}(\log\log\Delta)$ time with $\Delta = |q - q^+|$ and later show how to achieve $\Delta = \min\{|q - q^-|, |q - q^+|\}$.

8.2.1. Overview of the Search

We will give an overview of the predecessor and successor search. The missing details will be explained in the following subsections.

Fix some strictly increasing sequence $h_0, \ldots, h_m$ with $0 \leq h_i \leq w$ and $h_m = w$ for $i = 1, \ldots, m$. We will later give a definition for that sequence. Let $d_i := w - h_i$ be a sequence for $i = 1, \ldots, m$. The hash map T_Δ stores for each $s \in S$ and each d_i, $i = 1, \ldots, m$

$$T_\Delta[p_i] = u,$$

where $p_i = q_0 \ldots q_{d_i - 1}$ and $p_i \in (u, v) \in E$ (i.e. p_i is on the edge e, Definition 2.21).

The predecessor or successor search works as follows. Start with $d_i = d_1$.

- If $T_\Delta[p_i] \neq \perp$: Notice that p_i is a prefix of the longest common prefix of q. Use p_i to find the lowest common ancestor of q in the compact trie. We showed how to find the predecessor (or successor) in $\mathcal{O}(T_{\min}(q))$ time when the lowest common ancestor of q is given (Algorithm 7).

  The details of this case will be explained in Section 8.2.4. We will see that finding the lowest common ancestor takes $\mathcal{O}(\log h_i)$ time and show in Section 8.5.2 how to find the minimal or maximal leaf of a subtree in $T_{\min}(q) \in \mathcal{O}(\log h_i)$ time.

- If $T_\Delta[p_i{+}1] \neq \perp$: Notice that $p_i{+}1$ is the successor of p_i, e.g. if $p_i = 0000$ then $p_i{+}1 = 0001$. If p_i has no successor, skip this case.

First obtain $u = T_\Delta[p_i + 1]$ and the child of v where p_i is on the edge $e = (u, v)$, i.e. $p_i \in e$. Then find the minimal element in the subtree of v, this is the successor q^+. Thus, $q^+ = v.min\_leaf$. The predecessor q^- can be found by following the pointer of q^+ to q^-.

This case takes $T_{\min}(q)$ time to find the minimal leaf of the subtree v. We already stated that we will show that $T_{\min}(q) \in \mathcal{O}(\log h_i)$.

In Section 8.2.2 we will cover the details of this case.

- If no case applies, then continue with d_{i+1}.

The runtime is $\mathcal{O}(k + \log h_k)$ where k is the number of steps in the loop and $\log h_k$ is the time to find the lowest common ancestor and the minimal or maximal leaf of a subtree.

Therefore to achieve $\mathcal{O}(\log\log\Delta)$ time, k should be at most $\mathcal{O}(\log\log\Delta)$ and $h_k = \mathcal{O}(\log^{\mathcal{O}(1)}\Delta)$ should be polynomial in $\log\Delta$. Furthermore, the space of T_Δ is at most n times m words, where m is the length of the sequence $(h_i)_{1\le i\le m}$.

In Section 8.2.3 we will see that $(h_i)_{1\le i\le m} = (2^{2^i})_{1\le i\le m}$, that $h_k = \mathcal{O}(\log^2\Delta)$, that $k = \Theta(\log\log\log\Delta)$ and that $m = \lceil\log\log w\rceil$. Thus, showing that the runtime is $\mathcal{O}(k + \log h_k) = \mathcal{O}(\log\log\Delta)$ and that the required space for T_Δ are $\mathcal{O}(n\log\log w) = \mathcal{O}(n\log\log\log|U|)$ words. In Section 8.3 we will show how to reduce the space of T_Δ to $\mathcal{O}(n)$ words.

8.2.2. Finding the Predecessor from $T_\Delta[p_i + 1]$

This section shows that the given procedure of the second case of the predecessor search is correct. In this case $T_\Delta[p_i] = \perp$ and $T_\Delta[p_i + 1] \neq \perp$ holds.

Definition 8.1 *I_p is the set $\{p0^{w-|p|}, \ldots, p1^{w-|p|}\} \subseteq U$ which contains all integers with the prefix p.*

If p would be a node in a trie, I_p would represent all elements that could be below that node. It is easy to see that $F_l = \{I_p \mid p \in \{0,1\}^l\}$ is a partition of U for all bit-strings p of length l.

Lemma 8.2 *$T_\Delta[p] \neq \perp$ iff $I_p \cap S \neq \emptyset$.*

Proof. If $I_p \cap S \neq \emptyset$, then let $x \in I_p \cap S$. Since x has a leaf in the trie and x has the prefix p, the leaf-to-root path of x contains an edge e with $p \in e$. Thus, $T_\Delta[p] = u$ is defined. If $I_p \cap S = \emptyset$, then there is no element in the trie with prefix p. But that means that there is no edge e with $p \in e$. Thus, $T_\Delta[p] = \perp$ is undefined. □

Since in this case is $T_\Delta[p_i] = \perp$ and $T_\Delta[p_i + 1] \neq \perp$, it follows from the above lemma that $I_{p_i} \cap S = \emptyset$ and $I_{p_i+1} \cap S \neq \emptyset$. That means that the minimum in $I_{p_i+1} \cap S$ is the successor of q, i.e. $q^+ = \min(I_{p_i+1} \cap S)$.

Let $e = (u, v)$ be the edge with $(p_i+1) \in e$. In the overview of the predecessor search we basically said that $q^+ = \min(I_v \cap S)$. Thus, we have to show $\min(I_v \cap S) = \min(I_{p_i+1} \cap S)$. The reason is illustrated in Figure 10. It is clear that $I_v \subseteq I_{p_i+1} \subsetneq I_u$ and therefore $I_v \cap S \subseteq I_{p_i+1} \cap S$. Since (p_i+1) is on the edge $e = (u, v)$, (p_i+1) is either the node v or no node at all. In both cases has the subtree of $(p_i + 1)$ no more elements than the subtree of v. Thus, $I_v \cap S = I_{p_i+1} \cap S$.

8.2.3. Choosing the correct Height Sequence h_i

In this section, we define the sequence $(h_i)_{1 \leq i \leq m}$, show that the length of the sequence is $m = \lceil \log \log w \rceil$ and that the number of loop iterations is at most $k = \mathcal{O}(\log \log \log \Delta)$.

Let $\#succ_i(q)$ be the ***number of covered successors*** in the i-th iteration, i.e.

$$\#succ_i(q) = |\{x \in I_{p_i} \cup I_{p_{i+1}} \subseteq U \mid x > q\}| \ .$$

Notice that $\#succ_i$ is monotonically increasing, $\#succ_1(q) \leq \#succ_2(q) \leq \ldots \leq \#succ_m(q)$.

Assume that we are at the end of the i-th iteration. Observe that $|I_{p_i}| = |I_{p_i+1}| = 2^{h_i}$ and that $|(I_{p_i} \setminus \{q\}) \dot{\cup} I_{p_{i+1}}| = 2^{h_i+1} - 1 < 2^{h_i+1}$. Thus, $2^{h_i} \leq \#succ_i(q) < 2^{h_i+1}$.

Each time a loop continues, q^+ was neither in I_{p_i} nor in I_{p_i+1}. Hence, $2^{h_i} \leq \#succ_i(q) < \Delta$.

Let k be the iteration where the first time $q^+ \in I_{p_k}$ or $q^+ \in I_{p_k+1}$. But then covered the search more than Δ successors, i.e. $\Delta \leq \#succ_k(q) < 2^{h_k+1}$ and we know from the $(k-1)$-th iteration that $2^{h_{k-1}} \leq \#succ_{k-1}(q) < \Delta$. Thus,

$$2^{h_{k-1}} < \Delta < 2^{h_k+1} \qquad \text{and} \tag{21}$$

$$h_{k-1} < \log \Delta \leq h_k < h_k + 1 \ . \tag{22}$$

Therefore, $h_k \in \Omega(\log \Delta)$. We will now show that $h_k \in \mathcal{O}(\log^{\mathcal{O}(1)} \Delta)$.

Exponential Search

Define h_i as $h_i := 2^i$. Then $h_k = 2^k \in \mathcal{O}(\log \Delta)$, because

$$h_k = 2^k = 2h_{k-1} < 2 \log \Delta \ .$$

That also means that the number of loop iterations k is bounded by $\Theta(\log \log \Delta)$ and the number m of stored nodes in the hash map is $\lceil \log w \rceil$, since $\lceil \log w \rceil$ is the smallest number with $h_m \geq w$.

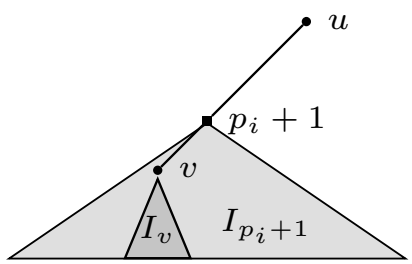


Figure 10: Δ-fast Trie - An illustration for $I_{p_i} \cap S = I_v \cap S$.

Doubly Exponential Search

The previous result can be improved by defining h_i as $h_i := 2^{2^i}$. In this case is $h_k = 2^{2^k} \in \mathcal{O}(\log^2 \Delta)$, since

$$h_k = 2^{2^k} = \left(2^{2^{k-1}}\right)^2 = (h_{k-1})^2 < \log^2 \Delta \ .$$

This improves k to $\Theta(\log\log\log\Delta)$ and m to $\lceil \log\log w \rceil$.

Note that a triply exponential search would yield poorer results, because h_k would not be polynomial in $\log \Delta$ anymore.

8.2.4. Continuing the Lowest Common Ancestor Search

In this section, we will give the details of the first case of the predecessor search of the Δ-fast trie. That means we will show how to find the lowest common ancestor of q from a given prefix $p_k \in pre(\mathrm{lcp}(q)) \subseteq pre(q)$. We know that $T_\Delta[p_k] \neq \perp$ holds in this case.

We will use p_k to find a synchronization point where the algorithm $search_I$ can continue. Remember that $search_I(q) = \max\{\alpha \in pre(q) \mid T[\alpha] \neq \perp\}$ was the main part of the lowest common ancestor search in the Z-fast trie (Algorithm 29).

Let us restate the parameters of the algorithm $search_I$:

- q' is the remaining suffix of q, which the binary search needs to check.
- lcp' is the currently found longest common prefix of q. The search maintains the invariant that $concat(lcp', q')$ is a prefix of q.
- lca' is the currently found lowest common ancestor of q.
- w' is the length of q'

The goal is to perform the lowest common ancestor search in $\mathcal{O}(\log h_k)$ time. For that we have to show that $search_I$ proceeds with a q' of only $w' = |q'| = \mathcal{O}(h_k)$ bits.

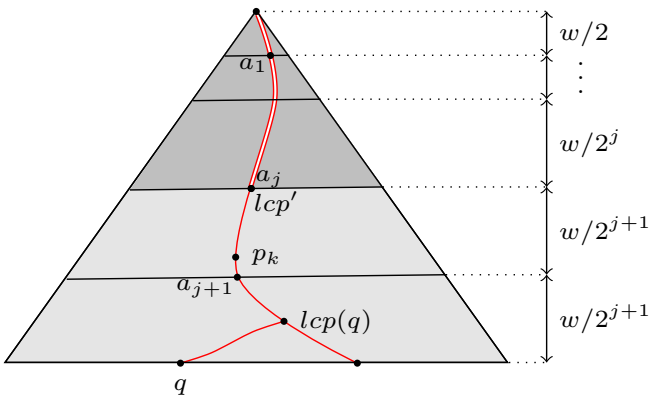


Figure 11: Δ-fast Trie - Finding lcp' and q' to perform the lowest common ancestor search $search_I$ in $\mathcal{O}(\log h_k) = \mathcal{O}(\log\log\Delta)$ time.

The main idea is that we split $q = q_0 \dots q_{w-1}$ at some position l and let $lcp' := q_0 \dots q_{l-1}$ and $q' := q_l \dots q_{w-1}$. Let $l_j := \sum_{i=1}^{j} \frac{w}{2^i}$ and let $a_j := q_0 \dots q_{l_j}$ for $j = 1, \dots, \log w$. See Figure 11 for an illustration of the positions of a_j in the trie.

Let j be the highest index where a_j is above p_k, i.e. $j := \max\{i \mid a_i \in pre(p_k)\}$. Since a_j is per definition aligned to a binary search on prefixes of q, we can choose $lcp' := a_j$ and $q' := q_j \dots q_{w-1}$ for the algorithm $search_I$.

Lemma 8.3 $w' = |q'| \in \mathcal{O}(h_k)$

Proof. Let m be $|a_{j+1}|$. Since a_j was chosen as $a_j \leq p_k < a_{j+1}$, $|lcp'| = |a_j| \leq |p_k| < |a_{j+1}| = m$ and $-m < -|p_k|$ holds. $|p_k|$ is defined as $d_k = w - h_k$. Now notice that

$$|lcp'| = \sum_{i=1}^{j} \frac{w}{2^i} = 2\left(\sum_{i=1}^{j+1} \frac{w}{2^i}\right) - w = 2m - w \ . \tag{23}$$

Thus,

$$|q'| = w - |lcp'| = w - (2m - w) = 2w - 2m \tag{24}$$

$$< 2w - 2|p_k| = 2w - 2(w - h_k) = 2h_k \tag{25}$$

□

The only problem left is how to find a_j algorithmically. The next lemma shows that we do not have to compute a_j.

Lemma 8.4 $a_j = p_k$

Proof. Since w is a power of two, the following two equalities hold.

$$\sum_{i=l_1}^{l_2} \frac{w}{2^i} = 2^{\log w - l_1} + 2^{\log w - l_1 - 1} + \dots + 2^{\log w - l_2} = \sum_{i=\log w - l_2}^{\log w - l_1} 2^i \tag{26}$$

$$w = 1 + \sum_{i=1}^{\log w} \frac{w}{2^i} \tag{27}$$

Notice that

$$|lcp'| = \sum_{i=1}^{j} \frac{w}{2^i} = w \sum_{i=1}^{j} 2^{-i} = w - 2^j \ . \tag{28}$$

The following calculation shows that $|h_k| = w - 2^j = |lcp'|$.

$$|a_j| = \sum_{i=1}^{j} \frac{w}{2^i} \leq |p_k| < \sum_{i=1}^{j+1} \frac{w}{2^i} = |a_{j+1}| \tag{29}$$

$$w - \sum_{i=1}^{j+1} \frac{w}{2^i} < h_k \leq w - \sum_{i=1}^{j} \frac{w}{2^i} \tag{30}$$

$$1 + \sum_{i=1}^{\log w} \frac{w}{2^i} - \sum_{i=1}^{j+1} \frac{w}{2^i} < h_k \leq 1 + \sum_{i=1}^{\log w} \frac{w}{2^i} - \sum_{i=1}^{j} \frac{w}{2^i} \qquad (Eq.\ 27) \tag{31}$$

$$1 + \sum_{i=j+2}^{\log w} \frac{w}{2^i} < h_k \leq 1 + \sum_{i=j+1}^{\log w} \frac{w}{2^i} \tag{32}$$

$$1 + \sum_{i=0}^{\log w - j - 2} 2^i < h_k \leq 1 + \sum_{i=0}^{\log w - j - 1} 2^i \qquad (Eq.\ 26) \tag{33}$$

$$2^{\log w - j - 1} < h_k \leq 2^{\log w - j} \tag{34}$$

Since $h_k = 2^{2^k}$, $2^{\log w - j - 1} < 2^{2^k} \leq 2^{\log w - j}$ and $\log w - j - 1 < \log w - j \leq 2^k \leq \log w - j$. Thus, $2^k = \log w - j$, $h_k = 2^{2^k} = 2^{\log w - j} = \frac{w}{2^j}$ and $|p_k| = w - h_k = w - \frac{w}{2^j} = |lca'|$. □

The following algorithm describes the first part of the lowest common ancestor in the Z-fast trie.

Algorithm 35: Computing $search_I$ from a given prefix p_k in $\mathcal{O}(\log h_k)$ time.

```
search*_I(T_z, T_Δ, q, p_k, w):
  lca = T_Δ[p_k]
  q' = q_{|p_k|+1} ... q_{w-1}

  return search_I(T_z, q', lca, p_k, h_k)
```

The next algorithm describes the second part of the lowest common ancestor in the Z-fast trie and is copied from Algorithm 25. They differ in one line where $search_I$ was substituted with $search_I^*$.

Algorithm 36: The lowest common ancestor search from a given prefix in $\mathcal{O}(\log h_k)$ time.

```
lca*(T_z, T_Δ, q, p_k, w):
  # special case q ∈ S
  if T[q] ≠ ⊥:
    return T[q]

  α = search*_I(T_z, T_Δ, q, p_k, w)
  u = T[α]

  # special case α = ε
  if α = ε:
    return u

  let (u, v) be the edge of α

  # lcp(q) is on the edge
  if α ≤ q ≤ v:
    return u

  return v
```

8.2.5. Result

Putting the results from the Sections 8.2.2 to 8.2.4 together into the following algorithm for the predecessor search.

Algorithm 37: Predecessor search in $\mathcal{O}(\log\log\Delta)$ time.

```
1  predecessor(T_z, T_Δ, q, w):
2    for i in 1,...
3      h_i = 2^(2^i)
4      d_i = w - max(0, h_i)
5
6      p_i = q_0 ... q_{d_i - 1}
7
8      if T_Δ[p_i] ≠ ⊥:
9        lca = lca*(T_z, T_Δ, q, p_i, w)
10       return predecessor from lca
11
12     if T_Δ[p_i + 1] ≠ ⊥:
13       u = T_Δ[p_i + 1]
14       v = child of u, which
             contains p_i
15       return v.min_leaf.prev_leaf
```

And the analogous successor search.

Algorithm 38: Successor search in $\mathcal{O}(\log\log\Delta)$ time.

```
1  successor(T_z, T_Δ, q, w):
2    for i in 1,...
3      h_i = 2^(2^i)
4      d_i = w - max(0, h_i)
5
6      p_i = q_0 ... q_{d_i - 1}
7
8      if T_Δ[p_i] ≠ ⊥:
9        lca = lca*(T_z, T_Δ, q, p_i, w)
10       return successor from lca
11
12     if T_Δ[p_i + 1] ≠ ⊥:
13       u = T_Δ[p_i + 1]
14       v = child of u, which
             contains p_i
15       return v.min_leaf
```

8.3. Linear Space

Using the same trick as in μ-fast tries, we can achieve linear space. The trick was to store the depth of the branch node u in T_Δ, instead of storing u itself. In order that the access $u = T_\Delta[p]$ works, we need an additional hash map T_b, which stores all branch nodes of the compact trie.

When accessing $u = T_\Delta[p]$ in the algorithm, first get the depth $d = |u|$ of the branch node u from the hash map T_Δ. After that get the branch node $u = T_b[q_0 \dots q_{d-1}]$ from the hash map T_b and finally check, whether u is really the lowest branch node of p. If any of those steps fail, return $\perp$.

We have to store $\mathcal{O}(n \log\log w)$ nodes in the hash map T_Δ, where each prefix p maps to the depth of its lowest branch node. The depth can be encoded with $\lceil\!\lceil \log w \rceil\!\rceil$ bits and it is the associated data of the prefix.

The space requirement for a retrieval only hash map for n' items and r bits associated data is $\mathcal{O}(n' \log\log \frac{|U|}{n'} + n'r)$ bits (Theorem 2.4). Therefore, the space requirement for T_Δ is

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

where $n' = n \log\log w$, $r = \lceil\!\lceil \log w \rceil\!\rceil$ and $w = \log |U|$. Or stated differently $\mathcal{O}(n)$ w-bit words.

Lemma 8.5 *The Δ-fast trie needs $\mathcal{O}(n)$ words space.*

Proof. We have seen, that the hash maps T_Δ, T_b and T_Z, as well as the compact trie needs $\mathcal{O}(n)$ words space. □

8.4. Static Predecessor Problem

Theorem 8.1 *The static Δ-fast trie solves the static predecessor problem with each operation in $\mathcal{O}(\log\log\min\{|q-q^-|, |q-q^+|\})$ time and linear space. The preprocessing time is $\mathcal{O}(n\log\log\log|U|)$ on sorted input.*

Proof. The normal search for $q \in S$ can be done in $\mathcal{O}(1)$ time by a lookup $T_z[q]$ in the hash map of Z-fast trie. We have seen that the predecessor and the successor can be found in $\mathcal{O}(\log\log|q-q^+|)$ time (Algorithms 37 and 38).

Note that $\mathcal{O}(\log\log|q-q^-|)$ can be achieved by exchanging $T_\Delta[p_i-1]$ with $T_\Delta[p_i+1]$ in the algorithm. The correctness follows the same arguments as in the case of $T_\Delta[p_i+1]$. With the difference that $T_\Delta[p_i-1]$ will find the predecessor element instead of the successor element. Thus, the search time is $\mathcal{O}(\log\log|q-q^-|)$.

Therefore when interleaving both computations the runtime will be $\mathcal{O}(\log\log\min\{|q-q^-|, |q-q^+|\})$. The pragmatically way would be to add the case $T_\Delta[p_i-1] \neq \perp$ between $T_\Delta[p_i] \neq \perp$ and $T_\Delta[p_i+1] \neq \perp$ in the Algorithms 37 and 38.

The preprocessing time primarily depends on the time to fill the hash map T_Δ, since the construction of the compact trie and the hash maps T_z and T_b can be done in linear time. The preprocessing time is $\mathcal{O}(n\log\log\log|U|)$, because $\mathcal{O}(n\log\log w)$ nodes have to be inserted into the hash map. The space requirement is linear (Lemma 8.5). □

8.5. Dynamic Predecessor Problem

We will now consider the dynamic case. That means we will argue the runtime of the update operations and show that the minimal and maximal leaves can be managed in $\mathcal{O}(\log\log\Delta)$ time.

8.5.1. Lowest Common Ancestor

The lowest common ancestor operation is needed for the update operations. It would be nice if the lowest common ancestor could be found in $\mathcal{O}(\log\log\Delta)$ time. We will use the same algorithm to find the lowest common ancestor as we did in the predecessor search, but without the cases $T[p_i-1] \neq \perp$ and $T[p_i+1] \neq \perp$.

Algorithm 39: Lowest common ancestor search in $\mathcal{O}(\log\log\Delta)$ expected worst-case and $\mathcal{O}(\log\log|U|)$ worst-case time.

```
lca(T_z, T_Δ, q, w):
  for i in 1,...
    h_i = 2^(2^i)
    d_i = w - max(0, h_i)

    p_i = q_0 ... q_{d_i - 1}

    if T_Δ[p_i] ≠ ⊥:
      lca = lca*(T_z, T_Δ, q, p_i, w)
      return lca
```

The following lemma shows that the Algorithm 39 can not always find the lowest common ancestor in $\mathcal{O}(\log\log\Delta)$ time and that there are cases where a lot of different queries have a bad runtime.

Lemma 8.6 *For each w, there exists a set $S \subseteq U$ of elements where the Algorithm 39 needs $\Omega(\log\log|U|)$ time for $q_1, \ldots, q_{\Omega(\sqrt{|U|})}$ different queries, which have all constant distance $\Delta = \mathcal{O}(1)$ to their successor. The number of loop iterations are $k = \Omega(\log\log w)$.*

Proof. Note that $2^{w/2} = \sqrt{|U|}$. Let

$$P_i = \left[(i-1)\cdot 2^{w/2},\ i\cdot 2^{w/2}\right), \qquad i = 1, \ldots, 2^{w/2}$$

be a partition of U, $q_i = \max P_i$ for $i = 1, \ldots, 2^{w/2} - 1$ be the queries and $s_i = \min P_{i+1}$ for $i = 1, \ldots, 2^{w/2} - 1$ be the elements of S. Clearly, $\Delta_i = |q_i - s_i| = 1$. But, the length of the lowest common ancestor $|\operatorname{lca}_i|$ is $w/2 - 1$. Thus, the number of remaining bits are $w - |\operatorname{lca}_i| = w/2 + 1$. Thus, the binary search will take $\Omega(\log w) = \Omega(\log\log|U|)$ time and the first time the doubly exponential search covers with $h_k = 2^{2^k}$ more than $w/2$ remaining bits is at $k \geq \log\log(w/2)$ □

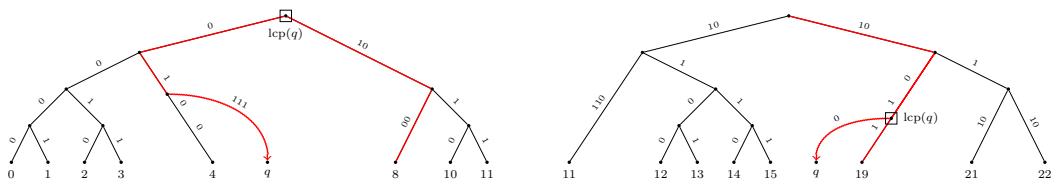
The lemma also shows that the update time may take $\Omega(\log\log w)$ time, because at least $\Omega(\log\log w)$ prefixes of q need to be inserted into the hash map T_Δ. But, the authors of [Bos+13] claim, that they support $\mathcal{O}(\log\log\Delta)$ expected update time. Their main idea is, that when having two randomly chosen elements x and y, their lowest common ancestor will be at the expected height $\mathcal{O}(\log|x-y|)$.

They accomplish this by adding a randomly chosen constant r to each query q. This results into a new trie structure which is similar to a trie built upon random elements. See Figure 12 for an example. The lemma is as follows:

Lemma 8.7 ([Bos+13], Lemma 4) *After performing a random shift by r of U, the expected height of the lowest common ancestor of two elements x and y is $\mathcal{O}(\log|x-y|)$.*

Corollary 8.8 *The lowest common ancestor of q can be found in $\mathcal{O}(\log\log\Delta)$ expected worst case time. The runtime is expected over the random choice of r.*

Proof. Set $x = q$, $y = q^+$ and let $\Delta = |q - q^+|$. Do the doubly exponential search on the prefixes of q (without checking $p_i + 1$) to find h_k. After that resume the lowest common ancestor search



(a) $q = 7 = 0111$ and $q^+ = 8 = 1000$ are near $|q - q^+| = 1$, but the longest common prefix is ε.

(b) $q = 7 + r = 10010$ and $q^+ = 8 + r = 10011$ are still near $|q + r - (q^+ + r)| = 1$, but the longest common prefix is 1001 and reflects now the distance between q and q^+.

Figure 12: Δ-fast Trie - Before and after a shift by a randomly chosen $r = 11$.

on the remaining h_k bits. Since the remaining bits h_k are expected to be $\mathcal{O}(\log \Delta)$ and the number of loop iterations k are expected to be $\mathcal{O}(\log \log \log \Delta)$, the runtime is expected to be $k + \log h_k = \mathcal{O}(\log \log \Delta)$. The runtime itself is worst case, since the random choice of r will be done in the initialization of the data structure. □

8.5.2. Managing the Min- and Maximal Leaves of the Subtrees

In the dynamic case of the predecessor problem the management and the runtime of the minimal and maximal leaves of a given subtree plays a important role. In the static case it suffices to have a pointer from a node to its minimal and maximal leaf. But in the dynamic case this is insufficient, because an update can cause $\Omega(\log |U|)$ pointer updates. The Δ-fast trie is based on the Z-fast trie which naturally had the same issue. The Z-fast trie solved the problem with an update and lookup time of $\mathcal{O}(\log \log |U|)$. But, the Δ-fast trie is aiming for $\mathcal{O}(\log \log \Delta)$ update and lookup time. So we have to adapt the currently employed strategy.

The Z-fast trie uses the fact that a minimal leaf (or maximal leaf) is shared by at most w nodes and that all those nodes form a subpath of a leaf-to-root path in the trie (Section 6.5). Therefore, using a height balanced binary search tree to manage the w nodes yields an $\mathcal{O}(\log w)$ update and query time. Whereas in the Δ-fast trie the update and query time depends on the height h_i (i.e, the remaining bits) of the query. Therefore we partition the heights $\{0, 1, \ldots, w\}$ of a subpath into the sets

$$T_{-1} = \{0\}, \qquad T_i = [2^i, 2^{i+1}) \quad \text{for } i = 0, \ldots, \log w - 1 \quad \text{and} \qquad T_{\log w} = \{w\}.$$

See Figure 13 for an illustration of the following. Each of the sets are managed by a height balanced binary tree and all roots of those trees are linked together. Clearly, the height of the i-th binary search tree is $\log |T_i| = \mathcal{O}(i)$. 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 $\mathcal{O}(i)$ time by following the pointers to the root of T_i and the pointers down to T_{-1}.

If a node v has $h_k = \mathcal{O}(\log \Delta)$ remaining bits, the node is within the tree $T_{\lfloor \log h_k \rfloor}$. Thus, it takes $\mathcal{O}(\log h_k) = \mathcal{O}(\log \log \Delta)$ time to find the minimum.

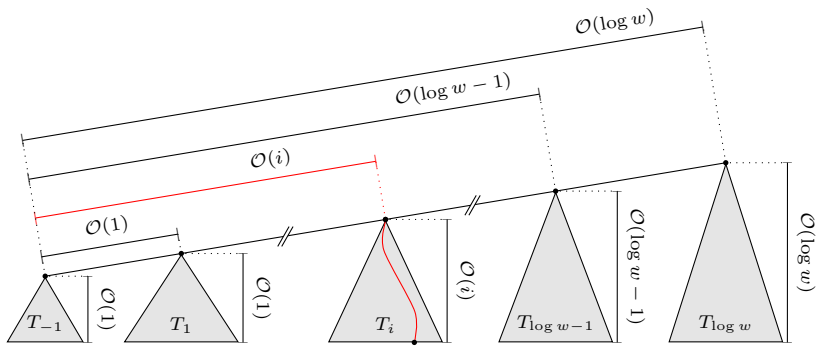


Figure 13: Δ-fast Trie - A preferred path decomposed in subpaths managed by the trees T_i for $i = -1, \ldots, \log w$. The minimum or maximum can be found in T_{-1}.

8.5.3. Update the Min- and Maximal Leaves of the Subtrees

The update of the min- and maximal leaves of the subtrees requires to cut a subpath at height h and to stitch two paths p_1, p_2[1] together. Those operations are used in the tango tree to change the preferred children. See Section 6.5 for the full details.

Cut a Subpath into two Paths

When cutting a path at height h into two paths p_1 and p_2, first cut the links between the trees $T_{\lfloor \log h \rfloor - 1}$, $T_{\lfloor \log h \rfloor}$ and $T_{\lfloor \log h \rfloor + 1}$. After, that cut the tree $T_{\lfloor \log h \rfloor}$ at height h into $T'_{\lfloor \log h \rfloor}$ (all nodes with height smaller than h) and $T''_{\lfloor \log h \rfloor}$. Then link the roots of the trees $T_{\lfloor \log h \rfloor - 1}$ and $T'_{\lfloor \log h \rfloor}$ as well as $T''_{\lfloor \log h \rfloor}$ and $T_{\lfloor \log h \rfloor + 1}$ back together. Clearly, those trees represent the path p_1 (all nodes have height smaller h) and p_2. We have seen in Section 6, that cutting a tree with height h takes $\mathcal{O}(\log h)$ time. Thus, the modified cut operation takes $\mathcal{O}(\log \log \Delta)$ time.

Join two Subpaths together

The first path p_1 contains the trees T'_i for $i = -1, \ldots, \lfloor \log h \rfloor$ and the second path p_2 contains the trees T''_i for $i = \lfloor \log h \rfloor, \ldots, w$. So joining them is simple, because we only have to deal with the intersection of the trees at height $\lfloor \log h \rfloor$. Thus, cut the roots from $T'_{\lfloor \log h \rfloor - 1}$ and $T'_{\lfloor \log h \rfloor}$, as well as the roots from $T''_{\lfloor \log h \rfloor}$ and $T''_{\lfloor \log h \rfloor + 1}$. Next, join $T'_{\lfloor \log h \rfloor}$ and $T''_{\lfloor \log h \rfloor}$ together and name the resulting tree $T_{\lfloor \log h \rfloor}$. Finally, link the roots of the trees $T'_{\lfloor \log h \rfloor - 1}$, $T_{\lfloor \log h \rfloor}$ and $T''_{\lfloor \log h \rfloor + 1}$ back together. The runtime for joining at height h takes $\mathcal{O}(\log h)$ time. Therefore, the modified join operation takes $\mathcal{O}(\log \log \Delta)$ time.

8.5.4. Updating an Element

We know from the Lemma 8.7, that the lowest common ancestor has expected height $h_k = \mathcal{O}(\log \Delta)$.

Lemma 8.9 *Inserting an element q into a Δ-fast trie takes $\mathcal{O}(\log \log \Delta)$ expected time over the random choice of r.*

Proof. Inserting an element into the trie will split an edge (u, v) in the trie into two edges (u, b) and (b, v) and will result in two newly added nodes. One is a branch node, while the other is a leaf. The branch node is the lowest common ancestor and has expected height $h_k = \mathcal{O}(\log \Delta)$. So, finding it will take $\mathcal{O}(\log \log \Delta)$ expected time (Corollary 8.8).

Now let us consider the update time of the T_Δ hash map. Remember that T_Δ stores for each node its lowest branch node. That means that all prefixes on the edge (b, v) which are stored in the hash map T_Δ need to be updated. And prefixes at certain depths which are on the edge (b, q) need to be inserted. 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 $\mathcal{O}(\log \log \log \Delta)$ 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 $d_i = w - h_i$, $|b| = w - \mathcal{O}(\log \Delta)$ and $h_i = 2^{2^i}$, $c \log \Delta < 2^{2^i}$ when $i > \log \log (c \log \Delta)$. Thus, $i = \Theta(\log \log \log \Delta)$.

[1] p_1 has nodes with height smaller than h, whereas p_2 has nodes with height larger than h.

After that, the min- and maximal elements have to be updated. This might issue an preferred child change at height $h_k = \mathcal{O}(\log \Delta)$, which – as we have seen – takes with the cut and join operation $\mathcal{O}(\log h_k) = \mathcal{O}(\log \log \Delta)$ time. □

Lemma 8.10 *Deleting an element q from a Δ-fast trie takes $\mathcal{O}(\log \log \Delta)$ expected time over the random choice of r.*

Proof. It is similar to the insert case. Find the lowest common ancestor in $\mathcal{O}(\log \log \Delta)$ time, then delete the node of q and the branch node. After that update the nodes below the lowest common ancestor in the hash map T_Δ in $\mathcal{O}(\log \log \log \Delta)$ time. And finally, update the min- and maximal elements in $\mathcal{O}(\log \log \Delta)$ time. □

8.5.5. Final Result

Theorem 8.2 *After performing a random shift by r of U, the Δ-fast trie solves the dynamic predecessor problem, where the static operations take $\mathcal{O}(\log \log \Delta)$ time and the update operations need $\mathcal{O}(\log \log \Delta)$ expected worst-case time for $\Delta = \min |q - q^+|, |q - q^-|$. The space usage is $\mathcal{O}(n)$ words.*

Proof. We have seen that the update operations, as well as finding the lowest common ancestor can be done in $\mathcal{O}(\log \log \Delta)$ expected time (Lemmas 8.9 and 8.10 and Corollary 8.8). The predecessor, successor and normal search needs $\mathcal{O}(\log \log \Delta)$ time (Algorithms 37 and 38). The space usage is linear by Lemma 8.5. □

8.6. Remarks

Interestingly, a query time of $\mathcal{O}(\log \log \Delta)$ enables faster queries for elements that are "near" in S. This offers a smooth transition between the worlds of hash maps, which can answer in $\mathcal{O}(1)$ time and van-Emde-Boas based solutions, which answer in $\mathcal{O}(\log \log |U|)$ time.

The same result, but for the pointer machine, was already given in the Eighties by Johnson [Joh81]. Van Emde Boas wrote in his invited paper [Boa13] about the results of Johnson:

> "As observed by the author: there seems to be no way to obtain this improvement when the recursive cluster galaxy decomposition approach is used."

The cluster galaxy decomposition is the concept used in the van-Emde-Boas Tree, as seen in Section 3. For this concept it might be impossible, but with the trie based solution it is. But it seems, that van Emde Boas did not know of the result from Bose *et al* [Bos+13], since both papers came out in 2013.

9. Conclusion

9.1. Summary

In this thesis, we studied the predecessor problem for the w-bit word RAM. We restricted us to van-Emde-Boas tree based solutions which use linear space and tried to use the $\boldsymbol{AC^0}$ instruction set as much as possible. We studied the following different approaches. The first approach was the van-Emde-Boas tree which is based on a galaxy cluster decomposition. The second approach was the X-fast trie which does a binary search on the word size w by using a hash map. The Y-fast trie was the third approach where we used a technique called indirection to improve upon the data structure X-fast trie.

We gave a complete new technique for the dynamic Z-fast trie by using tango trees. This approach was the fourth. We noticed that every binary search on a fixed w-bit word and a given fixed trie structure will have a predetermined search order. By carefully choosing an associated key α_e for an edge e which exploits this fact, it became possible to use a compact binary trie directly. We earlier showed that a compact binary trie has only linear complexity. Thus, the data structure needed by design only linear space.

The fifth approach was the μ-fast trie. This data structure used the idea of compressing multiple words into one and a neat trick to replace the membership query by a retrieval only query. We used a dynamic perfect retrieval only hash table to accomplish the word compression. The seventh approach was the Δ-fast trie. We combined the Z-fast trie and μ-fast trie to solve an open problem.We gave pseudo code for almost every operation and analyzed the data structures in detail.

9.2. Further Work

The given data structures are optimal for $w = \Theta(\log n)$. Thus, further improvement is not possible. It would be interesting how all those data structures would perform in a high optimized implementation against current standard implementations of IP-packet routing. It would also be interesting, if there is a data structure where each predecessor operation works without randomization, but within the same time and space bounds.

Acknowledgements

First, I want to thank my family for their support and love. I want to especially thank my two sons who were sometimes really demanding, but most times a pleasant diversion. Of course I will not leave out my live-in girlfriend which is my pillar and my dear one.

Second, I want to thank Wolfgang Mulzer for his support and inspiration. I missed almost no lecture since I came across him as an undergraduate. He motivated me to learn a lot about algorithms, data structures and other interesting stuff from the field of theoretical computer science. I also want to thank the working group of theoretical informatics for their interesting lectures, challenging exercises and sometimes even more demanding exams.

Last but not least, I want to thank my colleagues, my exercise partners, my fellow teaching assistants and my dear friends who accompanied me through my years of diligent study.

References

[And+96] A. Andersson, P.B. Miltersen, S. Riis, and M. Thorup. "Static Dictionaries on AC^0 RAMS: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient". In: *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on.* Oct. 1996, pp. 441–450. DOI: 10.1109/SFCS.1996.548503.

[And+98] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. "Sorting in Linear Time?" In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 74–93. ISSN: 0022-0000. DOI: 10.1006/jcss.1998.1580.

[BBV10] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. "Dynamic Z-Fast Tries". In: *String Processing and Information Retrieval.* Ed. by Edgar Chavez and Stefano Lonardi. Vol. 6393. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 159–172. ISBN: 978-3-642-16320-3.

[BCH86] Paul W Beame, Stephen A Cook, and H James Hoover. "Log Depth Circuits for Division and Related Problems". In: *SIAM J. Comput.* 15.4 (Nov. 1986), pp. 994–1003. ISSN: 0097-5397. DOI: 10.1137/0215070.

[Bel+09] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. "Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses". In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009.* 2009, pp. 785–794.

[BF02] Paul Beame and Faith E. Fich. "Optimal Bounds for the Predecessor Problem and Related Problems". In: *Journal of Computer and System Sciences* 65.1 (2002), pp. 38–72. ISSN: 0022-0000. DOI: 10.1006/jcss.2002.1822.

[BF99] Paul Beame and Faith E. Fich. "Optimal Bounds for the Predecessor Problem". In: *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing.* STOC '99. New York, NY, USA: ACM, 1999, pp. 295–304. ISBN: 1-58113-067-8. DOI: 10.1145/301250.301323.

[BH89] Paul Beame and Johan Hastad. "Optimal Bounds for Decision Problems on the CRCW PRAM". In: *J. ACM* 36.3 (July 1989), pp. 643–670. ISSN: 0004-5411. DOI: 10.1145/65950.65958.

[BKZ76] P. van Emde Boas, R. Kaas, and E. Zijlstra. "Design and implementation of an efficient priority queue". In: *Mathematical systems theory* 10.1 (1976), pp. 99–127. ISSN: 0025-5661. DOI: 10.1007/BF01683268. URL: 10.1007/BF01683268.

[Boa13] Peter Van Emde Boas. "Thirty nine years of stratified trees(Invited talk)". In: *2nd International Symposium on Computing In Informatics And Mathematics (ISCIM 13).* Ed. by Loukas Georgiadis Ilir Capuni and Oguz Altun. Vol. 1. Rinas–Tirana, Albania: Epoka University, 2013, pp. 1–14. ISBN: 978 - 9928-135-10-0.

[Boa77] P. van Emde Boas. "Preserving order in a forest in less than logarithmic time and linear space". In: *Information Processing Letters* 6.3 (1977), pp. 80–82. ISSN: 0020-0190. DOI: 10.1016/0020-0190(77)90031-X.

[Bos+13] Prosenjit Bose, Karim Douïeb, Vida Dujmović, John Howat, and Pat Morin. "Fast Local Searches and Updates in Bounded Universes". In: *Comput. Geom. Theory Appl.* 46.2 (Feb. 2013), pp. 181–189. ISSN: 0925-7721. DOI: 10.1016/j.comgeo.2012.01.002.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0-262-03384-4 978-0-262-03384-8.

[CW79] J. Lawrence Carter and Mark N. Wegman. "Universal classes of hash functions". In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 143–154. ISSN: 0022-0000. DOI: 10.1016/0022-0000(79)90044-8.

[Dem+06] Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. "De Dictionariis Dynamicis Pauco Spatio Utentibus". In: *LATIN 2006: Theoretical Informatics*. Ed. by JoséR. Correa, Alejandro Hevia, and Marcos Kiwi. Vol. 3887. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 349–361. ISBN: 978-3-540-32755-4.

[Dem+07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. "Dynamic Optimality – Almost". In: *SIAM Journal on Computing* 37.1 (2007). See also FOCS'04, pp. 240–251.

[Dem12] Erik Demaine. *Advanced Data Structures (6.851)*. 2012. URL: http://courses.csail.mit.edu/6.851/spring12/ (visited on 09/02/2015).

[FW93] Michael L. Fredman and Dan E. Willard. "Surpassing the information theoretic bound with fusion trees". In: *Journal of Computer and System Sciences* 47.3 (1993), pp. 424–436. ISSN: 0022-0000. DOI: 10.1016/0022-0000(93)90040-4.

[GBT84] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. "Scaling and Related Techniques for Geometry Problems". In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. STOC '84. New York, NY, USA: ACM, 1984, pp. 135–143. ISBN: 0-89791-133-4. DOI: 10.1145/800057.808675.

[Han02] Yijie Han. "Deterministic Sorting in O(Nlog Log N) Time and Linear Space". In: *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*. STOC '02. New York, NY, USA: ACM, 2002, pp. 602–608. ISBN: 1-58113-495-9. DOI: 10.1145/509907.509993.

[HT02] Yijie Han and Mikkel Thorup. "Integer Sorting in 0(N Sqrt (Log Log N)) Expected Time and Linear Space". In: *Proceedings of the 43rd Symposium on Foundations of Computer Science*. FOCS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 135–144. ISBN: 0-7695-1822-2.

[Joh81] DonaldB. Johnson. "A priority queue in which initialization and queue operations takeO(loglogD) time". In: *Mathematical systems theory* 15.1 (1981), pp. 295–309. ISSN: 0025-5661. DOI: 10.1007/BF01786986.

[MN90] Kurt Mehlhorn and Stefan Näher. "Bounded Ordered Dictionaries in O(log log N) Time and O(n) Space". In: *Inf. Process. Lett.* 35.4 (1990), pp. 183–189. DOI: 10.1016/0020-0190(90)90022-P.

[MNA88] Kurt Mehlhorn, Stefan Näher, and Helmut Alt. "A Lower Bound on the Complexity of the Union-Split-Find Problem". In: *SIAM Journal on Computing* 17.6 (1988), pp. 1093–1102. DOI: 10.1137/0217070.

[Mul09] Wolfgang Mulzer. "A Note on Predecessor Searching in the Pointer Machine Model". In: *Inf. Process. Lett.* 109.13 (June 2009), pp. 726–729. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2009.03.003.

[Păt10a] Mihai Pătraşcu. *Retrieval-Only Dictionaries*. WebDiarios de Motocicleta. Sept. 27, 2010. URL: http://infoweekly.blogspot.de/2010/09/retrieval-only-dictionaries.html (visited on 03/23/2015).

[Păt10b] Mihai Pătraşcu. *Van Emde Boas and its space complexity.* WebDiarios de Motocicleta. Sept. 19, 2010. URL: `http://infoweekly.blogspot.de/2010/09/van-emde-boas-and-its-space-complexity.html` (visited on 03/23/2015).

[Păt10c] Mihai Pătraşcu. *vEB Space: Method 4.* WebDiarios de Motocicleta. Sept. 21, 2010. URL: `http://infoweekly.blogspot.de/2010/09/veb-space-method-4.html` (visited on 03/23/2015).

[PT07] Mihai Pătraşcu and Mikkel Thorup. "Randomization Does Not Help Searching Predecessors". In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA '07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 555–564. ISBN: 978-0-89871-624-5.

[PT10] Mihai Pătraşcu and Mikkel Thorup. "The Power of Simple Tabulation Hashing". In: *CoRR* abs/1011.5200 (2010).

[PT14] Mihai Pătraşcu and Mikkel Thorup. "Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search". In: *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014.* Oct. 2014, pp. 166–175. DOI: `10.1109/FOCS.2014.26`.

[Pug90] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Commun. ACM* 33.6 (June 1990), pp. 668–676. ISSN: 0001-0782. DOI: `10.1145/78973.78977`.

[Ruž09] Milan Ružić. "Making Deterministic Signatures Quickly". In: *ACM Trans. Algorithms* 5.3 (July 2009), 26:1–26:26. ISSN: 1549-6325. DOI: `10.1145/1541885.1541887`.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation.* 1st. International Thomson Publishing, 1996. ISBN: 0-534-94728-X.

[Smo87] R. Smolensky. "Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity". In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing.* STOC '87. New York, NY, USA: ACM, 1987, pp. 77–82. ISBN: 0-89791-221-7. DOI: `10.1145/28395.28404`.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-adjusting binary search trees". In: *J. ACM* 32.3 (July 1985), pp. 652–686. ISSN: 0004-5411. DOI: `10.1145/3828.3835`.

[Tho03] Mikkel Thorup. "On AC0 Implementations of Fusion Trees and Atomic Heaps". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA '03. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 699–707. ISBN: 0-89871-538-5.

[Tho07] Mikkel Thorup. "Equivalence Between Priority Queues and Sorting". In: *J. ACM* 54.6 (Dec. 2007). ISSN: 0004-5411. DOI: `10.1145/1314690.1314692`.

[TZ12] Mikkel Thorup and Yin Zhang. "Tabulation-Based 5-Independent Hashing with Applications to Linear Probing and Second Moment Estimation". In: *SIAM Journal on Computing* 41.2 (2012), pp. 293–331. DOI: `10.1137/100800774`.

[Wil83] Dan E. Willard. "Log-logarithmic worst-case range queries are possible in space $\Theta(N)$". In: *Information Processing Letters* 17.2 (1983), pp. 81–84. ISSN: 0020-0190. DOI: `10.1016/0020-0190(83)90075-3`.

List of Figures

List of Tables

List of Algorithms

Appendices

A. Original Blog Posts of Mihai Pătrașcu

A.1. Van Emde Boas and its space complexity [Pät10b]

Here is a quick review of vEB if you don't know it. Experienced readers can skip ahead.

The predecessor problem is to support a set S of $|S| = n$ integers from the universe $\{1, ..., u\}$ and answer:

$$predecessor(q) = \max\{x \in S | x \leq q\}$$

The vEB data structure can answer queries in $\mathcal{O}(\lg \lg u)$ time, which is significantly faster than binary search for moderate universes.

The first idea is to divide the universe into $\sqrt{u}$ segments of size $\sqrt{u}$. Let $hi(x) = \lfloor x/\sqrt{u} \rfloor$ be the segment containing x, and $lo(x) = xmod\sqrt{u}$ be the location of x within its segment. The data structure has the following components:

- a hash table H storing hi(x) for all $x \in S$.
- a top structure solving predecessor search among $\{hi(x) | x \in S\}$. This is the same as the original data structure, i.e. use recursion.
- for each element $\alpha \in H$, a recursive bottom structure solving predecessor search inside the α segment, i.e. among the keys $\{lo(x) | x \in S \wedge hi(x) = \alpha\}$.

The query algorithm first checks if $hi(q) \in H$. If so, all the action is in q's segment, so you recurse in the appropriate bottom structure. (You either find its predecessor there, or in the special case when q is less than the minimum in that segment, find the successor and follow a pointer in a doubly linked list.)

If q's segment is empty, all the action is at the segment level, and q's predecessor is the max in the preceding non-empty segment. So you recurse in the top structure.

In one step, the universe shrinks from u to $\sqrt{u}$, i.e. $\lg u$ shrinks to $\frac{1}{2} \lg u$. Thus, in $\mathcal{O}(\lg \lg u)$ steps the problem is solved.

So what is the space of this data structure? As described above, each key appears in the hash table, and in 2 recursive data structures. So the space per key obeys the recursion $S(u) = 1 + 2S(\sqrt{u})$. Taking logs: $S'(\lg u) = 1 + 2S'(\frac{1}{2} \lg u)$, so the space is $\mathcal{O}(\lg u)$ per key.

How can we reduce this to space $\mathcal{O}(n)$? Here are 3 very different ways:

A.1.1. Brutal bucketing.

Group elements into buckets of $\mathcal{O}(\lg u)$ consecutive elements. From each bucket, we insert the min into a vEB data structure. Once we find a predecessor in the vEB structure, we know the bucket where we must search for the real predecessor. We can use binary search inside the bucket, taking time $\mathcal{O}(\lg \lg u)$. The space is $(n/\lg u) \cdot \lg u = \mathcal{O}(n)$.

A.1.2. Better analysis.

In fact, the data structure from above does take $\mathcal{O}(n)$ space if you analyze it better! For each segment, we need to remember the max inside the segment in the hash table, since a query in the top structure must translate the segment number into the real predecessor. But then there's no point in putting the max in the bottom structure: once the query accesses the hash table, it can simply compare with the max in $\mathcal{O}(1)$ time. (If the query is higher than the max in its segment, the max is the predecessor.)

In other words, every key is stored recursively in just one data structure: either the top structure (for each segment max) or the bottom structure (for all other keys). This means there are $\mathcal{O}(\lg \lg u)$ copies of each element, so space $\mathcal{O}(n \lg \lg u)$.

But note that copies get geometrically cheaper! At the first level, keys are $\lg u$ bits. At the second level, they are only $\frac{1}{2} \lg u$ bits; etc. Thus, the cost per key, in bits, is a geometric series, which is bounded by $\mathcal{O}(\lg u)$. In other words, the cost is only $\mathcal{O}(1)$ words per key. (You may ask: even if the cost of keys halves every time, what about the cost of pointers, counters, etc? The cost of a pointer is $\mathcal{O}(\lg n)$ bits, and $n \leq u$ in any recursive data structure.)

A.1.3. Be slick.

Here's a trickier variation due to Rasmus Pagh[1]. Consider the trie representing the set of keys (a trie is a perfect binary tree of depth $\lg u$ in which each key is a root-to-leaf path). The subtree induced by the keys has $n - 1$ branching nodes, connected by $2n - 1$ unbranching paths. It suffices to find the lowest branching node above the query. (If each branching node stores a pointer to his children, and the min and max values in its subtree, we can find the predecessor with constant work after knowing the lowest branching node.)

We can afford space $\mathcal{O}(1)$ per path. The data structure stores:

- a top structure, with all paths that begin and end above height $\frac{1}{2} \lg u$.
- a hash table H with the nodes at depth $\frac{1}{2} \lg u$ of every path crossing this depth.
- for each $\alpha \in H$, a bottom structure with all paths starting below depth $\frac{1}{2} \lg u$ which have α as prefix.

Observe that each path is stored in exactly one place, so the space is linear. But why can we query for the lowest branching node above some key? As the query proceeds, we keep a pointer p to the lowest branching node found so far. Initially p is the root. Here is the query algorithm:

- if p is below depth $\frac{1}{2} \lg u$, recurse in the appropriate bottom structure. (We have no work to do on this level.)
- look in H for the node above the query at depth $\frac{1}{2} \lg u$. If not found, recurse in the top structure. If found, let p be the bottom node of the path crossing depth $\frac{1}{2} \lg u$ which we just found in the hash table. Recurse to the appropriate bottom structure.

The main point is that a path is only relevant once, at the highest level of the recursion where the path crosses the middle line. At lower levels the path cannot be queried, since if you're on the path you already have a pointer to the node at the bottom of the path!

[1] Mihai: "A correction: Method 3 is originally due to Milan Ružić, from [Ruž09]"

A.2. vEB Space: Method 4 [Pă t10c]

In this post, I will describe a 4th method. You'd be excused for asking what the point is, so let me quickly mention that this technique has a great application (1D range reporting, which I will discuss in another post) and it introduces a nice tools you should know.

Here is a particularly simple variant of vEB, introduced by Willard as the "y-fast tree". Remember from the last post that the trie representing the set has $n-1$ branching nodes connected by $2n-1$ "active" paths; if we know the lowest branching ancestor of the query, we can find the predecessor in constant time. Willard's approach is to store a hash table with all $\mathcal{O}(n \lg u)$ active nodes in the trie; for each node, we store a pointer to its lowest branching ancestor. Then, we can binary search for the height of the lowest active ancestor of the query, and follow a pointer to the lowest branching node above. As the trie height is $\mathcal{O}(\lg u)$, this search takes $\mathcal{O}(\lg \lg u)$ look-ups in the hash table.

Of course, we can reduce the space from $\mathcal{O}(n \lg u)$ to $\mathcal{O}(n)$ by bucketing. But let's try something else. We could break the binary search into two phases:

1. Find v, the lowest active ancestor of the query at some depth of the form $i \cdot \sqrt{\lg u}$ (binary search on i). Say v is on the path $u \to w$ (where u, w are branching nodes). If w is not an ancestor of the query, return u.
2. Otherwise, the lowest branching ancestor of the query is found at some depth in $[i \cdot \sqrt{\lg u},\ (i+1)\sqrt{\lg u}]$. Binary search to find the lowest active ancestor in this range, and follow a pointer to the lowest active ancestor.

With this modification, we only need to store $\mathcal{O}(n\sqrt{\lg u})$ active nodes in the hash table! To support step 1., we need active nodes at depths $i \cdot \sqrt{\lg u}$. To support step 2., we need active nodes whose lowest branching ancestor is only $\leq \sqrt{\lg u}$ levels above. All other active nodes can be ignored.

You could bring the space down to $\mathcal{O}(n \lg^{\epsilon} u)$ by breaking the search into more segments. But to bring the space down to linear, we use heavier machinery:

Retrieval-only dictionaries. Say we want a dictionary ("hash table") that stores a set of n keys from the universe $[u]$, where each key has k bits of associated data. The dictionary supports two operations:

- membership: is x in the set?
- retrieval: assuming x is in the set, return $data[x]$.

If we want to support both operations, the smallest space we can hope for is $\log(\binom{u}{n}) + nk \approx n(\lg u + k)$ bits: the data structure needs to encode the set itself, and the data.

Somewhat counterintuitively, dictionaries that only support retrieval (without membership queries) are in fact useful. (The definition of such a dictionary is that $retrieve(x)$ may return garbage if x is not in the set.)

Retrieval-only dictionaries can be implemented using only $\mathcal{O}(nk)$ bits. I will describe this in the next post, but I hope it is believable enough.

When is a retrieval-only dictionary helpful? When we can verify answers in some other way. Remember the data structure with space $\mathcal{O}(n\sqrt{\lg u})$ from above. We will store branching nodes

in a real hash table (there are only $n-1$ of them). But observe the following about the $\mathcal{O}(n\sqrt{\lg u})$ active nodes that we store:

1. We only need $k = \mathcal{O}(\lg \lg u)$ bits of associated data. Instead of storing a pointer to the lowest branching ancestor, we can just store the height difference (a number between 1 and $\lg u$). This is effectively a pointer: we can compute the branching ancestor by zeroing out so many bits of the node.
2. We only need to store them in a retrieval-only dictionary. Say we query some node v and find a height difference δ to the lowest branching ancestor. We can verify whether v was real by looking up the δ-levels-up ancestor of v in the hash table of branching nodes, and checking that v lies on one of the two paths descending from this branching node.

Therefore, the dictionary of active nodes only requires $\mathcal{O}(n\sqrt{\lg u}\cdot\lg \lg u)$ bits, which is $\mathcal{O}(n)$ words of space! This superlinear number of nodes take negligible space compared to the branching nodes.

A.3. Retrieval-Only Dictionaries [Pät10a]

We saw two cool applications of dictionaries without membership; now it's time to construct them. Remember that we are given a set S, where each element $x \in S$ has some associated $data[x]$, a k-bit value. We want a data structure of $\mathcal{O}(nk)$ bits which retrieves $data[x]$ for any $x \in S$ and may return garbage when queried for $x \notin S$.

A conceptually simple solution is the "Bloomier filter" of [Chazelle, Kilian, Rubinfeld, Tal SODA'04]. This is based on the power of two choices, so you should first go back and review my old post giving an encoding analysis of cuckoo hashing.

Standard cuckoo hashing has two arrays $A[1..2n]$, $B[1..2n]$ storing keys, and places a key either at $A[h(x)]$ or $B[g(x)]$. Instead of this, our arrays A and B will store k-bit values ($\mathcal{O}(nk)$ bits in total), and the query retrieve-data(x) will return $A[h(x)]$ xor $B[g(x)]$.

The question is whether we can set up the values in A and B such that any query $x \in S$ returns $data[x]$ correctly. This is a question about the feasibility of a linear system with n equations (one per key) and $4n$ variables (one per array entry).

Consider a connected component in the bipartite graph induced by cuckoo hashing. If this component is acyclic, we can fix A and B easily. Take an arbitrary node and make it "zero"; then explore the tree by DFS (or BFS). Each new node (an entry in A or B) has a forced value, since the edge advancing to it must return some $data[x]$ and the parent node has been fixed already. As the component is acyclic, there is only one constraint on every new node, so there are no conflicts.

On the other hand, if a component has a cycle, we are out of luck. Remark that if we xor all cycle nodes by some δ, the answers are unchanged, since the δ's cancel out on each edge. So a cycle of length k must output k independent data values, but has only $k-1$ degrees of freedom.

Fortunately, one can prove the following about the cuckoo hashing graph:

- the graph is acyclic with some constant probability. Thus, the construction algorithm can rehash until it finds an acyclic graph, taking $\mathcal{O}(n)$ time in expectation.
- the total length of all cycles is $\mathcal{O}(\log n)$ with high probability. Thus we can make the graph acyclic by storing $\mathcal{O}(\log n)$ special elements in a stash. This gives construction time $\mathcal{O}(n)$

w.h.p., but the query algorithm is slightly more complicated (for instance, it can handle the stash by a small hash table on the side).

These statements fall out naturally from the encoding analysis of cuckoo hashing. A cycle of length k allows a saving of roughly k bits in the encoding: we can write the k keys on the cycle ($k \log n$ bits) plus the k hash codes ($k \log(2n)$ bits) instead of $2k$ hash codes ($2k \log(2n)$ bits).

Further remarks. Above, I ignored the space to store the hash functions h and g. You have to believe me that there exist families of hash functions representable in $\mathcal{O}(n\epsilon)$ space, which can be evaluated in constant time, and make cuckoo hashing work.

A very interesting goal is to obtain retrieval dictionaries with close to kn bits. As far as I know, the state of the art is given by [Pagh-Dietzfelbinger ICALP'08] and [Porat].