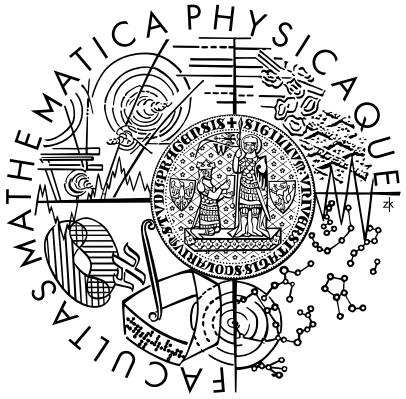
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Vladimír Čunát

Datové struktury pro různá rozdělení dat
Data structures for various distributions of data

Department of Theoretical Computer Science and Mathematical Logic
Supervisor: doc. RNDr. Václav Koubek, DrSc.
Branch: Theoretical Computer Science, Algorithms and Complexity

I would like to thank to my supervisor, doc. RNDr. Václav Koubek, DrSc., for his effort spent on consultations and on reviewing my work. His guidance, many helpful suggestions and questions have helped a lot to improve this thesis. I also thank to my family and friends for continuous support during my whole studies.

Prohlašuji, že jsem tuto diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

I hereby declare that I have written this thesis on my own and using exclusively the cited sources. I authorize Charles University to lend this document to other institutions and individuals.

In Prague on February 23, 2011 Vladimír Čunát
(*this is a slightly updated version*)

Contents

Název práce: Datové struktury pro různá rozdělení dat
Autor: Vladimír Čunát
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: doc. RNDr. Václav Koubek, DrSc.
E-mail vedoucího: Vaclav.Koubek@mff.cuni.cz

Abstrakt: Práce se zabývá studiem problému předchůdce, kde datová struktura udržuje dynamickou uspořádanou množinu klíčů. Kromě přehledu nejdůležitějších publikovaných výsledků ukazujeme podrobný popis konkrétní možnosti, jak lze docílit pravděpodobnostní úpravy van Emde Boasovy struktury. Tato úprava snižuje paměťovou náročnost na optimum, akorát stejné časové složitosti $\Theta(\log\log N)$ již není dosahováno v nejhorším případě, ale v amortizovaném očekávaném případě.

Nejlepší očekávaná amortizovaná složitost dosahovaná na třídě $(s^\alpha, s^{1-\delta})$-hladkých distribucí je rovna $\mathcal{O}(\log\log n)$. Kombinací známých technik dostáváme novou datovou strukturu, která dosahuje stejné složitosti, ale na širší třídě distribucí než bylo doposud možné. Navíc lze jako podstrukturu využít optimální amortizované řešení problému navržené Beamem a Fichem, což zaručí omezení amortizované složitosti nové struktury na asymptoticky optimální hodnotu rovnou $\Theta\left(\sqrt{\log n / \log\log n}\right)$.

Klíčová slova: dynamické datové struktury, problém předchůdce, složitost v očekávaném případě

Title: Data structures for various distributions of data
Author: Vladimír Čunát
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: doc. RNDr. Václav Koubek, DrSc.
Supervisor's e-mail address: Vaclav.Koubek@mff.cuni.cz

Abstract: In this thesis we study the predecessor problem, which consists of maintaining a dynamic ordered set of keys. After a survey of the most important published results, we provide a detailed description and analysis of a randomized variant of van Emde Boas tree structure. The variant achieves asymptotically optimal space usage, but the $\Theta(\log\log N)$ time bounds are no longer worst-case but expected amortized.

The best published expected amortized time bound that is achieved on the $(s^\alpha, s^{1-\delta})$-smooth class of distributions is equal to $\mathcal{O}(\log\log n)$. We combine the known techniques into a new structure that achieves the same time bound on a wider class of input distributions. Moreover, the new structure can utilize the optimal amortized structure proposed by Beame and Fich, which ensures that the amortized time complexity is also bound by the optimal $\Theta\left(\sqrt{\log n / \log\log n}\right)$.

Keywords: dynamic data structures, predecessor problem, expected time complexity

1 Introduction

In the past years the computers have been increasing their processing power and storage capacity very rapidly. As the process is expected to continue in the future, it might seem that efficient algorithms and data structures become less important. However, the more one has, the more he wants. The demand to process huge amounts of data grows at least as fast as the speed of hardware. In the days of first computers the size of processable data was quite small. Therefore, most of the work on making the programs perform faster was done on the low level, by tweaking the code to improve multiplicative constants. As the size of data grows, these constants become less important and they are slowly overcome by asymptotic behaviour of the algorithms used. As a consequence, the optimization work in high-performance computing change from low-level tweaking to high-level design changes, utilizing more advanced data structures and algorithms.

In this thesis we focus on one simple data-structure problem that has been studied for a long time and also frequently appears as a sub-problem in various algorithms. Contrary to the classical textbook theory we use a different computation model that is becoming popular in recent publications. We chose it because it more closely matches the abilities of real hardware that is available today and in the foreseeable future.

The layout of this text is as follows. We start in the next chapter by formulating our central problem. We also describe the computation model, the applicability on related problems and we discuss the state of the art in the worst-case solution of our problem. Then in chapter 3 we survey notable advances in the history of expected-time solutions. Starting with a well-known array search algorithm, we follow the development up to the most recent dynamic data structures. In chapter 4 we concentrate on a known structure that uses a clever decomposition. We present a detailed design of its randomized variant with reduced space requirements and we analyse its properties. Then in chapter 5 we utilize the structure's principles and adapt it to the input models known from expected-time solutions. As a result we obtain a new structure with performance comparable to the state of the art from chapter 3. Finally, we sum up our thesis in chapter 6.

2 The basics

2.1 Models

The predecessor problem

Through the whole thesis we will work with data structures for the predecessor problem. It appears in many equivalent variations under different names, such as successor problem, dynamic ordered set problem or ordered dictionary. To avoid confusion, we define the problem as follows.

Definition 1. Data structure solving the **predecessor problem** maintains a dynamic subset of an ordered universe set. It implements the following operations:

- Member(x) tests if the value x belongs into the represented set
- FindMin() and FindMax() search for a bounding value of the set
- Insert(x) and Delete(x) modify the represented set by including or excluding one element
- Predecessor(x) finds the greatest contained element that is less than x
- Successor(x) finds the least contained element that is greater than x

Note that the argument of Predecessor or Successor doesn't have to be contained in the set. Sometimes we consider a static version of the problem, where the structure is constructed in advance and the modifying operations Insert and Delete are not allowed.

Computational model

In the classical model where only comparison operations are allowed on stored elements, the predecessor problem is already solved optimally (in the sense of asymptotic time and space). The $\mathcal{O}(\log n)$ time bound on the operations can be achieved easily by balanced trees. If we were able to perform the operations quicker, we could easily break the $\Omega(n \log n)$ lower bound on comparison-based sorting by first building the structure from all elements by Inserts and then removing them in order by FindMin and Delete operations.

However, the comparison model isn't realistic, because all ordinary computer architectures can perform much wider range of operations. That is why we use another popular model, the word-RAM [14].

RAM is a family of models that work only with integers. The memory looks like an array of integers and can be indexed (the integers may be used as pointers). The original model was too strong and there are several used ways of restricting it to reasonable strength.

The word-RAM modification restricts the size of the integers processable in unit time and calls them *words*. We will use w to denote the maximum bit-length of words. Since in this model we are only able to address space of 2^w words, we suppose in addition that for an input of length n we have words of size $w \geq \log n$.[1] Otherwise we wouldn't even be able to read the whole input.

All operations of word-RAM work in constant time and on a constant number of words. The available types of arithmetic operations consist of those that are available in standard programming languages, such as C. Sometimes, however, arbitrary operations that belong to the AC_0 complexity class are allowed instead.[2] Such model is often called AC_0-RAM. Although this restriction may seem theoretical, AC_0 operations are performed much quicker on most modern processors. That is caused by the fact that the AC_0 class by definition consist of problems with efficient digital circuits.

The most useful advantage over the comparison model is that RAM allows us to compute values from input and use the results as addresses. Similar tricks are widely used (even in practice) and they are known for a very long time. Among the first notable occurrences are hashing and radix sort.

Restriction to integers

When solving the predecessor problem, we will only consider the situation when we store a set of nonnegative integers that fit into one word. We will show that this restriction isn't as bad as it may appear, because any solution of this basic case can be extended to solve the same problem on some other kinds of keys.

Adding support for negative integers can be done, for example, by increasing the value of every key by a certain amount so all of them are nonnegative. Such transformation is very simple, invertible and preserves ordering completely.

The most common models for non-integers used today are floating-point numbers defined by the IEEE-754 standard. They can represent a subset of rational numbers in form $p\,2^q$ for integers p and q. The standard ensures that the ordering of two nonnegative floating-point numbers is the same as lexicographical ordering of their bit representations, assuming the numbers use the same kind of format [12, section 2.1]. That means we can interpret the keys as integers without disturbing comparison results. If we needed both negative and positive floats, we could store

[1] We will use "log" to denote a base-2 logarithm.

[2] Common non-AC_0 operations are multiplication, division and modulus.

their absolute values separately in two structures and simulate operations on their composition by a few operations on the two substructures.

Andersson and Thorup [2] showed how to generalize an arbitrary structure for the predecessor problem to work with strings. In such structures the keys can span multiple words and they are ordered lexicographically.

When designing set data structures, we usually only consider the values of keys. In practice, however, we often need to associate some other data with every stored element. This simplification is due to the fact that most data structures can be easily generalized to handle such cases. In cases where it isn't possible, the additional data can be stored outside, in a dictionary structure containing the same set of keys. We will use this technique later and describe it in more depth in section 4.1. As an alternative on RAM models, it is possible to simply append a pointer to every stored key. All keys become longer by w bits (the length of the machine word), which usually doesn't impact asymptotic bounds on time or space. Obviously, the order of the keys isn't affected, but the Member operation gets more complicated because it needs to be simulated by a call to Successor.

2.2 Worst-case bounds

The asymptotic worst-case complexity of the predecessor problem on word-RAM is already understood quite well. Beame and Fich [5, 4] proved lower bounds on the static version of the problem. They used the cell-probe model which is strictly stronger than word-RAM (it only counts memory accesses into the complexity), but we will formulate equivalents of their results on the word-RAM model.

Consider a static structure storing arbitrary n elements from the universe set U of size N on a word-RAM with w-bit words. We assume that the structure needs $n^{\mathcal{O}(1)}$ words of space. Then it holds:

- If $w \in (\log N)^{\mathcal{O}(1)}$, then the predecessor queries need $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ time in the worst case [4, Corollary 3.10].
- If $w \in 2^{(\log N)^{1-\Omega(1)}}$, then the predecessor queries need $\Omega\left(\frac{\log\log N}{\log\log\log N}\right)$ time in the worst case [4, Corollary 3.9].

Note that if we allowed too big words, we could efficiently store the set as a bit map – we would achieve constant time per query for $w \in \Omega(2^{\log N}) = \Omega(N)$. Similarly, if we allowed to use too big number of words, we could precompute the answers for all possible queries.

Beame and Fich also proposed a static structure that achieves the asymptotic bounds [4, Theorem 4.5]. It answers the predecessor queries in worst-case time $\mathcal{O}\left(\min\left\{\sqrt{\frac{\log n}{\log\log n}}, \frac{\log\log N}{\log\log\log N}\right\}\right)$ and it can be constructed in time and space $\mathcal{O}(n^{2+\epsilon})$ for an arbitrary $\epsilon > 0$.

Moreover they showed that the structure can be dynamized by Andersson's exponential trees. The trees were first designed for converting a static structure for the predecessor problem into an amortized structure, but later Andersson and Thorup proposed a modified version that achieves identical worst-case time bounds (complete descriptions can be found in [3]).

The structure created as a combination of the static structure from Beame and Fich with the worst-case version of Andersson's exponential trees needs $\Theta(n)$ space and performs all operations of the (dynamic) predecessor problem in time

$$\mathcal{O}\left(\min\left\{\begin{array}{c}\sqrt{\frac{\log n}{\log\log n}}\\ \log\log n\cdot\frac{\log\log N}{\log\log\log N}\\ \log\log n+\frac{\log n}{\log w}\end{array}\right\}\right) \text{ in the worst case [3, Corollary 1.4].}$$

Here the first upper bound in the minimum is *optimal*, because even for the static problem with polynomial construction the queries can't be answered faster in the worst case. The second bound is not likely to be tight and it should also be compared to the $\mathcal{O}(\log\log N)$ time of van Emde Boas trees. However, the VEBT implementations either use enormous space of $\mathcal{O}(N)$ or need hashing which makes the time bounds expected instead of worst-case (VEBT structures are discussed in chapter 4). The third bound comes from an Andersson's modification of fusion tree structure (for more details see [3, section 4.1]).

The worst-case time bound of $\Theta\big(\sqrt{\log n/\log\log n}\,\big)$ on the predecessor problem is settled. However, it can still be outperformed if we know more information about the input and we minimize the expected time instead of the worst-case time. Studying such techniques is the main point of our work.

3 Interpolation search methods

3.1 Searching sorted arrays

The first attempts to utilize restricted input distributions concentrated on effective searching in sorted arrays. That corresponds to the static version of our predecessor problem. Sorted arrays can be trivially searched by binary search, but that only results in $\Theta(\log n)$ time.

Interpolation search was designed for arrays containing numbers uniformly distributed over an interval. The algorithm was first proposed by Peterson [25]. It also works on every distribution known in advance, provided that the cumulative distribution function is continuous and easily computable. During search the algorithm maintains such a segment of the array that the searched value belongs between the bounds of the segment. In every iteration the segment is split on a guessed position and the algorithm continues with the correct one of the two.

Let A denote the searched ordered array containing values chosen independently according to a continuous cumulative distribution function F. When searching for a value x in a segment with indices $A_l \dots A_h$, the algorithm probes the index

$$i = \left\lceil \frac{F(x) - F(A_l)}{F(A_h) - F(A_l)} \, (h - l - 1) \right\rceil + l$$

(there are several versions of the algorithm that differ in the way of rounding the indices).

It was later proved in several ways that the expected number of probes in interpolation search is $\log\log n + \Theta(1)$ [32, 23, 13] and that this is also the lower bound on the expected time of any algorithm choosing the probed indices [32]. This is an exponential speedup compared to the classical method of binary search. However, the worst-case number of probes is $\Theta(n)$.

Perl and Reingold [24] proposed a modified algorithm that interleaves the interpolation steps with unary and binary steps. Contrary to the original algorithm it guarantees an $\mathcal{O}(\log n)$ worst-case bound and it is much simpler to show that the expected number of probes is $\mathcal{O}(\log\log n)$.

Willard [29] noted that the interpolation search chooses probed indices analogously to the regula falsi method of finding roots in numerical analysis. He used ideas from improved versions of regula falsi to construct a new algorithm for choosing the probed index and he showed that the expected number of probes is $\mathcal{O}(\log\log n)$ even when the data are generated according to an arbitrary (unknown) distribution from a class of "regular" distributions.

The constant factors hidden in Willard's method are much bigger than in the original interpolation search. This issue was addressed by Carlsson and Mattsson [6] by combining interpolation steps with "extrapolation steps". They proved that the new algorithm does at most four probes more than interpolation search (on average) when used on uniformly distributed data. Carlsson and Mattsson also proposed a method of selecting the starting interval of the search based on the knowledge of an estimate of the input distribution. They use precomputed least-squares line approximation of the input's cumulative distribution function. Combination of these two improvements was compared experimentally to the original interpolation search on several distributions (uniform, normal and exponential). On nonuniform distributions the original interpolation search method degraded quickly, performing much worse than a simple binary search, and the proposed combination only needed a low number of probes on average. It was a little quicker even on the uniform distribution. However, no bounds for behaviour on unknown distributions were proved.

3.2 A dynamic model

The interpolation search algorithm is very quick and simple, but it suffers from two big problems. One of them is the bad behaviour on unknown nonuniform distributions which was already discussed above [29, 6]. The other issue is that interpolation search only solves the static version of predecessor problem.

We are going to describe several publications that discuss solution of the dynamic predecessor problem. The latest ones don't have much in common with the original method, but they are still usually called dynamic interpolation search. The expected complexity analysis of fully dynamic data structures brings new complications. We use the model of μ-random insertions and random deletions, because it is customary in the publications that we work with.

Definition 2. **μ-random insertions** take values according to a density μ that doesn't change during the whole life of the structure. **Random deletions** remove uniformly from the set of *contained* elements. All the modifying operations are arbitrarily intermixed and independent.

This model is convenient because it preserves "randomness" of the stored set – at any point of execution the set appears to be created by independent insertions only. That is usually essential for complexity analyses of the structures we will discuss. A more thorough study on various random deletion models was published by Knuth [18].

3.3 Interpolation search tree

The first structure using this randomized model to achieve reasonable search times is interpolation search tree (IST) by Mehlhorn and Tsakalidis [20].[1] There was some previous work that could be counted into the same category, but it doesn't achieve such a good asymptotic performance and we don't use their results or ideas [15, 11].

IST structure was designed to work with a class of "smooth" distributions, which is a superset of Willard's "regular" class. The smooth class was later generalized by Andersson and Mattsson [1] and we will use their notation for IST as well. The class of distributions used in IST corresponds by the following definition to the $(s^\alpha, \sqrt{s})$-smooth distributions (for an arbitrary constant $\alpha < 1$).

Definition 3. Let μ be the density of a probability distribution over an interval $\langle a, b\rangle$. Given two functions f_1 and f_2, μ is $(\mathbf{f_1}, \mathbf{f_2})$**-smooth** iff

$$\exists\beta\ \forall c_1, c_2, c_3 \quad a \le c_1 < c_2 < c_3 \le b \quad \forall s \in \mathbb{N}$$
$$\Pr\left[X \in \left\langle c_2 - \frac{c_3 - c_1}{f_1(s)}, c_2 \right\rangle \,\middle|\, X \in \langle c_1, c_3\rangle\right] \le \frac{\beta f_2(s)}{s}$$

There is quite a simple intuition behind this complicated condition. It implies that if we cut some interval $\langle c_1, c_3\rangle$ into $f_1(s)$ subintervals and μ-generate s values from $\langle c_1, c_3\rangle$, every subinterval is expected to get $\mathcal{O}(f_2(s))$ elements.

The static case

We will first focus on building an ideal IST and discuss adding the modifying operations later. For the sake of simplicity we will omit all rounding of IST's parameters, because the it doesn't affect asymptotic properties of the structure.

Definition 4. Let $A = \{x_1 < x_2 < \cdots < x_n\} \subseteq \langle a, b\rangle$ be the set to be stored and let $k = \sqrt{n}$. The **ideal IST** for A on $\langle a, b\rangle$ and a parameter $\alpha < 1$ consist of:

- An array REP$[1 \ldots k]$ of representatives from A. The sample is chosen to be equally spaced. That is, it contains $\{x_{\sqrt{n}}, x_{2\sqrt{n}}, x_{3\sqrt{n}}, \ldots, x_{k\sqrt{n}}\}$.
- An array TREE$[0 \ldots k]$ that contains the ideal IST structures for subsets $A_0, A_1, \ldots, A_k$ created by partitioning A with the elements of REP. The representatives are not stored recursively and they act as new boundaries of the subtrees (their a and b parameters). The parameter α remains the same in all subtrees.

[1] The IST structure is a bit tricky, because it was published in two different versions. The first one was presented in 1985 on ICALP and LNCS, but it had some pages in wrong order and some missing. Here we rather refer to the other version from ACM 93 journal. There some proofs were modified, some added and the final parts were completely changed. Also, the definition of "simple IST" was dropped and the new definition of IST corresponds to the original "augmented IST".

- An array ID$[1 \dots m]$, $m = n^\alpha$, containing indices to REP satisfying

$$\text{ID}[i] = j \quad \longleftrightarrow \quad \text{REP}[j] \; < \; a + i\frac{b-a}{m} \; \leq \; \text{REP}[j+1]\,.$$

The ID array can be viewed as an approximation of the quantile function (the inverse to the distribution function). When searching for x in IST (not necessarily ideal), we at first use ID to estimate the position of x in REP according to the definition of ID. This way we directly obtain an upper and lower bound on the index in REP where x can occur (the values in REP are in ascending order). Then, using binary search, we either find the exact position of x in REP or we find the index of the subtree in which x belongs and we search it recursively. Searching for predecessors and successor works the same, because one can easily traverse the contained elements in order.

Dynamization

The ideal IST can only store a static set, so the modifying operations are defined as follows (leading to a non-ideal IST). Inserting is simply done into the leaf that would be found when searching for the inserted value – a new one-valued IST is inserted into the right empty place. When deleting, the value is found and it is only marked to be deleted. The ID and REP arrays are never changed on insertions or deletions, but it is necessary to periodically rebuild individual subtrees.

A counter is added into every node of the structure to store the number of modifying operations since the last rebuild. When the counter exceeds $n_0/4$ (where n_0 is the number of elements stored by the subtree during the last rebuild), the whole subtree of the node is completely rebuilt. All unmarked contained elements are extracted and a new ideal IST is built as defined above. The rebuild can be obviously done in time $\Theta(n_0)$, so the needed time can be amortized into additional constant time per modifying operation.

Asymptotic complexity

Now we briefly discuss the asymptotic complexities proved by Mehlhorn and Tsakalidis. More details on IST and proofs can be found in [20]. The expected bounds will hold when the modification queries are μ-random insertions and random deletions where μ is a $(s^\alpha, \sqrt{s})$-smooth distribution. That is, the averaging is done over the distribution of the input and the structure uses no random bits. We will use n to denote the current number of elements in IST.

When the parameter $\alpha < 1$, the space complexity is $\Theta(n)$, otherwise it would be superlinear.[2] The ideal IST has child trees for subsets of size $\Theta(\sqrt{n})$, so its depth is $\Theta(\log \log n)$. The modifying operations can increase the depth, but it

[2] In our work we will usually only consider structures with linear worst-case space complexity.

is bounded by $\mathcal{O}(\log n)$ in the worst case. The expected depth of an arbitrary element in IST is still $\Theta(\log \log n)$ in the chosen model of input.

When operating on IST, the binary search in the REP array clearly needs $\mathcal{O}(\log n)$ time in the worst case. Since the worst-case depth is $\mathcal{O}(\log n)$, the worst-case search time is bounded by $\mathcal{O}(\log^2 n)$. However, the expected case is better. We use binary search on a range of REP given by the ID array. The length of this range is only constant on average thanks to choosing the length of ID according to the smoothness of μ.

To sum up, the time complexities of IST on predecessor-problem operations are:

- $\text{searching} = \begin{cases} \mathcal{O}(\log^2 n) & \text{worst-case} \\ \mathcal{O}(\log \log n) & \text{expected} \end{cases}$
- $\text{modification} = \text{searching} + \begin{cases} \mathcal{O}(n) & \text{worst-case} \\ \mathcal{O}(\log n) & \text{amortized} \\ \mathcal{O}(\log \log n) & \text{expected amortized} \end{cases}$

3.4 Augmented sampled forest

Andersson and Mattsson improved the interpolation search tree structure in several ways [1]. We are only going to briefly cover the differences of their *augmented sampled forest* (ASF) against IST.

Improved rebuilding

The first change is simplification of the rebuilding scheme. In IST is every subtree of size n_0 completely rebuilt after $\Theta(n_0)$ modifications. In ASF this is only done with the whole tree which is static between the rebuilds. That means we need to have some simple dynamic substructures in the leaves (balanced trees were chosen for ASF). After a rebuild, every leaf substructure contains exactly one element, but it can grow or shrink until the next rebuild.

Since the layout of the tree is only determined by static parameters and n_0 (the number of contained elements during the last rebuild), the whole tree can be made implicit and allocated in one huge memory region. That is an advantage for practical implementation, because it helps to decrease constant factors of time and space complexities.

Andersson and Mattsson also proposed to use the technique of *global rebuilding* [22] to spread the tree rebuild into $\Theta(1)$ extra work done on every insertion or deletion. In this way it is possible to cut down the worst-case asymptotic time bounds to the values of amortized bounds.

Generalization of input distribution

The IST was tailored for the class of $(s^\alpha, \sqrt{s})$-smooth distributions (which were called just "smooth" by Mehlhorn and Tsakalidis). Andersson and Mattsson noticed that the class can be generalized and they extended the parametrization of the structure. The ASF containing n leaves is characterized by three nondecreasing functions $H(n)$, $R(n)$ and $I(n)$. They determine the height, the degree of the root node and the length of the ID array in the root.

The parametrization functions aren't independent. If we want to achieve height $H(n)$, we have to choose $R(n) = n \,/\, H^{-1}(H(n) - 1)$.[3] $I(n)$ should be as large as possible to allow fast searching on a wider class of distributions. The only restriction is that the total consumed space for ID arrays is $\mathcal{O}(n)$.

Andersson and Mattsson proved that total linear space can be guaranteed by choosing $I(n) = n \cdot g(H(n))$ for any function g that satisfies $\sum_{i=1}^{\infty} g(i) = \Theta(1)$. Then we get an ASF that supports searches and updates in $\Theta(H(n))$ expected time when the input distribution is $(\, s \cdot g(H(s)),\ H^{-1}(H(s) - 1)\,)$-smooth [1, Theorem 6].

For comparison with the IST structure they chose $H(n) = \Theta(\log\log n)$ and $g(x) = 1/x^{1+\epsilon}$ for an arbitrary $\epsilon > 0$. Then the $\Theta(\log\log n)$ expected time of ASF holds on the class of $\big(s/(\log\log s)^{1+\epsilon}, s^{1-\delta}\big)$-smooth distributions for any $\delta > 0$. This class is a superset of the $(s^\alpha, \sqrt{s})$-smooth class of distributions with $\alpha < 1$, which was used in IST.

Constant time for bounded densities

A special case was pointed out by Andersson and Mattsson when choosing depth

$$H(n) = \begin{cases} 0 & \text{for } n = 1 \\ 1 & \text{for } n > 1 \end{cases}$$

Then the ASF collapses to one level of n buckets dividing the space into identical intervals represented by balanced trees. This way we get expected $\Theta(1)$ time for any bounded distribution (the $(s, 1)$-smooth class).

In particular, this is a convenient representation for uniformly distributed sets because it is very simple and all the operations are even quicker than on a sorted array with interpolation search.

3.5 Recent improvements

The most recent published improvements of dynamic interpolation search were discovered by Kaporis et al. [16]. They noticed that the REP array in ASF is not

[3] That is a simple observation – the denominator is the number of elements contained in every child of the root node.

needed to achieve quick search times and the ID array itself can store pointers to the subtrees. This way the tree doesn't divide the stored set into parts with roughly equal sizes, but it rather partitions the universe set. To reduce the depth, small subsets are represented differently. Kaporis et al. chose the *q*-heap* structure for sets of polylogarithmic size, which makes the leaf structures work in $\Theta(1)$ worst-case time.

The q*-heap structure [30] is a heavy-weight word-RAM data structure that uses precomputed tables of size $o(N)$ when operating on a universe set of size N. It can perform all operations of the predecessor problem in time $\mathcal{O}(1 + \log n / \log\log N)$ in the worst case [30, Corollary 2] and it only needs $\mathcal{O}(n)$ words of memory (not counting the table). As a consequence, sets of size $n \in (\log N)^{\mathcal{O}(1)}$ can be handled in constant time. The needed tables can be precomputed in time $\mathcal{O}(N^\epsilon)$ for an arbitrary $\epsilon > 0$ and they can be shared by all instances of the structure. However, Kaporis et al. do not mention this precomputation requirement and they also don't seem to account the needed time because their stated time bounds are independent of N [16, Theorem 2].

Behaviour on various distributions

Now we summarise the complexities claimed by Kaporis et al. The $\Theta(\log\log n)$ expected time bound is achieved on the same class of distributions as the ASF structure. Similarly to ASF, the new structure has also constant depth with high probability on bounded distributions and moreover also on $(s, \log^{\mathcal{O}(1)} s)$-smooth distributions. This difference is caused by the fact that the newer structure uses much stronger leaf substructures.

Kaporis et al. also sketched how to use the new structure to store elements taken from power law or binomial distributions. They cut the interval that is too dense to satisfy the smoothness condition and represent it separately in van Emde Boas tree structure [27, 28]. The remaining one or two intervals are then represented in the usual way.

Worst-case space usage

It is stated in [16, Theorem 2] that "the space usage of the data structure is $\mathcal{O}(n)$". However, from the description of structure's layout it follows that it only holds with high probability, not in the worst case.[4]

A simple problematic input consists of n consecutive values such that they don't fit into one leaf. Obviously, the structure will shape itself into a linear chain of nodes with several leaves at the end. Every non-leaf node takes $\Theta(f_1(n))$ space and the length of the chain does *not* depend on n but on the size of the universe

[4] The description only considers *ideal* trees where no node of the tree has a child with its subtree bigger than half of its parent's subtree.

set. That can be arbitrarily large compared to n and thus the worst-case space usage *is not bounded* in terms of n.

The situation can be easily improved, for example, by a different representation of such chains of nodes. That would ensure the nodes form at least binary tree and the depth is $\mathcal{O}(n/f_L(n))$, but the space usage would still *not* be $\mathcal{O}(n)$.

Summing up

The work of Kaporis et al. shows several interesting ideas. For example, the partitioning of the universe set doesn't depend on the stored values, so the expected-time analysis is simpler. Moreover, the worst-case search time is kept on the optimal bound of $\Theta\left(\sqrt{\log n / \log\log n}\right)$ by a combination with the optimal worst-case structure discussed on page 8.

On the other hand, there are many unclear properties of the structure. Some of them were already mentioned. Also, no efficient algorithm is given for the Predecessor and Successor operations. Kaporis et al. state that proofs and some more details about the structure are omitted and they are available in the full version [17]. Unfortunately, we weren't able to obtain the full version to resolve the issues.

3.6 A different approach

We finish the survey of methods based on interpolation search by a brief mention of a different approach from Demaine et al. [9]. We will not use their ideas or results, but we list them to complete the collection of related research.

Instead of using the randomized model (defined on page 11) with restricted distribution classes, they introduce a new *deterministic* metric of "well-behaved data" – for $x_1 < x_2 < \cdots < x_n$ they define the maximum gap ratio $\Delta = {}^{\max\{x_i - x_{i-1}\}}/_{\min\{x_i - x_{i-1}\}}$.

Demaine et al. propose a simple static structure that divides the interval $\langle x_1, x_n\rangle$ into n buckets of the same length where the elements of every bucket are stored in a balanced search tree. Every bucket stores all elements that naturally belong into it (by their value) and in addition it contains the nearest stored neighbour above and below (if they exist). This layout is very similar to the structure from Kaporis et al. when used on bounded distributions (the work of Kaporis et al. was published later). It is obvious that the proposed structure uses $\Theta(n)$ space and can be built in $\Theta(n)$ time in the worst case (from an ordered list). They also show a simple proof that the structure only needs $\mathcal{O}(\log\Delta)$ worst-case time for searching. The predecessor or successor can be easily found directly in the corresponding bucket because of the choice that buckets additionally store their nearest neighbours.

To obtain some comparison with the previously known structures, Demaine et al. also prove that for uniformly distributed independent data the gap ratio is polylogarithmic in n with high probability, that is, $\Delta \in (\log n)^{\mathcal{O}(1)}$ w.h.p. As a

consequence, the static structure needs expected time $\mathcal{O}(\log \Delta) \subseteq \mathcal{O}(\log \log n)$, but both the ASF structure and the structure of Kaporis et al. perform better on these conditions. They only need expected constant time, because the uniform distribution belongs into the $(s, 1)$-smooth class.

Moreover, Demaine et al. show how to obtain a dynamic version of the structure that needs $\mathcal{O}(\log \Delta_{max})$ time per operation where Δ_{max} denotes the highest value of Δ over the whole life of the structure. The original bound of the structure is amortized, but it is claimed that the rebuilds can be spread out to achieve a worst-case bound.

4 van Emde Boas trees

There is a data structure designed by van Emde Boas [26, 27, 28] that solves the predecessor problem on a universe set of fixed size N. It can perform all the operations in time $\Theta(\log \log N)$ in the worst case. Therefore, if the stored set covers relatively big fraction of the universe set, this structure is very fast. However, if the universe is more than exponentially larger, $n \in o(\log N)$, the structure is asymptotically slower even than the trivial solution by balanced trees.

The greatest disadvantage of the structure is its space consumption which is only bounded by $\mathcal{O}(N)$. That can be very large in terms of n (the size of the stored set), so the usability is limited compared to structures with the usual $\mathcal{O}(n)$ space bound. In this chapter we describe a modification of the van Emde Boas structure that uses hashing to achieve the $\mathcal{O}(n)$ optimal bound. However, the operation time bounds can no longer be worst-case. An almost identical structure was proposed by Mehlhorn and Näher [19], which we discovered after modifying the VEBT design.

In the following section 4.1 we discuss the needed hashing techniques. Then the construction itself is split into two steps corresponding to the sections 4.2 and 4.3.

4.1 Dynamic dictionaries

The defined structure makes use of dynamic dictionaries, which solve much simpler problem than the predecessor problem. We will often shorten the name just to "dictionary". We first define their semantics of operation and then briefly discuss the possibilities of implementation and asymptotic complexities.

Definition 5. Dynamic dictionary is any data structure that stores a dynamic set of keys where every key is associated with a value. It provides the following three operations:

- Find(x) searches for key x in the stored set and returns either the associated value or a special value null if the key is not contained.
- Insert(x, y) inserts the key x associated with the value y. It can be used only when x is not contained in the stored set.
- Delete(x) removes the key x from the stored set (together with the associated value). It can be used only when x is represented in the set.

Since here we work with the word-RAM model, the universe set for key values is always in the form of $U = \{0, \ldots, N-1\}$ for some $N \in \mathbb{N}$. We only consider $N < 2^w$, so the keys fit into one machine-word and thus the arithmetic operations with keys take constant time (it can be easily extended for keys fitting into a constant number of words without affecting asymptotic complexities).

A trivial example of a dictionary implementation is directly addressed array, which is used in the original VEBT structure. That causes a huge waste of space in the typical case when the dictionary only stores a very small fraction of the universe set.

We want to use an implementation that only uses $\Theta(n)$ words of memory when storing n keys and needs expected amortized constant time on any operation. That is typically achieved by hashing techniques. The conditions are satisfied, for example, by dynamic perfect hashing [10], which in addition ensures constant-time Find in the *worst case*.

However, we don't really need such a complicated dictionary. A simple dynamic hashing scheme with separate chaining is sufficient. For example, we can maintain the size m of the hash table such that $n \in \left(\frac{m}{2}, 2m\right)$ and we resize (rehash) the table to $m = n$ whenever n gets out of the interval. In such setting the rehashing *into* a table of size $m = n$ occurs after at least $\frac{m}{2}$ modifying operations since the last rehashing. Moreover, the invariant on the table size ensures that we always have $m \in \Theta(n)$, so $\frac{m}{2} \in \Omega(n+m)$. During the rebuild, a new hash function is chosen for the new table size m, then the whole old table is read and all elements are moved into the new table. That takes time $\Theta(n+m)$, because the old table size is at most $2m$. Consequently, this rebuilding time can be amortized into additional constant time for the $\Omega(n+m)$ modifying operations since the last rebuild.

For the hash table we need to use a hash function that distributes the universe set uniformly in the table. We also require that it is randomly chosen from a *universal* family of functions to avoid the existence of "bad inputs". We won't explain here the theory of universal hashing, because it is widely known and it can be found in most textbooks about data structures, for example in *Introduction to Algorithms* [7, in section 11.3]. The universal hashing ensures that if excluding the table rebuilding, the expected time needed for any operation is $\mathcal{O}\left(1 + \frac{n}{m}\right)$, which is $\mathcal{O}(1)$ in our setting. Moreover, the required time is independent of the input – the time is expected constant for *any* input, computing the expectation *only* over the random choice of the hash function.[1]

To sum up, our simple combination of universal hashing with table resizing yields a dynamic dictionary with the desired properties. The space requirements of the table with chains is obviously $\Theta(n+m)$, which is always $\Theta(n)$. The expected time of the Find operation is $\Theta(1)$. Into the modifying operations we need to add the amortized $\Theta(1)$ time for rebuilding work, so we get amortized expected $\Theta(1)$ time for Insert and Delete.

[1] Here it is supposed that the input is independent of the random choice (i. e. the adversary doesn't know our random bits).

Conclusion 6. There is a randomized dynamic dictionary structure that needs $\Theta(n)$ words of space in the worst case and it meets the following bounds for *any* sequence of operations. The Find operation needs $\Theta(1)$ expected time. The Insert and Delete operations need $\Theta(1)$ amortized expected time.

Note that a dynamic dictionary can be a useful extension to any predecessor-problem data structure. Maintaining a dictionary on the same set of keys only needs $\Theta(1)$ amortized expected additional time per Insert and Delete. Then we can implement Member by a call to Find on the dictionary (in $\Theta(1)$ expected time) and it even allows us to associate every key in the set with a value, if the original structure didn't support it. This scheme is also used in the next section for decomposing the predecessor problem, where one field (HIGH) that contains a structure for the predecessor problem is paired with another field (LOW) that contains a dictionary on the same set of keys.

4.2 HVEBT

As the first step in designing a space-efficient structure we define and analyze a simple modification of van Emde Boas trees that uses dynamic dictionaries from the previous section instead of arrays. We will use the acronym HVEBT to refer to the structure.

The basic idea of van Emde Boas layout is that the k bits of every stored number are split into two parts where the most significant $\lfloor k/2 \rfloor$ bits are used as an index of tree that stores the remaining $\lceil k/2 \rceil$ bits recursively.[2] Another way to view the process is that the universe set U, denoting $N \equiv |U|$, is split into $\sqrt{N}$ segments of size $\sqrt{N}$ (approximately).

Definition 7. Let $A \subseteq U \equiv \{0, \ldots, 2^k - 1\}$, $A \neq \emptyset$. **HVEBT**(k) storing the set A contains:

- MIN, MAX – the minimum and maximum value in A. The value in MIN is not stored recursively in subtrees, but the MAX only works as a cache and it is also stored recursively as any regular value.
- LOW – a dynamic dictionary indexed by keys from $\{0, \ldots, 2^{\lfloor k/2 \rfloor} - 1\}$ containing nonempty subtrees of type HVEBT$(\lceil k/2 \rceil)$ as values. We will use the dictionary as a black box, so we only need its implementation to meet the criteria discussed in section 4.1.
- HIGH – a subtree HVEBT$(\lfloor k/2 \rfloor)$ containing the nonempty indices of the LOW dictionary.

[2] We will use $\lfloor x \rfloor$ and $\lceil x \rceil$ to denote the floor and ceiling of x, respectively.

The MIN and MAX fields ensure two key properties. First, they allow to query the border values in constant time, which is heavily used in algorithms for the operations. Moreover, the fact that MIN isn't stored recursively means that any child subtree stores less values than the whole tree, so the structure for sets of constant size can be constructed in constant time (independent of k). Here we make a compromise by storing just one value directly without recursion. That will enable us to implement two useful additional operations.

The core of the decomposition is in the LOW field. The represented set A is split into subsets A_i by the value i of the high halves of the numbers' bit-representations. In mathematical notation we have $A_i = \left\{ x \in A \,\middle|\, \lfloor x/2^{\lfloor k/2 \rfloor} \rfloor = i \right\}$. The values within one subset have the same high halves of their bits, so only the low halves are represented in a HVEBT with half bit-length. That is, the corresponding HVEBT represents the set $A'_i = \left\{ x \bmod 2^{\lfloor k/2 \rfloor} \,\middle|\, x \in A_i \right\}$[3] and it is stored in the LOW dictionary as a value associated with the key i (the value of the high bits serves as the key). Note that only *nonempty* subsets have an entry in the LOW dictionary and the minimum value in the stored set is excluded. The dictionary itself can't provide any ordering-related operations, so the ordering of the subsets is maintained in the HIGH field. It stores the keys used in the LOW dictionary (the i values) in a HVEBT subtree of half bit-width.

The recursive construction obviously stops when $|A| = 1$, where the only value is stored in MIN. That happens in the worst case for all subtrees of a HVEBT(1) structure.

Operations on HVEBT

We show here straightforward algorithms for operations on the HVEBT structure. They are analogous to the well-known operations on standard van Emde Boas tree. Together with algorithm description we present pseudocode notation to specify a possibility of implementation more formally.

Performing the operations FindMin and FindMax is trivial, since the minimum and maximum are always stored directly in the root. Creating a new tree containing one element is simple as well, it only initializes the border values and creates a new empty dictionary, as shown in the pseudocode on the following page.

The rest of the operations uses simple auxiliary methods SplitBits and JoinBits. They are set apart to make the remaining code more readable as it often needs to work separately with both halves of bit representations of some values. The pseudocode is shown on the next page.

We use operators from the C language to perform bit shifting and bit masking. When using them we provide equivalents in mathematical notation. The bit twiddling operations are quite unusual on the field of theoretical computer science, but we believe that here the notion of values as bit-strings is more suitable and

[3] We use "mod" to denote the modulo operator, which computes the remainder of division of two natural numbers.

illustrative than the equivalent power-of-two notations. The left shift by b bits (denoted $x \ll b$) corresponds to multiplication by 2^b, whereas the right shift by b bits (denoted $x \gg b$) corresponds to division by 2^b (we only consider nonnegative numbers).

---

alg. HVEBT(k).NewTree(x) {creates a new structure storing value x}
  MIN, MAX $\leftarrow$ x
  HIGH $\leftarrow$ null
  LOW $\leftarrow$ "a new empty dictionary"
  return this

alg. HVEBT(k).SplitBits(x) {splits bits of x into high and low part}
  var $n_l \leftarrow (k+1)/2$ {the number of low bits, $n_l = \lceil k/2 \rceil$}
  var $x_h \leftarrow x \gg n_l$ {the value of high bits, $x_h = \lfloor x/2^{n_l} \rfloor$}
  var $x_l \leftarrow x - (x_h \ll n_l)$ {the value of low bits, $x_l = x - x_h \cdot 2^{n_l}$}
  return (x_h, x_l)

alg. HVEBT(k).JoinBits(x_h, x_l) {does exactly the inverse of SplitBits}
  var $n_l \leftarrow (k+1)/2$
  return $(x_h \ll n_l) + x_l$ {$= x_h \cdot 2^{n_l} + x_l$, the joint value}

Before we discuss the nontrivial operations Find, Insert, Delete and Successor in more depth, we want to point out the key parts of their design. Some of the algorithms are complicated, because many special cases can occur, but the basic idea is very simple. The algorithms always solve the problems by recursively solving the problems on the two halves of the input, that is by performing operations on the subtrees. The only important trick is to allow at most one nontrivial recursive call in any case. We will make sure that on every level of recursion we only make a constant number of primitive operations and a constant number of dictionary operations in the worst case. Since the dictionary operations need either expected constant time or expected amortized constant time (according to Conclusion 6), for the total time we will get an expected or expected amortized bound proportional to the depth of the tree.[4] We bound the depth to $\mathcal{O}(\log k)$ in the lemma below, so ensuring these properties on the operations discussed on several following pages implies the $\mathcal{O}(\log k)$ time complexities summarized in Conclusion 9 on page 28.

Lemma 8. *The depth*[5] *of HVEBT(k) is at most* $\lceil \log k \rceil + 1$.

Proof. We already noted, that a HVEBT(1) structure can only have one child, which is a leaf. According to Definition 7, the parameter k decreases to $\lceil k/2 \rceil$

[4] Note that here the expectation will only come from the usage of dynamic dictionaries, so the averaging is independent of the input set. That means there are no "bad" inputs, only unlucky combination of inputs and (pseudo)random bits.

[5] We consider that the root node is in depth 0 and every subtree is depth one larger than the parent. The depth of the tree is then the maximum from the depth of all nodes.

and $\lfloor k/2 \rfloor$ in the child nodes. Consequently, increasing the parameter k can't decrease the parameter in the descendants, so we can bound the maximal depth of HVEBT(k) by the maximal depth of HVEBT(k') for an arbitrary $k' > k$. We choose the nearest power of two, $k' := 2^{\lceil \log k \rceil}$.

For a power of two we have $\lceil k/2 \rceil = \lfloor k/2 \rfloor$ and this also equals a power of two, so the property is inherited in all descendants. As a result we have a HVEBT(1) in the depth $\log k'$, so the total maximum depth of HVEBT(k') is $1 + \log k'$. As a result, the maximum depth of HVEBT(k) for any $k \in \mathbb{N}$ is bound by $1 + \log k' = 1 + \log\left(2^{\lceil \log k \rceil}\right) = 1 + \lceil \log k \rceil$. □

Member operation When testing membership of a value x, the algorithm first checks if x is stored directly in the MIN field. If it isn't and the node contains subtrees, the algorithm splits the bits of x into the low and high part (x_h and x_l). Then it looks up the high part x_h in the LOW dictionary and the associated subtree is recursively tested on membership of the low part x_l. The value x is then contained in the represented set iff both the lookup and the recursive call succeed. The pseudocode is on the current page.

The search corresponds to a pass from the root to some inner node, performing one dictionary lookup and a little constant work on every level. Since one lookup takes expected constant time and the depth is bounded by $\mathcal{O}(\log k)$, the whole operation needs $\mathcal{O}(\log k)$ expected time.

---

alg. HVEBT(k).Member(x) {tests membership of x, returns true or false}

```
{special cases}
if x = MIN then
    return true
if HIGH = null then {no other element is in the tree}
    return false

{the general case}
var (x_h, x_l) ← SplitBits(x)
var t ← LOW.Find(x_h)    {finds the subtree for values sharing x_h}
if t = null then
    return false    {the subtree doesn't exist}
else
    return t.Member(x_l)    {search the subtree}
```

Successor operation From the standard predecessor operations there remain Predecessor and Successor. Since the algorithms are completely symmetrical, we only discuss the latter. The algorithm that searches for the successor of x first checks if it is less than the MIN field. If it is, MIN is the result. Otherwise, the value is split into x_h and x_l.

Then the subtree with key x_h is looked up in the dictionary. If it is found and its maximum is less than x_l, we know that the searched value is represented in the subtree. The successor of x_l is found recursively in the subtree and its value joined with x_h makes the result. Otherwise (when the subtree doesn't exist or its maximum is at least x_l), the searched value is represented in the minimum of the next subtree. The algorithm recursively finds the successor of x_h in the HIGH structure and uses it as a key for the LOW dictionary to find the subtree (if the recursion fails, no successor of x exists). The result is formed by joining the value of the subtree's key with its minimum.

Note that again we don't make two recursive calls at the same time. Either the algorithm directly finds a suitable subtree and searches it recursively for the successor, or the algorithm doesn't find it directly and it looks for the next one by calling successor on the HIGH field and takes its minimum. We show the pseudocode on this page.

The algorithm only performs at most two lookups on the dictionary and a recursive call. As one lookup takes expected constant time, the whole operation needs $\mathcal{O}(\log k)$ expected time. No amortization is needed in this case, just as in the Member operation.

---

alg. HVEBT(k).Successor(x) {finds the minimal element greater than x}
  if $x <$ MIN **then**
    return MIN
  var $(x_h, x_l) \leftarrow$ SplitBits(x)
  {first try among the elements starting with x_h}
  var $t \leftarrow$ LOW.Find(x_h)
  if $t \neq$ null and $x_l < t$.FindMax() **then**
    var $y_l \leftarrow t$.Successor(x_l)
    return JoinBits(x_h, y_l)
  {now try the next subtree}
  var $y_h \leftarrow$ HIGH.Successor(x_h)
  if $y_h \neq$ null **then**
    var $y_l \leftarrow$ LOW.Find(y_h).FindMin()
    return JoinBits(y_h, y_l)
  else
    return null

Insert operation The algorithm for insertion of a value x first handles the corner cases. If checks if the value isn't already contained in the MIN field to avoid double insertion. If x is less than the minimum, the values are swapped and we continue inserting the former MIN value (it wasn't represented in the subtrees). Then the MAX field is updated if x becomes the new maximum value (but we continue the insertion anyway, because the MAX value only serves as a cache).

In the general case the algorithm splits x into x_h and x_l and it looks up x_h in the LOW dictionary. If the lookup succeeds, the associated subtree represents the correct subset and the algorithm recursively inserts x_l into the subtree. If the lookup fails, a new subtree containing only x_l is created, it is added into the dictionary with key x_h and the key x_h has to be also inserted into the HIGH subtree.

Note that the two recursive calls for insertion into a HVEBT subtree are exclusive – either the lookup succeeds and we insert into the found subtree, or it fails and we only insert into the HIGH subtree. The two cases can't occur both at the same time. We present the pseudocode on on the next page.

On one level of recursion the Insert algorithm does some constant work and at most two dictionary operations. Since one of the operations is insertion, which requires expected amortized constant time, the total expected amortized complexity is bounded by $\mathcal{O}(\log k)$.

Delete operation The algorithm for deletion of value x starts by handling several special cases. Deleting the last value destroys the whole structure. When deleting the minimum, the algorithm finds its replacement as follows. It finds the high bits by a FindMin query on the HIGH field, then it looks up the corresponding subtree in the LOW dictionary and finally it simply queries for the minimum of the subtree and joins the halves of the bit representation (no recursion is needed here). After that the algorithm continues as if deleting the new minimum, so it isn't represented twice.

The general case is very simple – the deleted value is split into x_h and x_l, the value of the high bits x_h is looked up in the LOW dictionary and the value of the low bits x_l is recursively deleted from the found subtree. If either the lookup or the recursive delete fails, the value wasn't present in the structure. It is necessary to handle the case, when the recursive delete completely empties the subtree. Then the empty subtree has to be removed from the dictionary and also its key x_h has to be deleted from the structure in HIGH field. This makes a second recursive call of delete, but we do it *only* when the first call immediately returned without further recursion, because deletion from a one-valued set is an $\mathcal{O}(1)$ operation in the worst case. As a result we always make at most one nontrivial recursive call and thus we meet the stated criteria.

Finally, if the operation deletes the maximum value, it has to find the new one and store it in the MAX field. That is exactly symmetrical to finding the

alg. HVEBT(k).Insert(x) {inserts x into the tree, returns true on success}

```
{special cases}
if x = MIN then
    return false
if x < MIN then
    swap(x, MIN)    {continue inserting the former minimum}
else if x > MAX then
    MAX ← x

{the general case}
var (x_h, x_l) ← SplitBits(x)
var t ← LOW.Find(x_h)
if t ≠ null then
    return t.Insert(x_l)

{create a new subtree}
var u ← HVEBT(⌈k/2⌉).NewTree(x_l)
LOW.Insert(x_h, u)
if HIGH ≠ null then
    HIGH.Insert(x_h)
else
    HIGH ← HVEBT(⌊k/2⌋).NewTree(x_h)
return true
```

replacement for minimum, except that we don't delete the value. The pseudocode for Delete is shown on this page.

Similarly to insertion, the Delete algorithm only performs some constant work and at most three dictionary operations. One of them is deletion, which needs expected amortized constant time. Consequently, the whole algorithm needs expected amortized $\mathcal{O}(\log k)$ time.

---

alg. HVEBT(k).Delete(x) {deletes x from the tree, returns true on success}

```
{special cases}
if HIGH = null then {the tree contains one element}
    if x = MIN then
        this ← null   {destroying the structure}
        return true
    else
        return false
if x = MIN then {we find the new minimum and delete it from the subtree}
    var y_h ← HIGH.FindMin()
    var y_l ← LOW.Find(y_h).FindMin()
    MIN ← JoinBits(y_h, y_l)
    x ← MIN   {we continue deleting the new minimum}

{the general case}
var (x_h, x_l) ← SplitBits(x)
var t ← LOW.Find(x_h)
if t = null or t.Delete(x_l) = false then {x was not found in the structure}
    return false

{more special cases}
if t = null then {we emptied the subtree}
    LOW.Delete(x_h)
    HIGH.Delete(x_h)   {second recursion, but the first one was trivial}
if x = MAX then {we need to update the MAX field}
    var y_h ← HIGH.FindMax()
    var y_l ← LOW.Find(y_h).FindMax()
    MAX ← JoinBits(y_h, y_l)
return true
```

Conclusion 9. In HVEBT(k) containing n elements the operations NewTree, FindMin and FindMax take constant worst-case time. The operations Insert and Delete take $\mathcal{O}(\log k)$ amortized expected time. The operations Member, Predecessor and Successor take expected time $\mathcal{O}(\log k) = \mathcal{O}(\log \log N)$.

Space requirements

The original VEBT structure worked similarly, but HVEBT was rather inspired by a simplified modern implementation from Demaine's lecture notes [8] that only differs from HVEBT by using arrays as dictionaries instead of hash tables. Using arrays has the advantage that the structure isn't randomized and thus the time bounds are worst-case. On the other hand it causes that the memory consumption depends more on $N \equiv |U|$ than on $n \equiv |A|$, because only the array in the root node takes $\Theta\left(\sqrt{N}\right)$ space.

In the first publication about VEBT [26] an $\mathcal{O}(N \log\log N)$ space bound was shown and it was soon improved in [27] to $\Theta(N)$ by clustering the stored elements (we will use this technique in section 4.3). The clustering technique could also be used on Demaine's implementation, but we present a proof that even without it the structure only needs $\Theta(N)$ memory as well.

Lemma 10. *Let $S(N)$ denote the number of words used by the Demaine's VEBT structure (like HVEBT, but implemented with arrays as dictionaries) for the universe set of size $N = 2^k$. Then $S(N) \in \mathcal{O}(N)$.*

Proof. We will show by induction that $\exists c, d > 0\ \forall N \geq 2 :\ S(N) \leq cN - d$. We can suppose that $S(2)$ is a constant, so let us choose it as the base case. By the recursive construction for $N \geq 4$, the universe set is split into u segments of size v and $u, v \geq 2$. That gives us

$$S(N) \;=\; S(uv) \;=\; \underbrace{uS(v)}_{\text{subtrees}} + \underbrace{S(u)}_{\text{HIGH}} + \underbrace{\Theta(u)}_{\text{LOW, etc.}} \;\leq\; uS(v) + S(u) + eu$$

for some constant $e > 0$. In the construction we have $u, v < uv$, so we can use the induction hypothesis, resulting in

$$S(uv) \;\leq\; u(cv - d) + cu - d + eu \;=\; c(uv) - d + u(c - d + e)\,.$$

Now it suffices to satisfy $c - d + e \leq 0$ and the base case $S(2) \leq 2c - d$. That can be done by choosing $c := S(2) + e$ and $d := c + e = S(2) + 2e$. □

In this light it might seem that our HVEBT containing n elements only needs $\Theta(n)$ space thanks to using linear-space dictionaries, but that is *not* true.[6] The problem is when too many inputs get into the HIGH subtree (it can be as many as $n - 1$). Let us show a counter-example – define the set A_k of k-bit integers for some $k = 2^l$

$$A_k := \left\{1 \ll (k - 2^i)\right\}_{i=0}^{l} \quad \text{or equivalently} \quad A_k := \left\{2^{k-2^i}\right\}_{i=0}^{l}$$

[6] The scribed lecture notes from Demaine [8, 2003: lecture 2] claim that using hashing instead of arrays (that is essentially our HVEBT) results into a structure that only needs $\Theta(n)$ space. They also provide a sketch of a proof, but it is contradicted by our counter-example (there is a barely noticeable problem in their proof sketch).

Obviously we have $|A_k| = l + 1 = 1 + \log k$, $\min A_k = 1 \ll (k - 2^l) = 2^{k-2^l} = 1$. Moreover it holds

$$A_{2k} = \{1\} \cup \{x \ll k : x \in A_k\} = \{1\} \cup \{x2^k : x \in A_k\}.$$

Now consider storing A_k in HVEBT(k) for an arbitrary $k = 2^l \geq 2$. The minimal value 1 is stored exclusively and the rest of the inputs is bit-split in half. The lower halves are always zero (which is not interesting), but the upper halves make exactly the set $A_{k/2}$. In the root node we need $\Omega(l) = \Omega(\log k)$ space for the LOW field and we need to store the set $A_{k/2}$ in the HIGH field. That gives us $\sum_{i=l}^{1} \Omega(i) \subseteq \Omega(l^2)$ space for a set of size $l + 1 = 1 + \log k$, which is clearly not linear.

However, it is simple to show a little higher upper bound of $\mathcal{O}(n \log \log N)$. That tightly matches our counter-example, because there $n = l + 1 = 1 + \log \log N$.

Lemma 11. *The number of HVEBT nodes in a HVEBT(k) structure containing n elements is $\mathcal{O}(n \log k)$.*

Proof. By induction on the modification algorithms. HVEBT(k) containing one number has one node. The insertion algorithm can allocate at most one new node on every level of recursion. Since the depth of HVEBT(k) is $\mathcal{O}(\log k)$, one insertion can add at most $\mathcal{O}(\log k)$ nodes. Now it suffices to note that the shape of the structure is exactly given by the contents of the stored set, so even after performing Delete operations, the structure looks as if it was created by insertions only. As a result, any HVEBT(k) containing n elements has at most $\mathcal{O}(n \log k)$ nodes. □

Corollary 12. The number of words needed by a HVEBT(k) structure containing n elements is $\mathcal{O}(n \log k)$.

Proof. We know that dictionaries need space linear in the number of contained elements (Conclusion 6). Therefore, the space consumed by every LOW dictionary can be split into constant pieces and charged to the corresponding child nodes. Doing this in the whole HVEBT leaves every node with only a constant amount of space that is needed for the MIN and MAX fields and for the amount charged from the parent node. The number of nodes is bound by $\mathcal{O}(n \log k)$ according to the previous lemma, which completes the proof. □

Additional operations

There are many more operations that can be done efficiently on HVEBT structures (for example combination of several sets by a boolean formula), but here we will only need two additional algorithms – for building a new structure and for iterating its contents (implemented as listing all elements into a queue).

List operation Operation List outputs all contained elements in increasing order into a queue. In order to make the recursion work efficiently, we added an argument x_a – the value to be added to every output element. The algorithm is very simple, it first outputs MIN+x_a into the queue and then it creates a temporary queue and fills it by a recursive call on the HIGH subtree. All elements of the queue are taken in order and for every value x_h the corresponding subtree is found in the LOW dictionary and its contents is recursively listed directly into the output queue, using $x_a + \mathsf{JoinBits}(x_h, 0)$ as the additive element. Here the additive element is increased, because the elements in subtrees in LOW are missing the high bits and their value should be x_h (addition is used as an alternative method of joining the bit representations).

The pseudocode for List is shown on the current page. It is clear that the algorithm only needs expected time proportional to the space usage of the structure, because it touches every node exactly once, each operation on queues takes worst-case constant time and every lookup in a dictionary takes expected constant time.

---

alg. HVEBT(k).List(x_a, Q) {pushes all elements in increasing order into Q}
  Q.Push(x_a + MIN)
  if HIGH $\neq$ null **then**
    var $H \leftarrow$ "a new empty queue" {to contain all valid high bit halves}
    HIGH.List($0, H$)
    for all x_h **in** H **do**
      LOW.Find(x_h).List($x_a + \mathsf{JoinBits}(x_h, 0)$, Q)

Build operation The inverse operation Build for building a structure from a list of numbers is more complicated. It is essential here that the input is already in increasing order. Moreover, to lower the time complexity of the algorithm we do not try to find which parts of the input belong to individual subtrees, but we let the recursion find the part that fits in. An additional argument x_s is added containing the value to be subtracted from every element in the queue. When a value too big for the structure is encountered ($\geq 2^k$ after subtracting x_s), the build is stopped and the value remains the first in the queue.

The building starts by popping the first value of the queue into the MIN field (after subtraction of the argument), also the LOW field is initialized to an empty dictionary and a new empty queue Q_h is created to gather the future contents of the HIGH field.

Then the algorithm loops while the input queue is nonempty. In every iteration it first looks at the next value in the input queue, subtracts the argument x_s and computes x_h – the high half of the result's bit representation. That is the key of the next subset to build. If the value is too high the loop is broken, since the allowed portion of the input queue was consumed. Otherwise a subtree is built

recursively by passing the same input queue and a new value to subtract which is computed as $x_s + \mathsf{JoinBits}(x_h, 0)$ (here the recursion uses subtraction to cancel out an unknown number of starting bits). The newly built subtree is inserted with key x_h into the LOW dictionary and the key x_h is also pushed onto the Q_h queue. Then the loop continues.

After the loop stops (either by emptying the input queue or reaching a big value), it only remains to recursively build the HIGH structure from the Q_h queue and to find the maximum (the same as in the Delete operation). We show the pseudocode for Build on this page.

---

alg. HVEBT(k).Build(x_s, Q) {constructs a new structure}

```
MIN ← Q.Pop() − x_s
LOW ← "a new empty dictionary"
var Q_h ← "a new empty queue"    {to hold indices of subtrees in LOW}

{build the subtrees recursively}
while not Q.Empty() do
    var x ← Q.Top() − x_s    {examine the next element to store}
    if x ≥ (1 ≪ k) then {x is too big for this structure, end the loop}
        break
    var (x_h, x_l) ← SplitBits(x)
    var t ← HVEBT(⌈k/2⌉).Build( x_s + JoinBits(x_h, 0), Q )
    LOW.Insert(x_h, t)
    Q_h.Push(x_h)

{build the summary tree HIGH and find MAX}
if Q_h.Empty() then {no subtrees were created}
    HIGH ← null
    MAX ← MIN
else
    HIGH ← HVEBT(⌊k/2⌋).Build(0, Q_h)
    var x_h ← HIGH.FindMax()
    var x_l ← LOW.Find(x_h).FindMax()
    MAX ← JoinBits(x_h, x_l)
return this
```

Now we prove, similarly to List, that Build needs expected time proportional to the space complexity of the resulting structure. From the algorithm description it is clear that the time needed to build one vertex of the tree (not counting recursion) is given by the number of iterations of the while-loop. In every complete iteration we build one subtree and consume expected amortized constant extra time as we perform one insertion into a dictionary and several constant-time operations. Since we build the dictionary from scratch, the total expected time spent on insertions into the dictionary is given by the final number of its elements, which

is the number of subtrees in the LOW field. Now we can split the total time of iterating without recursion into equal (de-amortized) parts and charge one part (expected constant time) to the each child node. That will only leave constant-time work common to all iterations (the constants are independent of k). We apply the scheme recursively, so every vertex of the structure is at most charged expected constant time from its unique parent. That leaves at most expected constant time associated with every vertex of the tree, which completes the proof.

Conclusion 13. Using the Lemma 11, HVEBT(k) with n elements can perform the operations List and Build in expected $\mathcal{O}(n \log k)$ time.

The pair of operations List and Build is going to be very useful as a straightforward way of transforming a set stored by a HVEBT in expected $\mathcal{O}(n \log k)$ time. The contents is simply listed into a queue, there the elements are transformed and finally the structure is constructed from the result. We could write special algorithms to perform individual types of transformations directly on HVEBTs, but it would be less transparent and it wouldn't help the theoretical analysis either.

4.3 LVEBT

It is possible to reduce the space requirements from $\mathcal{O}(n \log k)$ to $\Theta(n)$ by a simple clustering trick without increasing the asymptotic times of mentioned operations. The clustering was originally used by van Emde Boas [27] and also in the first publication about the hashed VEBT [19]. We will denote this modified structure LVEBT to signify its linear space usage.

LVEBT(k) is composed of two levels. The contained numbers are grouped into $\Theta(n/\log k)$ clusters of size $\mathcal{O}(\log k)$ and the minimum of every cluster is stored in field TOP which is a HVEBT(k) structure. The implementation of the cluster structure can be, for example, a linked list – it can find the minimum and maximum in constant time and perform all other operations in linear time in the worst case. To be able to find clusters by their minimums, we can either associate every key in TOP structure with a pointer to the corresponding cluster structure, or we can maintain another dictionary mapping the minimums (or alternatively all elements) to the clusters.

When operating on LVEBT(k), we first find the right cluster by one or two Predecessor/Successor queries on the TOP field containing cluster minimums. Then we perform the desired operation inside the found cluster and finally, if the operation modified the cluster, we may have to correct the cluster size. Similarly to (a,b)-trees we choose parameters $a, b \in \Theta(\log k)$. When a cluster gets bigger than b, we split it. When it gets smaller than a, we move elements from its neighbour or we join it with its neighbour. As a consequence we might have to change the elements in TOP, but one operation on LVEBT(k) can only result in a constant number of operations on TOP. If we used a dictionary to map cluster minimums or all elements to the corresponding clusters, we also have to update the mapping,

but the number of affected clusters is $\mathcal{O}(1)$, so the number of required operations on dictionaries is $\mathcal{O}(\log k)$, which exactly fits into the $\mathcal{O}(\log k)$ expected time of HVEBT(k). To sum up, the asymptotic complexities of the predecessor problem operations remain the same as in HVEBT(k).

The advantage of maintaining a mapping from *all* elements to its cluster is that the Find operation can be implemented in $\Theta(1)$ expected time by only testing the existence of the mapping. However, that is no significant improvement – the asymptotic complexities of other operations are the same and the constants of modifying operations increase. Moreover, we can modify *any* structure for the predecessor problem in this way, as noted on page 21.

Operation List can be done in LVEBT(k) by calling TOP.List and listing the contents of the returned clusters. Similarly, for Build it suffices to create the clusters from consecutive blocks of b elements and use HVEBT(k).Build on the minimal values to create TOP. Using Conclusion 13, the running time of List and Build is clearly proportional to the total space usage of LVEBT(k), which is $\underbrace{\Theta(n)}_{\text{clusters}} + \underbrace{\mathcal{O}\left(\frac{n}{a}\log k\right)}_{\text{TOP field}} = \Theta(n)$.

Conclusion 14. In LVEBT(k) containing n elements the operations FindMin and FindMax take constant time in the worst case. The operations Find, Predecessor and Successor take expected time $\mathcal{O}(\log k) = \mathcal{O}(\log\log N)$. The operations Insert and Delete need expected amortized $\mathcal{O}(\log k)$ time. The operations List and Build consume $\Theta(n)$ time and the structure needs $\Theta(n)$ words of space.

4.4 An alternative

Willard [31] proposed a different approach of reducing the enormous space usage of original van Emde Boas trees. We will just briefly present their main ideas. When operating on a VEBT, the algorithms essentially first find the length of the longest prefix common to the query number and any contained number (by binary search on the bits), and then they perform some simple action to finish the operation. The *x-fast trie* works on the same principle, but it is realized in a different way than VEBT.

Binary trie of depth k is a natural way to store any set of k-bit numbers. A node in depth d represents a d-bit prefix and it is present iff there is at least one stored number with this prefix. As a special case, all leaves are in the depth k and they correspond one-to-one to the stored numbers. In x-fast tries, all present nodes of such a binary trie are grouped by depth and stored in dictionaries where the nodes are indexed by the binary prefixes they represent. There is no need to store any pointers between the nodes because every node can have at most two child nodes and their existence can be easily found by a lookup in the corresponding dictionary.

In this layout it is simple to binary-search for the longest present prefix common with the query by lookups in the dictionaries. This process finds the node of the trie where the path to the query number would fork away from the current trie and it takes $\Theta(\log k)$ lookups. To be able to search predecessors and successors quickly, all stored numbers are organized in a doubly linked list, where every number is referenced from its leaf in the trie. Moreover, every inner node that misses a son contains a direct pointer to the stored number that lies "closest" to the missing subtree. This pointed-to number is either the minimal or the maximal stored number with matching prefix (depends on whether the missing son is the left one or the right one) and it is also the predecessor or successor of the query, assuming the query number is not already stored in the trie. The other neighbour can be found by one step in the linked list.

One insertion in an x-fast trie of depth k can allocate at most k dictionary indices, so the space is bounded by $\mathcal{O}(nk)$. To reduce the bound to $\Theta(n)$, Willard uses clustering that is the same as the one used in LVEBT, except for the a and b bounds that need to be $\Theta(k)$. The resulting structure is called *y-fast trie*. Since the clusters can be represented by balanced trees, it is still possible to achieve $\mathcal{O}(\log k)$ time per operation inside the clusters and to maintain the overall $\mathcal{O}(\log k)$ time bound. Willard proposes to use a dynamic perfect hashing scheme for implementation of the dictionaries, so the bound for finding predecessors and successors is worst-case and the bound for modifying operations is expected amortized.

To sum up, the *y-fast trie* is an alternative to the LVEBT structure. When using the same type of dictionaries, it can achieve the same asymptotic bounds on time and space. The tries appear to be a little simpler, but we chose to use LVEBT as a substructure for our design in the following chapter before we knew that there already was a different solution.

5 Designing a new structure

When designing a new structure, we will require linear space usage and try to minimize the expected amortized time needed for operations from the predecessor problem. Our aims are the same as most of the research discussed in chapter 3, so we will try to use the same conventions to enable easier comparison.

We will focus on the model of μ-random insertions and random deletions, where μ is an unknown $(s^\alpha, s^{1-\delta})$-smooth distribution (see definitions on pages 11–12). The models for modification ensure that at any point of execution the stored set is μ-distributed, despite the insertions could be arbitrarily mixed with deletions. Then we will be able to use the "smoothness" of the stored set in a manner similar to the known data structures discussed in chapter 3.

We will make use of the LVEBT structure which managed to decrease the needed space to the desired bound. However, for sparsely populated universe sets its operation time of $\mathcal{O}(\log\log N)$ is too high. Thus the most important part of the design is to understand what the smoothness property ensures about the stored set, so the structure can utilise it to achieve a better performance in terms of n, the size of the represented set (contrary to N, the size of the universe set).

5.1 Static analysis of bucketing

Let μ be a $(s^\alpha, s^{1-\delta})$-smooth distribution for some $\alpha, \delta > 0$. The Definition 3 (smoothness) gives us

$$\exists\beta\ \forall c_1, c_2, c_3 \quad a \le c_1 < c_2 < c_3 \le b \quad \forall s \in \mathbb{N}$$
$$\Pr\left[X \in \left\langle c_2 - \frac{c_3 - c_1}{s^\alpha}, c_2 \right\rangle \,\middle|\, X \in \langle c_1, c_3\rangle\right] \;\le\; \frac{\beta s^{1-\delta}}{s} \;=\; \beta s^{-\delta}.$$

We choose to cover the whole range $\langle c_1, c_3\rangle := \langle a, b\rangle$. That removes the conditioning, because it is always fulfilled.

$$\exists\beta\ \forall c_2 \quad a < c_2 < b \quad \forall s \in \mathbb{N} \quad \Pr\left[X \in \left\langle c_2 - \frac{b-a}{s^\alpha}, c_2 \right\rangle\right] \;\le\; \beta s^{-\delta}$$

Now we consider dividing the range $\langle a, b\rangle$ into at least n^d equally-sized buckets for some parameter $d > 0$ that will be chosen later (n is the size of the set to be stored). That means the bucket index where a value belongs is given by its first $d\log n$ bits.

Note that this technique is different from the clustering used in LVEBT, on the other hand it is almost the same as splitting of bit representations in HVEBT. The clusters were created by the number of contained elements and thus the transformation decreased the size of the stored set, resulting in lower space usage. Bucketing, however, groups elements with many common starting bits, so it decreases the size of the universe set instead. That is essential, because we need to decrease the query time, which only depends on universe size in van Emde Boas tree variants.

If we choose c_2 as the endpoint of an arbitrary[1] bucket B and choose $s := \lfloor n^{d/\alpha} \rfloor$, then $s^\alpha = \lfloor n^{d/\alpha} \rfloor^\alpha \leq n^d$ and thus the probability covers at least the whole bucket. Let us denote $p_B \equiv \Pr[X_\mu \text{ falls into the bucket } B]$, then we get

$$p_B \;\leq\; \beta s^{-\delta} \;\leq\; \beta \lfloor n^{d/\alpha} \rfloor^{-\delta} \;\leq\; \beta \left(n^{d/\alpha} - 1\right)^{-\delta}.$$

Since $\lim_{n\to\infty} \frac{(n^{d/\alpha})^{-\delta}}{(n^{d/\alpha}-1)^{-\delta}} = 1$, we have $p_B \in \mathcal{O}(n^{-\delta\frac{d}{\alpha}})$.

In our random process the elements are taken independently, so the number of elements in B is given by the binomial distribution. If we use μ to generate n values in $\langle a, b\rangle$, then

$$n_B \equiv \mathrm{E}[\text{number of values in } B] \;=\; n \cdot p_B \quad\to\quad n_B \in \mathcal{O}(n^{1-\delta\frac{d}{\alpha}}).$$

By choosing $d \geq \frac{\alpha}{\delta}$ we ensure that $n_B \in \mathcal{O}(1)$, because $\alpha, \delta > 0$. We summarize our analysis in the following lemma.

Lemma 15. *Let us generate n values by a $(s^\alpha, s^{1-\delta})$-smooth distribution for some $\alpha, \delta > 0$, and divide the whole range into at least $n^{\alpha/\delta}$ equal-sized buckets. Then the expected number of elements in an* arbitrary *bucket is $\mathcal{O}(1)$.*

The number of elements in a bucket is expected to be constant, but it can be bounded much stronger. For example, choosing $d \geq 2\frac{\alpha}{\delta}$ and using Chernoff bounds [21, chapter 4.1] would moreover guarantee that the bad behaviour is very rare. We don't do this finer analysis, because there is a more pressing problem that we do not attempt to solve in this text – the unrealistic character of the used models. We use them for our analysis to enable comparison with known structures, as we only found one published structure that uses different models (discussed in section 3.6 on page 17).

[1] There is a technical difficulty with the last bucket, because the definition (taken from referred papers) doesn't allow us to choose $c_2 = b$ for some unknown reason. However, we can choose $c_2 := b - \epsilon$ for such an ϵ that the interval covers the whole bucket except for the maximal value. Adding one value to the last bucket separately then doesn't affect the validity of the implied Lemma 15.

5.2 An amortized scheme

In the analysis we found out that in our model the first $\Theta(\log n)$ bits are likely to be an almost unique identifier among n elements. We will use k to denote this number of significant bits. We are going to use van Emde Boas tree (LVEBT) to represent the k starting bits, which will give us $\mathcal{O}(\log\log n)$ expected amortized time for any operation, but we still need to solve two problems. One of them is the fact that k needs to change during the life of the structure according to n to satisfy Lemma 15. The other problem is that we want the structure to work efficiently without knowing the parameters α and δ.

To remove dependency on the distribution's parameters, we simply choose a growing function instead of the factor $\frac{\alpha}{\delta}$. The time complexity of van Emde Boas trees is so low that we can afford to use a $\Theta(\log n)$ function, giving us $k \in \Theta(\log^2 n)$ significant bits. More precisely, we will maintain $k \in \left(\frac{1}{2}\log^2 n, 2\log^2 n\right)$. When the structure is small ($\frac{1}{2}\log n < \frac{\alpha}{\delta}$), we might not fit into Lemma 15, but that is only for inputs of constant size ($n < 2^{2\alpha/\delta}$). It follows that the asymptotic complexities aren't affected and the expected size of any nonempty bucket is still $\mathcal{O}(1)$.

Note that this kind of averaging for the size of buckets is different from the one generated by our dictionaries. In the bucketing, similarly to all methods in chapter 3, there exist bad sets of input that always fall into the same bucket and thus they always cause the structure to perform slowly. On the other hand, in our dictionaries we avoided such dependency on the input by utilising universal hashing in the implementation. In this way the dictionary operations were made to work in $\Theta(1)$ time on average for *any set* of input values, only considering the average over the (pseudo)random bits.

To change k we rebuild the structure periodically. We will always choose k as a power of two and whenever $\log^2 n$ reaches a different power of two, we rebuild the whole structure with the new k. That will exactly ensure that we keep k in the desired interval.

The rebuild clearly happens when the value of $\log^2 n$ changes from $\log^2 n_0$ to $2\log^2 n_0$ or $\frac{1}{2}\log^2 n_0$, that is when n changes from n_0 to $n_0^{\sqrt{2}}$ or $n_0^{1/\sqrt{2}}$. It follows that during a rebuild, the number of modifying operations since the last rebuild is $\Omega(n)$, where n is the current number of contained values.

We suppose that the rebuilding time is at most $\mathcal{O}(n)$-times larger than the time needed for modifying operations, because otherwise we could simulate the rebuild by $\mathcal{O}(n)$ FindMin, Delete and Insert operations. It follows that the rebuilding time can be amortized into additional time that is asymptotically at most equal to the modification time. As a consequence we can charge the rebuilding time to preceding modifying operations in a way that the expected amortized asymptotic times of modification operations aren't affected by the spread-out price for periodical rebuilds.

5.3 Implementing the structure

In this section we describe in more detail a possible implementation of the proposed amortized structure. We will denote the new structure storing l-bit numbers as SVEBT(l) where "S" stands for "smooth" (the used model of input distribution). The bucketing used in SVEBT is like the HVEBT decomposition, only the bit representations of the stored numbers are split in a different position.

Definition 16. Let $A \subseteq U \equiv \{0, \ldots, 2^l - 1\}$. The **SVEBT**($l$) structure storing the set A contains:

- SIZE – the current size of the stored set (SIZE $= n \equiv |A|$)
- K – the current number of significant bits. K is always a power of two and it changes to $\log^2 \text{SIZE}$ whenever the expression reaches another power of two. More precisely, we maintain an invariant $\frac{1}{2} < \frac{\log^2 \text{SIZE}}{\text{K}} < 2$. For the border case where SIZE < 2 we define K $= 0$.
- LOW – a dynamic dictionary indexed by keys from $\{0, \ldots, 2^{\text{K}} - 1\}$ containing nonempty bucket structures as values.
- HIGH – an LVEBT(K) structure containing exactly the set of indices of nonempty buckets from the LOW dictionary.

Similarly to the HVEBT decomposition, the stored set is split into subsets by the values of high K bits of numbers' bit representations. Every subset is stored in a bucket structure which is stored as a value in the LOW dictionary and it is identified by the common high K bits as the key. Like in HVEBT, the numbers in bucket structures have their high K bits zeroed-out (although it wouldn't be necessary here). The structure in the HIGH field again represents the ordering of the subsets by their keys.

The bucket structure

All the substructures are used as black boxes, so they can be replaced by anything else that provides the required operations. We need the bucket structure to provide the standard predecessor-problem operations. We also use the additional List and Build operations, but it is sufficient to simulate them by the standard operations as discussed in the previous section 5.2. There we also showed that the expected size of any bucket is $\mathcal{O}(1)$, so even if the implementation needed linear time in the number of elements in the bucket, the expected complexity of the operations would still be $\mathcal{O}(1)$.

We also require that the bucket structure takes space linear in the number of contained elements. That will ensure that all the buckets together take $\mathcal{O}(n)$ space (where n is the size of the whole set stored in SVEBT), because every contained element belongs into exactly one bucket and the empty buckets are not stored.

Together with linear space usage of the LVEBT and dictionary structures in the HIGH and LOW fields, we get that the total space needed by SVEBT is $\mathcal{O}(n)$ in the worst case.

The most important parameter affected by the implementation of buckets is the amortized time of the operations (not expected), because for a sufficiently large n, all n inputs can get into one bucket. Consequently, the amortized performance of SVEBT is the same as the amortized performance of the bucket structure. Therefore it is best to use Andersson's dynamization of the static structure from Beame and Fich which has an asymptotically optimal dependency of amortized (or worst-case) time on n while using $\mathcal{O}(n)$ space. Their structure and the possibility of dynamization were briefly mentioned in section 2.2 on page 8, but we won't discuss it in more depth as it is very complicated. Anyway, in practice it would probably be better to use some more lightweight structure for the buckets. For example, various balanced trees offer a good tradeoff between simplicity and the $\mathcal{O}(\log n)$ worst-case performance.

Operations on SVEBT

The algorithms for operations on SVEBT are almost the same as those on HVEBT due to the same nature of decomposition. Thus we only describe the differences.

The bit representations in SVEBT aren't split in halves, but into K most significant bits and the remaining bits. The SVEBT structure don't have to care about the minimum and maximum value (these aren't needed to be maintained, although they could be), but in addition the modifying operations need to maintain the current size of the stored set. That together makes a constant time difference per operation in the worst case.

It follows that when performing the standard operations from the predecessor problem, the SVEBT algorithms do some constant work plus at most a constant number of dictionary operations plus at most one nontrivial call to the LVEBT(K) structure in the HIGH field or to one of the bucket structures. Since the expected size of any bucket is constant and the dictionary operations take expected constant or expected amortized constant time, the asymptotic times needed for operations on SVEBT are given by the the times required by the corresponding operations on LVEBT(K) which is $\mathcal{O}(\log \mathrm{K}) = \mathcal{O}\big(\log(\log^2 n)\big) = \mathcal{O}(\log \log n)$ (according to Conclusion 14).

The additional operations List and Build are also little affected. When performing List, the indices of the buckets are first listed from the HIGH field into a temporary queue, then the buckets are listed in order into the output (adding some value to the numbers as in HVEBT). Building first finds the size of the set and computes the corresponding K value. Then all buckets are constructed in order and inserted into the LOW dictionary, storing the keys into a temporary queue at the same time. Finally all the keys are passed to the Build method for the LVEBT(K) structure and stored in the HIGH field.

The last significant change to consider is the periodical rebuilding. During the modifying operations Insert or Delete, when the value of $\log^2$ SIZE reaches another power of two than K, the whole SVEBT structure is rebuilt. Using the additional operations, the whole contents is listed into a queue and an SVEBT with the new K is built from scratch. In the previous section we already showed that the time needed for rebuilding can be amortized into the modifying operations without any impact on the asymptotic performance.

Conclusion 17. We combined LVEBT with bucketing to obtain a new structure for the predecessor problem. The SVEBT structure needs $\Theta(n)$ words of memory. When following the model of μ-random insertions and random deletions for unknown $\alpha, \delta > 0$ and an unknown $(s^\alpha, s^{1-\delta})$-smooth distribution μ, the time bounds are as follows. The Member, Successor and Predecessor operations need expected time $\mathcal{O}(\log\log n)$. The operations Insert and Delete need expected amortized $\mathcal{O}(\log\log n)$ time. When using the amortized structure of Beame and Fich in buckets [4, Corollary 4.6], the operations need $\Theta\left(\sqrt{\log n / \log\log n}\right)$ worst-case or amortized time (depending on whether they change the stored set).

In practice, the rebuilding wouldn't have to be done from scratch, but some parts of the structure could be reused. That wouldn't affect the asymptotic performance, but it would probably still be very useful for a real implementation. We can utilize the fact that K is *always* doubled or halved, which correlates with the bit-splitting decomposition in the HVEBT structure. The buckets in LOW will be split or merged, but these will mostly be one-bucket–to–one-bucket transformations. Consequently, the clusters in LVEBT in the LOW field will mostly stay the same and thus only the number of bits of their minimums will change. That is, the numbers in LOW.TOP will likely only change their bit lengths. As they are stored in an HVEBT structure, the parts can be directly reused. When doubling, the LOW.TOP becomes the new LOW.TOP.HIGH and similarly, when halving, the LOW.TOP.HIGH becomes the new LOW.TOP. This was only an idea of where we can save some work – we don't discuss it further, because obviously it wouldn't be useful for the time or space analyses and we feel that the design of SVEBT isn't yet a structure directly usable in practice, because its design was driven by a very unrealistic model of smooth distributions and independent modifying operations.

5.4 Comparison with known structures

The properties of the proposed structure are very similar to the *Augmented Sampled Forest* and the structure by Kaporis et al. that were discussed in chapter 3. For example, Kaporis et al. also use bucketing to decrease the size of the universe, but the top level is different and they also regulate the bucket sizes instead of the depth.

Both of the published structures also achieve the same $\mathcal{O}(\log\log n)$ expected amortized time, but they need μ to be from the class of $\left(s/(\log\log s)^{1+\epsilon}, s^{1-\delta}\right)$-smooth distributions for some $\delta > 0$. That is a subset of the $(s, s^{1-\delta})$-smooth class. The proposed SVEBT structure can handle $(s^{\alpha}, s^{1-\delta})$-smooth distributions for an arbitrary α (even $\alpha > 1$) and it still needs only $\Theta(n)$ words of space and expected $\mathcal{O}(\log\log n)$ time per operation.

On the other hand, the proposed structure is in no way ideal. Here we didn't try to construct a de-amortized version of the structure, although it seems that it could be achieved by first de-amortizing the dictionaries by the global rebuilding technique and then (like in ASF) using the technique on the whole SVEBT structure.

It would also be better if the structure was able to adapt to the distribution of the input. We made SVEBT robust, so it doesn't have to know the α and δ parameters, but it also doesn't achieve better performance for easy distributions. The ASF structure needed to know the parameters of distribution's class in advance (or during rebuild), which allowed expected constant performance on bounded distributions (including the uniform distribution).[2] The structure of Kaporis et al. didn't need to know them, but the are some unresolved problems in using the q*-heaps, so we don't feel sure that the stated complexities really hold.

[2] We could similarly use prior knowledge of the α and δ parameters to choose to build a structure that is most suitable for them (for example ASF or SVEBT). As noted by Andersson, such choice can also be done on the start of every rebuild, based on the knowledge of previous input data.

6 Conclusion

We devoted this thesis entirely to the predecessor problem. In the textbook comparison model it has simple asymptotically optimal solutions, but the more realistic word-RAM model makes the problem much more interesting. The worst-case behaviour on word-RAM is also understood quite well, thanks to the combined effort of Beame, Fich and Andersson. By a combination of lower bound proofs and a complicated data structure they managed to bound the difficulty of the problem. Now we know that the most interesting part of the bound (the part that only depends on n) is asymptotically tight and it is equal to $\Theta\left(\sqrt{\log n / \log\log n}\right)$.

Since the field of expected-time bounds seemed to be less explored, we chose it as the primary focus for our research. We studied past approaches to solve the problem. They began with searching uniformly distributed data in sorted arrays and over time they allowed to generalize the distribution and to add modifying operations. Using the ideas, we proposed a new structure tailored specially for the same model of "smooth" input. The new structure performs the predecessor-problem operations in the same expected amortized $\mathcal{O}(\log\log n)$ time, but the bound is proven from a *weaker* condition on smoothness of the input distribution than the published structures (we don't need that $\alpha < 1$). For a substructure we chose a randomized modification of a structure by van Emde Boas. To make the properties clear, we described in detail how to modify the structure's design to decrease the needed space and we analyzed the complexities.

As we discussed at the end of the previous chapter, the SVEBT structure still has many directions in which it could be improved. Another problem is that the models used for expected-time analyses are very unrealistic. We adopted them from the published research because there seemed to be no better alternative. Perhaps it would be better to use a different approach, for example some models from the theory of approximation algorithms could be used. Then we might be able to prove, that some solution on the word-RAM is at most x-times worse than any other solution.

Bibliography

[1] Arne Andersson and Christer Mattsson. Dynamic interpolation search in $o(\log \log n)$ time. In *ICALP '93: Proceedings of the 20th International Colloquium on Automata, Languages and Programming*, pages 15–27, London, UK, 1993. Springer-Verlag.

[2] Arne Andersson and Mikkel Thorup. Dynamic string searching. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 307–308, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[3] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.

[4] P. Beame and F.E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[5] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem. In *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 295–304, New York, NY, USA, 1999. ACM.

[6] Svante Carlsson and Christer Mattsson. An extrapolation on the interpolation search. In *No. 318 on SWAT 88: 1st Scandinavian workshop on algorithm theory*, pages 24–33, London, UK, 1988. Springer-Verlag.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[8] Erik Demaine. Advanced data structures. MIT Lecture Notes, 2003–2010. Retrieved from `http://www.erikdemaine.org`.

[9] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. Interpolation search for non-independent data. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 529–530, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[10] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F.M. auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:524–531, 1988.

[11] Greg N. Frederickson. Implicit data structures for the dictionary problem. *J. ACM*, 30(1):80–94, 1983.

[12] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[13] Gaston H. Gonnet, Lawrence D. Rogers, and J. Alan George. An algorithmic and complexity analysis of interpolation search. *Acta Inf.*, 13:39–52, 1980.

[14] Torben Hagerup. Sorting and searching on the Word RAM. In *STACS '98: Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398, London, UK, 1998. Springer-Verlag.

[15] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 417–431, London, UK, 1981. Springer-Verlag.

[16] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. Dynamic interpolation search revisited. In *33rd International Colloquium for Automata, Languages and Programming (ICALP 2006)*, volume 4051 of *Lecture Notes in Computer Science*, pages 382–394. Springer, July 2006.

[17] Alexis Kaporis, Christos Makris, Spyros Sioutas, Athanasios Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. Dynamic interpolation search revisited. Technical Report TR 2006/04/02, Computer Technology Institute, April 2006.

[18] D. E. Knuth. Deletions that preserve randomness. *IEEE Trans. Softw. Eng.*, 3(5):351–359, 1977.

[19] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $\mathcal{O}(\log\log n)$ time and $\mathcal{O}(n)$ space. *Inf. Process. Lett.*, 35(4):183–189, 1990.

[20] Kurt Mehlhorn and Athanasios Tsakalidis. Dynamic interpolation search. *J. ACM*, 40(3):621–634, 1993.

[21] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[22] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*, chapter 5. Springer-Verlag, 1983.

[23] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a $\log\log n$ search. *Commun. ACM*, 21(7):550–553, 1978.

[24] Yehoshua Perl and Edward M. Reingold. Understanding the complexity of interpolation search. *Inf. Process. Lett.*, 6(6):219–222, 1977.

[25] W. W. Peterson. Addressing for random-access storage. *IBM J. Res. Dev.*, 1(2):130–146, 1957.

[26] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *SFCS '75: Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, Washington, DC, USA, 1975. IEEE Computer Society.

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

[28] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[29] Dan E. Willard. Searching unindexed and nonuniformly generated files in $\log \log n$ time. *SIAM Journal on Computing*, 14(4):1013–1029, 1985.

[30] Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000.

[31] D.E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[32] Andrew C. Yao and F. Frances Yao. The complexity of searching an ordered random table. In *SFCS '76: Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 173–177, Washington, DC, USA, 1976. IEEE Computer Society.