Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS

Milan Straka

Functional Data Structures and Algorithms

Computer Science Institute of Charles University

Supervisor of the thesis: doc. Mgr. Zdeněk Dvořák, Ph.D.
Study programme: Computer Science
Specialization: Discrete Models and Algorithms (4I4)

Prague 2013

I am grateful to Zdeněk Dvořák for his support. He was very accommodative during my studies. He quickly discovered any errors in my early conjectures and suggested areas of interest that prove rewarding.

I would also like to express my sincere gratitude to Simon Peyton Jones for his supervision and guidance during my internship in Microsoft Research Labs, and also for helping me with one of my papers. Our discussions were always very intriguing and motivating.

Johan Tibell supported my work on data structures in Haskell. His enthusiasm encouraged me to overcome initial hardships and continues to make the work really enjoyable.

Furthermore, I would like to thank to Michal Koucký for comments and discussions that improved the thesis presentation considerably.

Finally, all this would not be possible without my beloved wife and my parents. You make me very happy, Jana.

Název práce: Funkcionální datové struktury a algoritmy

Autor: Milan Straka

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí doktorské práce: doc. Mgr. Zdeněk Dvořák, Ph.D, Informatický ústav Univerzity Karlovy

Abstrakt: Funkcionální programování je rozšířené a stále více oblíbené programovací paradigma, které nachází své uplatnění i v průmyslových aplikacích. Datové struktury používané ve funkcionálních jazycích jsou převážně perzistentní, což znamená, že pokud jsou změněny, zachovávají své předchozí verze. Cílem této práce je rozšířit teorii perzistentních datových struktur a navrhnout efektivní implementace těchto datových struktur pro funkcionální jazyky.

Bezpochyby nejpoužívanější datovou strukturou je pole. Ačkoli se jedná o velmi jednoduchou strukturu, neexistuje jeho perzistentní protějšek s konstantní složitostí přístupu k prvku. V této práci popíšeme zjednodušenou implementaci perzistentního pole s asymptoticky optimální amortizovanou časovou složitostí $\Theta(\log\log n)$ a především téměř optimální implementaci se složitostí v nejhorším případě. Také ukážeme, jak efektivně rozpoznat a uvolnit nepoužívané verze perzistentního pole.

Nejvýkonnější datové struktury nemusí být vždy ty, které jsou založeny na asymptoticky nejlepších strukturách. Z toho důvodu se také zaměříme na implementaci datových struktur v čistě funkcionálním programovacím jazyku Haskell a podstatně zlepšíme standardní knihovnu datových struktur jazyka Haskell.

Klíčová slova: perzistentní datové struktury, perzistentní pole, algoritmy se složitostí v nejhorším případě, čistě funkcionální datové struktury, Haskell

Title: Functional Data Structures and Algorithms

Author: Milan Straka

Institute: Computer Science Institute of Charles University

Supervisor of the doctoral thesis: doc. Mgr. Zdeněk Dvořák, Ph.D, Computer Science Institute of Charles University

Abstract: Functional programming is a well established programming paradigm and is becoming increasingly popular, even in industrial and commercial applications. Data structures used in functional languages are principally persistent, that is, they preserve previous versions of themselves when modified. The goal of this work is to broaden the theory of persistent data structures and devise efficient implementations of data structures to be used in functional languages.

Arrays are without any question the most frequently used data structure. Despite being conceptually very simple, no persistent array with constant time access operation exists. We describe a simplified implementation of a fully persistent array with asymptotically optimal amortized complexity $\Theta(\log\log n)$ and especially a nearly optimal worst-case implementation. Additionally, we show how to effectively perform a garbage collection on a persistent array.

The most efficient data structures are not necessarily based on asymptotically best structures. On that account, we also focus on data structure implementations in the purely functional language Haskell and improve the standard Haskell data structure library considerably.

Keywords: persistent data structures, persistent arrays, worst-case algorithms, purely functional data structures, Haskell

Contents

Chapter 1

Introduction

Computer programming has been developing enormously ever since the first high-level languages were created,[1] and several fundamental approaches to computer programming, i.e., several programming paradigms, have been designed. The prevalent approach is the *imperative programming paradigm*, represented for example by the wide-spread C language.

The imperative paradigm considers a computer program to be a sequence of statements that change the program state. In other words, serial orders (imperatives) are given to the computer.

The *declarative programming* represents a contrasting paradigm to the imperative programming. The fundamental principle of declarative approach is describing the problem instead of defining the solution, allowing the program to express *what* should be accomplished instead of *how* should it be accomplished. The logic of the computation is described without dependence on control flow, as opposed to the imperative programing, where the control flow is a fundamental part of any program.

One of the well established way of realizing the declarative paradigm is functional programming.

1.1 Functional programming

Functional programming treats computations as evaluations of mathematic functions and the process of program execution is viewed as application of functions instead of changes in state.

[1]FORTRAN, “Formula Translator”, released in 1954, is considered the first high-level language with working implementation.

Referential Transparency

A major difference between functional and imperative programming is absence of side effects that change global program state. Function has a side effect, if, in addition to returning a value, it irreversibly modifies some global state or has an observable effect on the outside world, like displaying a message on a screen. Side effects are common in imperative programming, while in functional programming, output of a function depends solely on the input arguments and not on an internal state of a program. Therefore, calling a function twice with the same arguments produces the same result. Functional programs are *referentially transparent*, meaning that a function can be replaced by its resulting value without changing the behaviour of the program.

The functional languages that completely lack the side effects are usually called *purely functional* and considered declarative. Because purely functional language does not define a specific evaluation order, various evaluation strategies are possible. One of the most theoretically and practically interesting strategies is *lazy evaluation*. Under lazy evaluation, expressions are not evaluated immediately, but their evaluation is delayed until their results are needed by other computations. This contrasts with the standard strategy, also called eager evaluation, where expressions are evaluated when occurring.

A purely functional language compiler can also rewrite the programs substantially while preserving the semantics and can therefore introduce substantial optimizations,[2] that would be very difficult to perform on programs with side effects.

The absence of side effects and possibility to choose evaluation strategy is particularly suitable for developing parallel programs, because the program parts are independent except for explicitly marked dependences. This independence even allows parallel execution of programs designed as sequential, although the possibilities of parallel execution may be limited.

In addition, referential transparency simplifies reasoning about computer programs to the level of making formal proofs of program correctness practical.

Nevertheless, most functional languages do allow side effects. In that case, some program parts exhibit side effects and have specific evaluation order, i.e., utilize the imperative programming paradigm, usually to perform input and output operations like drawing to a screen, sending data over the network or reading keyboard input. The other program parts are side effect free and benefit from all

[2] An example of an useful optimization is deforestation (also known as fusion), which is a program transformation eliminating intermediate data structures.

the advantages of functional programming.

Apart from referential transparency, there are two particular additional characteristics of functional programming.

First-Class Functions

Functions behave like ordinary values in functional programming. They can be passed as arguments to other functions, which is a frequently mentioned functional programming feature. Functions can also be assigned to variables or stored in data structures. Finally, functional programming allows new function creation and function composition in a straightforward way.[3]

The higher-order functions, i.e., functions taking other functions as arguments, together with lazy evaluation have a huge impact on modularity, an issue described in great detail in [Hug89]. This work demonstrates that higher-order functions offer a great level of generality, allowing to express algorithms that can be specialized suitably in every situation. Lazy evaluation enables composing functions effectively, performing only computations that are really needed, without any additional programming effort. Proper usage of these idioms result in small and more general modules, that can be reused widely, easing subsequent programming.

Type Systems

Many functional languages are based on a typed lambda calculus, especially since the development of the Hindley-Milner type inference algorithm [DM82], because of its completeness and ability to automatically infer the most generic type. The strong type checking performed by the compiler prevents many errors and the automatic type inference frees the programmer from specifying a type for every binding.

1.1.1 Haskell

Haskell [PJ$^+$03] is a purely functional language with lazy evaluation. Although no side effects are allowed, specified program parts can behave imperatively, ex-

[3]Passing a function as arguments to other functions is undoubtedly an useful feature. However, although it is provided by nearly all imperative languages (e.g., via function pointers in C), it is not so useful without other function manipulation operations. Recently, imperative languages draw inspiration from functional programming and allow additional function manipulations, for example C++11 and C# 3.0.

ecuting the input and output actions sequentially. This is achieved by using monads [Wad90a, Wad92].

Even though Haskell utilizes lazy evaluation, Haskell programs achieve high performance and several large-scale projects are implemented in Haskell, for example a revision control system (darcs), several web servers and frameworks (happstack, snap, warp, yesod), tiling window manager (xmonad) and many Haskell compilers.

The Haskell language is standardized, the most widespread version is Haskell 98 [PJ$^+$03], which has been revised recently as Haskell 2010 [Mar10]. Although there are several Haskell compilers, one of them, GHC, the Glasgow Haskell Compiler, is most widely used and offers most features.

We chose Haskell as the functional language to use in this thesis, because it is one of the most used functional languages and despite being purely functional and providing advanced functional programming techniques like lazy evaluation, it has decent performance.

1.1.2 Purely Functional Data Structures

Purely functional data structures are data structures implemented in a purely functional language, therefore, without any side effects. Such data structures are *immutable*, i.e., it is not possible to change any existing value in the data structure, because overwriting memory is a side effect and thus not allowed in a purely functional language.

Purely functional data structures are usually implemented using algebraic data types, which are possibly recursive sum types of product types. In other words, an algebraic data type consists of several alternatives, usually called *constructors*, and each constructor is a product type containing several values, called *fields*.

A frequently appearing example of algebraic data type is the singly linked list. A singly linked list is either empty or consists of the first element and the rest of the list. This type can be defined in Haskell as follows:

```
data List a = Nil | Cons a (List a)
```

Operations on algebraic data types are defined using *pattern matching*, which allows inspecting both constructors and fields of a given type. For illustration, Haskell implementation of a method computing sum of list elements follows:

```
sum Nil = 0
sum (Const head tail) = head + sum tail
```

Many purely functional data structures exist with the same time complexity as their imperatively implemented counterparts, for example singly linked lists, stacks, queues, balanced binary search trees, random access lists and priority queues. [Oka99] describes a wide range of purely functional data structures and advanced techniques.

Purely functional data structures usually have higher space complexity compared to the imperative implementations, because they avoid overwriting existing values. On the other hand, these structures are immutable. Immutable structures are inherently thread-safe, i.e., can be used from multiple threads without any synchronization, because no thread can overwrite any parts of the structure which might be shared. In addition, immutable structures are usually conceptually simpler and offer higher security than structures where overwriting is allowed.

The most frequently used data structure is undoubtedly an array. Unfortunately, no purely functional array implementation exists which would have constant time complexity of operations. We deal with functional arrays in a great deal in this thesis, devoting Chapters 5 and 6 to the problem of functional arrays.

1.2 Persistent Data Structures

Interestingly, data structures used in purely functional programs do not necessarily need to be implemented in a purely functional way. It is sufficient for a structure to behave as if it was never modified, even though the implementation may utilize assignments and modify existing values. This leads to the notion of persistence.

A data structure is *persistent*, if it preserves the previous version of itself when modified. The structure is *partially persistent*, if only the newest version can be modified and all versions of the structure accessed. If all versions of the structure can be modified and accessed, the data structure is *fully persistent*.

Versions of a fully persistent data structure form a rooted tree, where the root is the initial version of the structure and an edge denotes that a version was created from its predecessor. In case of partially persistent structures, versions form a directed path.

Versions of a persistent data structure can be identified in several ways. One possibility, prevalent in purely functional data structures, is to consider each version to be an independent structure. This way, no explicit identification of a version is needed, but on the other hand, it is complicated to recognize structure parts shared by multiple versions. Therefore, the structure versions are usually immutable in this case. The other possibility, quite common in imperative implementations of persistent structures, is to represent all versions using one shared data structure and identify the versions using a usually integral key, possibly accompanied by access pointers of the structure (i.e., a tree root or a list head) or other data. In this manner, sharing of parts of the structure is explicit, but the implementation is imperative, overwriting parts of the shared data structure when a new version is created.

Any fully persistent data structure can be used in a purely functional language, because modifying the structure does not cause any observable side effects. Nevertheless, persistence is useful on its own accord. Consider the classic problem of planar point location, where the plane is subdivided into polygons by n line segments intersecting only at their endpoints. Given a sequence of query points in the plane, the goal is to determine for every query point the polygon which contains it.

The planar point location can be solved efficiently using persistent search trees, as demonstrated by [ST86]. When we draw a vertical line through every vertex, we divide the plane into vertical slabs. Every slab consists of regions separated by non-intersecting lines, which are linearly ordered. If we store the intersecting lines of a slab in a binary search tree, we can locate the region of the slab containing the query point in $\mathcal{O}(\log n)$ time. Therefore, if we represent each slab using a binary search tree and we also create a binary search tree containing the slab boundaries, we can locate a polygon containing the query point in $\mathcal{O}(\log n)$ time. An example of such construction is illustrated in Figure 1.1.

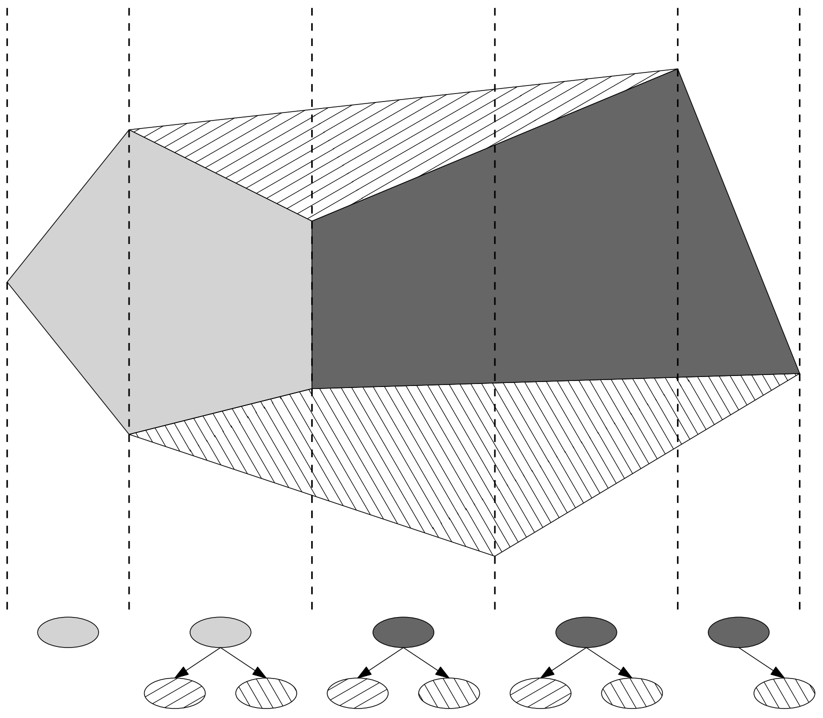


Figure 1.1: Planar location using partially persistent search tree

The problematic part of the described solution is that each slab can contain $\mathcal{O}(n)$ regions, so the whole construction could require $\mathcal{O}(n^2)$ space. However, the persistence comes to the rescue. Instead of constructing the slabs independently, we create the leftmost slab first and then construct further slabs incrementally, using partially persistent search trees. Because every line segment is added and removed at most once, constructing all slabs require only n insertions and n deletions in the partially persistent structure. The authors also describe a partially persistent variant of red-black trees, which perform insertions and deletions in $\mathcal{O}(\log n)$ time and $\mathcal{O}(1)$ space. Therefore, the whole preprocessing needs $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space and queries can be performed in $\mathcal{O}(\log n)$.

Apart from using persistent structures in functional languages and the described planar point location, there are numerous other applications of persistence, for example in computational geometry [Cha85, Col86, DM85, Ove81a, Ove81b], during object oriented programming inheritance computation [Die89], in tree pattern matching [CPT92] and when implementing continuations in functional programming languages [DST94].

Naturally, persistent structures are beneficial in search programs maintaining data structures while exploring trees, for example game tree searches. Fully persistent modifiable lists and sorted set are also often used, for instance in text editing [RTD83a, Mye84]. In addition, fully persistent structures are utilized in simulations detecting data races or deadlocks.

Persistent data structures can further be used in archival databases or revision control systems (the database of Git [Loe09], a widespread revision control system, is in fact an immutable fully persistent data structure).

1.2.1 Worst-Case and Amortized Complexity

There are two different ways how to measure complexity of an operation, assuming we are performing a whole sequence of such operations.

- We can measure the complexity of the slowest operation possible, i.e., the *worst-case complexity.*
- We can measure the "average" complexity of an operation. Specifically, if n operations take time $\mathcal{O}(f(n))$, we say that the *amortized complexity* of an operation is $\mathcal{O}(f(n)/n)$.

The amortized complexity is obviously never larger than the worst-case complexity. For several algorithms, the amortized complexity can even be asymptotically smaller than the worst-case complexity. Consider the well-known disjoint set union-find data structure [Tar83], which keeps track of a partition of a set into disjoint subsets and allows joining two subsets into a single subset. Blum [Blu86] showed that the worst-case complexity of this problem is $\Omega(\log n/ \log\log n)$ in the separable pointer machine model and Fredman and Saks [FS89] proved the lower bound even in the cell probe model. On the other hand, Tarjan [Tar75] showed that the amortized complexity of this problem is $\mathcal{O}(\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackermann function.

Amortized complexity is therefore suitable when optimizing throughput, at the cost of a possibly long running time of few operations. However, in many situations the unbounded running time of a single operation is disadvantageous:

- Structures with amortized complexity are unsuitable in real-time and interactive environments, for example, in a flight simulator it is vital to respond smoothly to the pilot's movements, and a power plant control system must respond immediately to the changing conditions of the power plant.

  Modern LISP systems use incremental garbage collection [Wad76, Bak78b], which can be considered a worst-case variant of garbage collection, instead of classical garbage collection, which is efficient in the amortized sense. The paper [Wad76] justifies incremental garbage collection on an example of a tennis-playing robot stopping in mid-swing to garbage collect.

- Amortized structures perform poorly in parallel environments. If a task is performed by several processors and all operations on one of the processors have a long running time, this processor finishes well after all others and slows down the entire task. Alternatively, if an amortized data structure is shared by several processors, the processors spend large amounts of idle time when waiting for the processor executing long running operation.

  For example, the paper [FT87] introduces Fibonacci heaps, which is an efficient amortized data structure that speeds up Dijkstra's shortest-path algorithm in sequential setting. However, in [DGST88] the attempts to use Fibonacci heaps in a parallel environment to achieve similar improvements fail because of the mentioned reasons. The authors therefore develop a worst-case variant of Fibonacci heaps, the relaxed heaps, which improves the performance of Dijkstra's shortest-path algorithm in parallel setting.

- The existing methods for making data structures persistent work efficiently only with worst-case structures [DSST89]. Moreover, we prove in Section 2.4 that fully persistent variants of amortized data structures can perform very poorly.

Therefore, persistent structures with worst-case complexities are more versatile and can be used in a wider range of applications. Often the worst-case complexity is the same as the amortized complexity and in that case, persistent data structures with worst-case complexity are preferable to structures with amortized complexity.

1.3 Structure of the Thesis

The goal of this thesis is to devise new data structures and improve the existing data structures that can be used in functional languages.

The thesis consists of two complementary parts. The first part deals with persistent data structures with good asymptotic complexities, concentrating on the fully persistent array data structure presented in [Str09]. The second part discusses purely functional data structures with supreme running times instead of low asymptotic complexity. We focus on the study of Adams' trees covered by the author's paper [Str12], and improve the Haskell de facto standard data structure library CONTAINERS, as described by the author's paper [Str10].

Persistent Data Structures

In the first part of the thesis we focus on persistent data structures. These structures usually have imperative implementations and use side effects internally, but retain all the previous versions and appear immutable from the outside. We are interested in asymptotic complexities of the structures and their optimal lower and upper bounds, extending the theoretical understanding of persistent data structures. However, we do not neglect the real performance of the structures and devote the second part of the thesis to designing efficient implementations.

The main contribution of the first part of the thesis is the study of persistent arrays presented in Chapter 5. We present an amortized and worst-case implementations of a fully persistent array, and show how to deal with garbage collection of these implementations. The sketch of an amortized implementation has already been published in the extended abstract [Die89], however, our construction uses simpler auxiliary structures and is more suitable as a basis for the worst-case implementation. The amortized implementation has optimal complexity and the worst-case implementation has nearly-optimal complexity, which can be improved to optimal if an open problem of worst-case dynamic integer set from [AT07] is solved. We also show how to deal with garbage collection of persistent arrays. This topic has not been addressed by existing persistent array publications, which makes the implementations unable to free the array versions that are no longer used. We describe novel algorithms for recognizing unused versions and reaching optimal memory utilization in case some of the array versions become unreachable.

We also explore the possibility of making amortized data structures persistent. We start Chapter 2 by describing general methods for making worst-case data structures persistent, both with amortized and worst-case complexities, summarising the results of [DSST89, Bro96, Ram92]. We then consider general methods of creating efficient persistent variants of amortized structures. These methods use additional operations of the amortized structures that are common and allow controlling the amortized complexity of the structures. Because these methods turn out to be quite inefficient, we concentrate primarily on proving the lower bounds.

Chapters 3 and 4 describe algorithms and data structures needed in the construction of the persistent array. In Chapter 3 we deal with the problem of navigating the version tree of persistent structures. This problem is related to two well-known problems, the list labelling and the list order problem. We describe solutions of the amortized and worst-case list order problem from [DS87] and simpler solutions of the amortized list labelling and list order problem from [BCD$^+$02]. We also present an algorithm for worst-case list labelling with polynomial labels, which we need in the construction of persistent arrays. Such an algorithm was designed by [BCD$^+$02] and should appear in a journal version of this paper. However, at the time of writing of this thesis, the journal version of the paper [BCD$^+$02] has not yet been submitted for publication. In order to give a complete description of persistent arrays, we present an algorithm for worst-case list labelling with polynomial labels, which we devised independently. Using this algorithm we describe a slightly simpler solution to the worst-case list order problem in the lines of the solution of [DS87], and utilize it in the construction of the persistent array.

Structures suitable for representing dynamic integer sets are described in Chapter 4. These structures are substantially faster than binary search trees and in conjunction with the list labelling provide efficient algorithms for searching in the version tree, sometimes with optimal complexity. This chapter mostly summarizes known results, we only propose an improved representation of worst-case van Emde Boas trees with smaller space complexity.

The detailed information about the contribution of this work can be found at the end of every chapter in the Chapter Notes section.

Purely Functional Data Structures

In the second part of the thesis we develop purely functional data structure implementations with best possible running times. The purely functional data structures are the prevalent data structures of functional languages and are used by a large number of users and companies.[4] We implement the data structures in Haskell, although we usually do not exploit advanced techniques like lazy evaluation and therefore the implementations can be ported to nearly all functional languages without loss of performance.[5]

Our main contribution is the study of Adams' trees in Chapter 7. Adams' trees are fully persistent balanced binary trees often used in functional languages. However, the existing correctness proof is flawed, which manifested itself in several implementations by violating the balance of the tree during deletions, which was discovered independently by us and by Taylor Campbell [Cam]. We present a valid correctness proof and investigate the space of Adams' trees parameters, measuring their impact on performance and height of the tree. In addition, we devise an improved representation leading to 20-30% memory reduction and improved time complexity of operations.

In Chapter 8 we concentrate on improving the actual performance of the standard Haskell library of purely functional data structures, the CONTAINERS package [PkgCont]. We start by comparing the data structures of this package to all available alternatives. We then describe the changes of the implementation and also of the compiler we employed in last three years. The library currently offers the most efficient implementations available and is used by every third Haskell package.[6] We also propose a new data structure based on hashing, which outperforms available string set implementations.

In Chapter 6 we devise a fully persistent array implementation and choose the best parameters of the implementation using benchmarking.

[4]The CUFP, commercial users of functional programming conference, has been taking place annually since 2004. Also, a list of companies using Haskell industrially can be found at `http://www.haskell.org/haskellwiki/Haskell_in_industry`.

[5]For example, the `IntMap` data structure from the Haskell CONTAINERS package has been ported to F# in the fsharpx project, retaining its efficiency.

[6]On 14th May 2013, the CONTAINERS package was used by 1782 out of 5132 packages available on HackageDB, which is a centralized repository of Haskell packages.

Part I

Persistent Data Structures

Chapter 2

Making Data Structures Persistent

Ordinary data structures are *ephemeral* – after modifying the structure only the new version of the structure remains and the original version is not accessible anymore. However, it is often useful to be able to access or even modify multiple versions of the structure.

Definition 2.1. A data structure is *persistent*, if it preserves the previous version of itself when modified. If only the newest version can be modified and all the versions of the structure can be accessed, the data structure is *partially persistent.* A *fully persistent* data structure allows accessing and modifying all its versions.

Many persistent structures have been devised: stacks [Mye83], lists [DST94], sequences [HP06], priority queues [BO96], search trees [Mye84, Ove81a, RTD83b, KM83, Ada92, Ada93, Str12, HY11], tries [OG98, Gou94, Bag01] and related structures [Cha85, Col86, DM85]. In addition, [Oka99] describes a wide range of purely functional data structures and advanced techniques.

Most of these structures have been constructed independently and without systematic approach at first. That changed with the paper [DSST89], which introduced a generic method of creating a fully persistent structures. This method works for any linked data structures with bounded in-degree.

Definition 2.2. A data structure is a *linked data structure*, if it is a collection of nodes, each containing a fixed number of *fields* – both *information fields* and *pointer fields*. A linked data structure has *bounded in-degree* if, for each node, the number of pointer fields referencing this node is bounded by a constant.

Such structures can be made partially persistent and even fully persistent. The operations on these structures must consist of a sequence of *access steps* and *update steps*. In an access step, a field of a node is retrieved. In an update step, a field of a node is changed or a new node is added. The resulting persistent structures have worst-case $\mathcal{O}(1)$ time complexity per access step and amortized $\mathcal{O}(1)$ time and space complexity. Therefore, the amortized complexity of persistent structures is the same as the worst-case complexity of the ephemeral structures. The method of creating persistent structures is described further in Section 2.2.

This method can be used to create persistent variants of many basic structures like lists, balanced trees, tries, priority queues and others. However, it has several limitations. One of the downsides of the method is that update operations have no worst-case bound. We describe several methods for improving the worst-case complexity in Section 2.3.

Also, the original method cannot be used on structures with amortized complexity. The problem is that amortization depends on the structure being ephemeral – if an operation takes a long time to execute, this long running time is compensated by many other operations executing quickly. But in the persistent setting, we can repeat the time consuming operation at will. Therefore, we consider general methods of creating efficient persistent variants of amortized data structures in Section 2.4. These methods use common additional operations of the amortized data structures to control the amortized complexity. The general methods turn out to be quite inefficient, therefore, we concentrate primarily on the lower bounds, showing matching upper bound only in specific (but quite common) settings.

Another important structure which cannot be made persistent in this way is a persistent array, because it is not even a linked structure. We deal with persistent arrays later in Chapter 5.

Some persistent structures allow not only to modify a version of the structure, but also to combine several different versions of the structure. A simple example is catenation of different versions of a persistent list [DST94]. Such structures are called *confluently persistent*. In [FK03] a generic method allowing to create confluently persistent structures is devised, but the resulting structures are not very efficient. The problem is that the version graph of confluently persistent structures is not a tree but a directed acyclic graph, which is very complicated to navigate. Nevertheless, several structures exist, most of them purely functional, that provide confluent operations with optimal complexity, e.g., list catenation [DST94], queue catenation [Oka99] or priority queue catenation [BO96].

2.1 Path Copying Method

The path copying method is a simplest method of creating persistent structures from linked data structures. It was described and named in [ST86], although many earlier papers independently used this method, for example [Mye83, Mye84, KM83, RTD83a].

We illustrate the method on a simple example of a singly linked list. In order to modify for example the third element, we create a node containing the new value of the third element, and we create a copy of all preceding nodes, updating the forward pointers accordingly. This situation is illustrated in Figure 2.1.

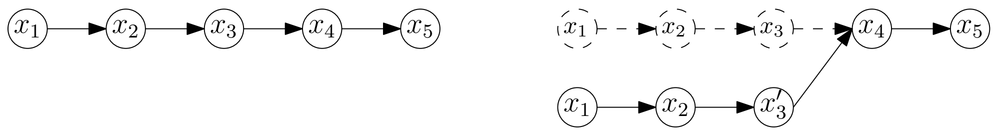


Figure 2.1: Result of path copying method after modifying list element x_3

If we are given any linked data structure, we can therefore create a persistent variant using a simple rule: when modifying a node, we create its updated copy and we also create a copy of all nodes that are (direct or distant) predecessors of this node.[1] A slightly more complex example showing modification of a value in a binary search tree is displayed in Figure 2.2.

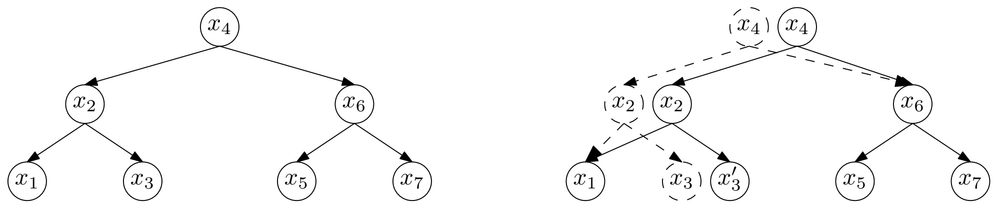


Figure 2.2: Result of path copying method after modifying tree element x_3

The complexity of the resulting structure can in theory be very bad. Nevertheless, when used on a structure where every node has at most one direct parent (e.g., linked lists, trees, tries, priority queues), the time complexity of the resulting persistent structure operations is the same as the worst-case complexity of the ephemeral structure operations. That follows from the fact that all predecessors of a node have to be visited in order to reach the node being modified. Unfortunately, the space complexity of the persistent structure is usually increased – it is the larger of the original space complexity and original *time* complexity because of the node copying.

[1]These predecessors usually form a path, hence the name of the method.

For illustration, note that updating the i-th element of a linked list takes time $\mathcal{O}(i)$ but also $\mathcal{O}(i)$ space. Updating a node in a balanced binary tree has $\mathcal{O}(\log n)$ time complexity and also the same space complexity.

The increased space usage is the only disadvantage of this method when the original ephemeral structure has at most one predecessor for every node. Apart from that, the resulting persistent structure has optimal time complexity. Moreover, the structure can be implemented *without overwriting any memory.* This is of huge importance – the resulting implementations are without side effects and node fields are written to only once when created. This makes the implementations simple, lock-free and suitable for purely functional languages, where side effects are either forbidden or discouraged.

Another important property of the method is that it does not maintain explicit identification of the structure versions. This is unlike any of the following methods described in the first part of this thesis – all other methods explicitly identify versions and modify fields in existing nodes, using the version identifiers as timestamps, which allows reconstructing all versions of the structure. This requires side effects and thus locking and is disadvantageous in parallel environment. In the path copying method, on the other hand, every version of the persistent structure is independent of others (except for sharing of read-only parts of the structure).

Therefore, many functional data structures in use are persistent structures obtained by the path copying method. For a nice introduction and also advanced topics like amortized complexity of the persistent structures, see the book [Oka99].

2.2 Making Linked Structures Persistent

In this section we describe the methods of [DSST89] for creating persistent structures from worst-case linked data structures (see Definition 2.2), sometimes of bounded in-degree. The whole section is a summary of [DSST89], we only use the accounting method instead of the potential method for analysing amortized complexity in the proofs, because we believe it is simpler to follow.

Suppose we have an ephemeral linked data structure defined in Definition 2.2, where any node contains at most d pointer fields. Access to this structure is provided by a fixed amount of *access pointers*, each referring to a so called *entry node*. We assume the structure allows two kind of operations, *access operations* and *update operations*. An access operation computes a set of accessed nodes, which is initially empty. The access operation consists of *access steps*, each adding a node to the set of accessed nodes. The new node must be either an entry node or referenced by a pointer in a previously accessed node. A good example of access operation is lookup in a binary search tree.

An update operation is any sequence of access steps and *update steps*. In the update step the structure is modified – either a new node is created and added to the set of accessed nodes, or a single field of an accessed node is modified. If a pointer field is modified, its new value must either reference a node in the access set or be `null`.

Every update operation creates a new version of the structure. We number the versions sequentially using integers, starting from 1, so update operation i creates version $i+1$ of the structure. Let m be the total number of update operations.

2.2.1 Partial Persistence

Recall that, according to Definition 2.1, partially persistent structures allow access to any version, but only the newest version can be modified.

We describe two methods for creating partially persistent structures. The first one is the *fat node method* which makes any linked data structure partially persistent with $\mathcal{O}(1)$ space per update step and $\mathcal{O}(\log m)$ time complexity of the access and update step in the worst case. The second one, the *node copying method*, applies to any linked data structure with bounded in-degree and creates partially persistent data structures with worst-case $\mathcal{O}(1)$ time complexity of an access step and amortized $\mathcal{O}(1)$ time and space complexity of an update step.

Fat Node Method

In the fat node method, nodes of the persistent structure correspond directly to the nodes of the ephemeral structure. For each field, all its modifications are stored, indexed by the version in which the modification happen. Therefore, each field does not contain only the newest value, but a whole binary search tree with all the modifications of this field value. An example of a linked list created by the fat node method is displayed in Figure 2.3.

The nodes can become arbitrarily large, hence the “fat” in the name of the method. Accessing a field value in a given version requires a lookup in the tree containing the field modification and can be performed in $\mathcal{O}(\log m)$ time. In an update step, a new value of a field is inserted to the tree of its modifications, with the corresponding version number. Therefore, $\mathcal{O}(\log m)$ time and $\mathcal{O}(1)$ space is needed.

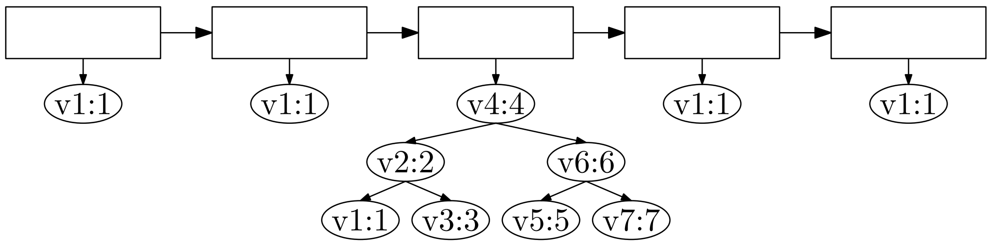


Figure 2.3: Partially persistent list created using the fat node method. The list initially contains five elements with value 1. Then the value of the third element is updated to 2, 3, 4, 5, 6 and 7.

Node Copying Method

The node copying method overcomes the problem of arbitrarily sized nodes present in the fat node method, but it can be used only if the ephemeral structure has bounded in-degree.

The basic idea of the node copying method is the following. Each node of the persistent structure contains a constant number of *extra fields*, which store modifications of the node fields. When we update a node, we record the modification in an extra field, together with the version when the modification happened. If all extra fields are used, a copy of the node is created, containing the newest values of the fields, and the update is performed. We also store pointers to the newest copy of the node in all its predecessors. If there is no space in a predecessor to store the new pointer, it is copied in the same way.

We now show that this idea leads to a partially persistent structure with amortized $\mathcal{O}(1)$ time and space complexity of an update step.

Each version of the ephemeral structure is identified using the version number and the value of all access pointers in this version.

Let the maximum number of predecessors of a node in the ephemeral structure be p. The node of the persistent structure contains the information fields and d pointer fields, like the node of the ephemeral structure. It also contains p *predecessor pointers*, one *copy pointer* and e *extra pointers*, which contain modifications of the pointer fields with corresponding versions.[2] The node also contains a version stamp.

The correspondence between the ephemeral structure and the persistent structure is the following. Each node of the ephemeral structure corresponds to a fam-

[2]Same as in [DSST89], we do not allow modifications of the information fields and create a copy of the node whenever a information field changes. If desired, it is be possible to allow storing information field modifications.

ily of nodes in the persistent structure, represented as a linked list of nodes from the oldest to the newest, connected using the copy pointers. The last member of this list is called a *live node* and represents the newest version of the ephemeral node. Any pointer in the persistent structure representing a newest value of a pointer field in the ephemeral structure is called a *live pointer*. Every live pointer points to a live node and we store its inverse in a predecessor pointer field.

Performing an access step is straightforward. To access a version i of a pointer field, we first search the extra pointers for a latest modification of the pointer field with a version at most i. If not found, we use the pointer field itself.

Now consider an update step modifying node x in version i. If the version of the node is i, we modify the appropriate field. If the update modifies a pointer field and there is an unused extra pointer, we store the modification in the extra pointer, together with version stamp i. Otherwise (i.e., if an information field is being modified or if there are no unused extra pointers), we create x', a copy of x with version stamp i, and update the copy pointer of x to point to x'. Then we fill the information and pointer fields of the node x' with their newest values and perform the required update. Because all the pointer fields contain live pointers, we update the predecessor pointers in all nodes referenced by x' to point to node x' instead of x. In any case, if the update modified a pointer field, we update the appropriate predecessor pointer in the referenced node.

It remains to update the live pointers pointing to x. We go through the predecessor pointers of x and for each node we update its appropriate pointer field to x' in version i. We perform each update recursively as described. During recursive updates, it can happen that a live pointer references a node which has already been copied during this update step, but the pointer itself has not yet been updated. We handle this situation using copy pointers – when we follow a live pointer, we check the copy pointer of the referenced node and follow it if it is nonempty. An example of a linked list created by the node copying method is displayed in Figure 2.4.

Finally we show that the update step finishes in $\mathcal{O}(1)$ amortized time. We use the accounting method and accumulate a credit of p times the number of nonempty extra fields in every live node. We give each update a credit of $p + 1$. We use one credit to perform the update operation without the recursive calls. If the modification is stored in an extra field, no node copying took place and we give the credit of p to the node to compensate for the used extra field. If the node copying took place, the original node has a credit of $p \cdot e$ which we can use.

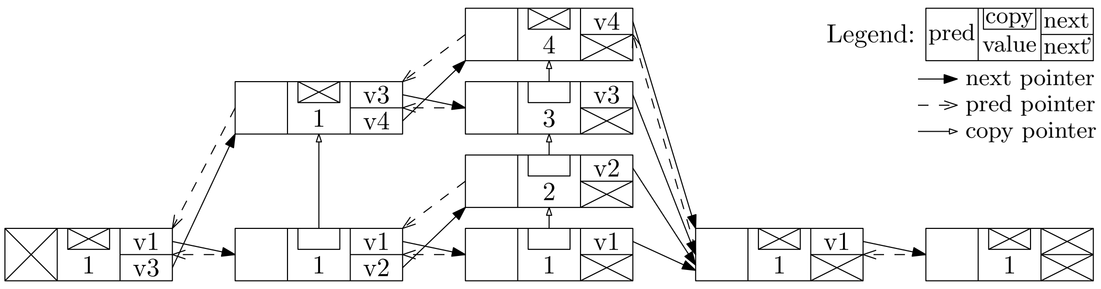


Figure 2.4: Partially persistent list created using the node copying method. The list initially contains five elements with value 1. Then the value of the third element is updated to 2, 3 and 4.

Therefore, we have a credit of $p \cdot (e + 1)$ and need to perform at most p recursive updates. Consequently, if $e \geq p$, we have enough credit to give to the recursive calls to update and the amortized time and space complexity of the update step is $\mathcal{O}(1)$.

Remark. The node copying method can be simplified in many cases. Consider for example a singly linked list or a binary search tree. Every node apart from the root has exactly one predecessor and cannot be accessed without the predecessor being accessed first. Therefore, there is no need to store the predecessor pointers and the copy pointers and even the time stamps in the nodes. The node of the persistent structure then contains in addition to the fields of the ephemeral structure only one extra pointer field with a version stamp.

2.2.2 Full Persistence

In this section we deal with the harder problem of making linked data structures fully persistent. We describe two methods analogous to the methods for creating partially persistent structures, with the same complexities. Namely, the fat node method makes any linked data structure fully persistent with $\mathcal{O}(1)$ space per update step and $\mathcal{O}(\log m)$ time complexity of the access and update step in the worst case. The *node splitting method*, a variant of the node copying method, applies to any linked data structure with bounded in-degree and creates fully persistent data structures with worst-case $\mathcal{O}(1)$ time complexity of an access step and amortized $\mathcal{O}(1)$ time and space complexity of an update step.

Navigating the Version Tree

In contrast to the versions of a partially persistent structure which have a natural linear ordering, the versions of a fully persistent structure form a tree and are only partially ordered. This lack of linear ordering makes it difficult to represent the versions such that for a given version its nearest predecessor can be found in a set of versions.

We deal with the problem of navigating the version tree in detail in Chapter 3. We now describe only the essentials we need in this chapter.

To get around the problem of partial ordering of the versions, we extend it to a linear ordering. Therefore, instead of a version tree, we a use a *version list*, which is a preorder traversal of the version tree, and order the versions according to this version list.

When we update a field in version v creating version w, at first we insert w in the version list just behind v. We then lookup the value of the field in version v and *store this value in a version following* w in the version list if it exists. Then we update the field in version w. Therefore, we store the required change in version w and an *undo* of this change in the version following w, so the value of the field is not affected in the versions following w. Consequently, if we want to know a value of a field in version v, we use the modification which happened in the version which is the nearest predecessor of v in the version list.

The last issue remaining to be solved is how to compare the versions according to their order in the version list. This list order problem has been extensively studied and we describe it in Section 3.3. Notably, we can use the Theorem 3.6 which allows comparing the versions in the version list in constant time, spending amortized $\mathcal{O}(1)$ time to insert the version in the version list. A worst-case variant of the algorithm is also available and described in Theorem 3.9.

Fat Node Method

The fat node method for obtaining a fully persistent structure is very similar to the partially persistent case. Each node in the persistent structure corresponds to a node of the ephemeral structure. For each field, the modifications performed on this field are stored in a balanced search tree, ordered according to the version list.

When accessing a field in version v, we find the value of the field in version v or its nearest predecessor in the tree of modifications performed to the field in question. This can be done in $\mathcal{O}(\log m)$ time.

To update a field in version v creating a version w, we insert the version w after version v in the version list. If there is a version following w in the version list, we lookup the value of the field in version v and store this value in the version following w. Finally, we add the required modification of the field in version w. The update step takes $\mathcal{O}(\log m)$ time and $\mathcal{O}(1)$ space in the worst case.

Node Splitting Method

The analogue of the node copying method in the case of fully persistent structures is the node splitting method. The method involves several technical details which we do not fully discuss here. An elaborate description of the method can be found in the original paper [DSST89].

The node of the persistent structure is the same as in the partially persistent case, i.e., the node contains the information fields, d pointer fields, p predecessor pointers, a copy pointers, e extra pointers and a node version. Each node of the ephemeral structure corresponds again to a family of nodes in the persistent structure, linked via the copy pointers in the order of increasing node versions.

The major difference from the partially persistent case is that all nodes can be modified. To accommodate this, we store a predecessor pointer for *every* forward pointer in the persistent structure, with the same version as the version of the corresponding forward pointer. We also allow predecessor pointers to be stored in extra pointer fields. Because access pointers need to be updated sometimes, we also store a predecessor pointer for every access pointer.

During an access step in version v we search the extra fields for a latest modification of the required field according to the version list. If such a modification is not found or if we are accessing an information field, we use the value in the field itself. Therefore, the access step takes $\mathcal{O}(1)$ time in the worst case.

To perform an update step modifying node x in version v creating version w, we start by inserting the version w into the version list after the version v. If the version of the node x is w, we update the required field directly. If the update modifies a pointer field and there is an unused extra pointer, we store the modification in the extra pointer. Otherwise, let u be the version following w in the version list. We create two new nodes, x_w with version w and x_u with version u, and update the copy pointers accordingly. We fill the forward and predecessor pointers and information fields of the two nodes according to their values in node x. We also move the extra fields of x with versions equal or larger to u to the node x_u and update their corresponding inverse pointers (forward pointers in the nodes referenced by predecessor pointers and predecessor pointers in the nodes referenced by forward pointers). Then we perform the required modification of the fields of x_w. If the version u does not exist, we proceed the same, just not creating node x_u. In any case, if the update modified a pointer field, we also update its inverse pointer.

The last step is to make sure a predecessor pointer exists for each forward pointer and vice versa. During the update, we maintain a list of newly created forward and predecessor pointer fields. For each such pointer field of node y in version u we verify that the original inverse pointer can be used to access the node y in version u (i.e., the inverse pointer is valid in version u) and if it can, we update the value of that pointer to y in version u. If not, we change the pointer in y to `null`.

The updates are performed recursively. If there are no extra fields left in a node during a recursive update, we split the node by finding a version v and moving the extra fields with version at least v to the new node, such that both nodes contain at least $e/2$ unused extra nodes.

To analyze the time complexity of the update step, we again use the accounting method. For every node with less than $e/2$ unused extra fields we accumulate a credit of $d+p$ times the number of nonempty extra fields minus $e/2$. This way a node with all extra fields used has a credit of $(d+p) \cdot e/2$ and the nodes created by a split of such a node have both zero credit. When updating a field in one version, we give the update a credit of $d + p + 1$. We use one credit to perform all the work, excluding the recursive updates. If an extra field is updated, the credit of the corresponding node is increased by $d+p$. If a node is full and splits, we gain $(d + p) \cdot e/2$ credit, therefore having a total credit of $(d + p) \cdot (e/2 + 1)$. After the split we need to update at most $d+p$ inverse pointers corresponding to forward and predecessor pointers of the newly created node. We conclude that if $e \geq 2d + 2p$, we have enough credit to pay for the recursive calls. Therefore, the amortized time and space complexity of the update step is $\mathcal{O}(1)$ in that case.

2.3 Making Linked Structures Persistent in the Worst Case

The described node copying and node splitting methods have amortized constant complexity. That can be disadvantageous in some applications, as discussed in Section 1.2.1. Therefore, there have been devised several improved method to bound the worst-case complexity of the update step, which we describe in this section.

2.3.1 Partial Persistence

Although the node copying method has amortized bounds, in special cases it can do better. Consider a structure with at most one predecessor, like a singly linked list or a binary tree. Even if the update step has amortized complexity, the complexity of the whole operation modifying a list or a tree is worst-case, because only the predecessor of a modified node can be copied and we have to access these predecessors anyway to locate the required node. However, in general case, the worst-case bound on the update step of the node copying method is $\mathcal{O}(m)$ if every node is copied during one update step.

In [Bro96] a method is described that makes partially persistent structures from linked data structures with bound in-degree, such that the time complexity of an access step and time and space complexity of an update step is $\mathcal{O}(1)$ in the worst case.

The method is a modification of the node copying method. The difference is that we do not copy the nodes recursively at the moment they have no free extra pointers. Instead, after every update step, we choose one live node and perform the node copying. If we choose the nodes appropriately, then it is possible to prove that there is a constant bound on the number of used extra pointers in any node.

We use the following algorithm to choose a node to be copied. We colour the live pointers in the structure with two colours, black and white. For every live node, there can be at most one white pointer referencing this node, and we say a live node is white iff there is a white pointer referencing it. The white pointers partition the nodes into components, each being a rooted tree of white nodes with a black root. Also, in each live node we store a queue containing all its predecessor pointers.

When an update is performed on a node x, we find the root r of the component where x belongs, possibly x itself. Then we break the whole component down by changing all white pointers in the component to black. Afterwards we perform node copying on r. Lastly, if the new copy of r has any predecessor pointers, we remove a predecessor from a queue, mark the corresponding live pointer white and return the predecessor to the end of the queue.

The Theorem 3 of [Bro96] proves that using this strategy, at most $2pd + 1$ extra pointers are used in any node and shows that this bound is tight.

It remains to show how to implement the described strategy in constant time, namely how to find the root of the component and change all white pointers in a component to black. We represent a component using a *component record* which contains a pointer to the root of the component or `null`. Every live node has a pointer to a component record and all nodes in one component share the same component record. If a node points to a component record containing `null`, the node is black and is the root of the component. Otherwise, the node is white and the root of the component is stored in the component record.

Using the component records makes it easy to find a root of a component of a given node. It also makes possible to break down the component – because all nodes in the component share the same component record, it is enough to set the pointer in the component record to `null`. To mark a black node white, we inspect the predecessor. If it is black, i.e., if the component record contains a `null` pointer, we create a new component record referencing the predecessor and store this component record in the predecessor. In any case, we then make the node in question point to the component record of its predecessor.

2.3.2 Full Persistence

In case of full persistence there is no known method that would allow performing both an access and update step in worst-case constant time.

However, there are several methods improving the worst-case complexity of fully persistent structures. In [Ram92] the following modification of the node splitting method is used to obtain two worst-case variants.

We modify the node splitting method to perform one node split after each update step, instead of splitting the nodes recursively. After we perform the update, we locate a node containing the most extra fields. If it contains at least $d+p+2$ extra fields, we split it and move at least $d+p+2$ and at most $2d+2p+2$ extra fields to the new node. That is always possible, because there are at most $d+p$ field modifications with the same version. Therefore, in every update step, we move constant number of extra fields and perform at most $d+p$ pointer updates in the split and add at most 2 field modifications in the update itself (if we change a pointer, we must update also the inverse pointer).

Using the Theorem 16 from [Ram92] we can prove, that if in one step we add at most $d+p+2$ extra fields to the whole structure and remove at least $d+p+2$ extra fields from the node with the most extra fields, the number of extra fields in any node is bounded by $\mathcal{O}((d+p+2)\log m)$, i.e., by $\mathcal{O}(\log m)$.[3]

Therefore, if we store the modifications in a binary search tree, we can perform an access and update step in $\mathcal{O}(\log\log m)$ time, using $\mathcal{O}(1)$ space in the update step.

The access complexity can be improved on a RAM, at the cost of increasing the complexity of an update step. We describe the algorithm proposed in [Ram92]

[3]This result is asymptotically optimal, even if we split the node in halves, see Theorem 3.7.

and fill in some details omitted in the paper. This description is quite technical and is not used in the rest of the thesis, therefore, it is safe to skip it and continue with the next section.

To improve the access step complexity, we use a faster structure than a binary search tree. There are several structures allowing to manipulate a set of integers faster than in logarithmic time. To be able to use such structures, we have to assign integer labels to the versions, such that the order of the labels and the versions in the version list is the same. We describe this list labelling problem in Section 3.2.

The original description mentions a manuscript of Dietz describing a worst-case solution of list labelling. We were not able to find it and as far as we know, such algorithm has never been published (see beginning of Section 3.2.2 for details). Fortuntely, we describe the algorithm in Theorem 3.5, because we use it in the construction of the persistent array in Chapter 5. The algorithm maintains for every version an integer label of length $\mathcal{O}(\log m)$ bits. During an insertion of a new version, $\mathcal{O}(\log m)$ labels are changed. This is asymptotically optimal, as discussed in Section 3.2.2.

We utilize the version labels by storing field modifications in a structure indexed by the version labels. When an new version is created, $\mathcal{O}(\log m)$ labels are changed and we have to update the structures containing modifications with this label. To limit a number of modifications associated with a single version, we use the extended version list described in Definition 3.2, which guarantees that only a constant number of modifications is associated with a single version.

The original description uses the small priority queue of [AFK84], which allows searching, inserting and deleting elements in constant time, assuming the number of elements is asymptotically limited by the number of bits in a word. When we use a structure with such parameters, the complexity of the access step is $\mathcal{O}(1)$ and an update step takes $\mathcal{O}(\log m)$ time and $\mathcal{O}(1)$ space in the worst case.

Nevertheless, the small priority queue of [AFK84] is defined in the cell probe model only and it is not obvious how to modify it to work on a RAM. Indeed, it has taken a decade before such modification, called a *Q-Heap*, was devised in [FW94]. Denoting the word size of a RAM as w, a Q-Heap is a priority queue working with w-bit numbers, able to perform operations findmin, insert and delete in worst-case constant time. Nevertheless, Q-Heap can contain at most $w^{1/4}$ elements and a precomputation in time linear with the number of maximum number of elements stored in the Q-Heap must be performed.

Despite the size limitation and the precomputation requirement, Q-Heaps allow us to manipulate with $\mathcal{O}(\log m)$ version labels in constant time. To overcome the size restriction of the Q-Heaps, we store the $\mathcal{O}(\log m)$ labels in a trie that has Q-Heaps in every node. Because the Q-Heap can store $\log^{1/4} m$ numbers, we store $\log^{1/4} m$ bits of the labels in every Q-Heap, therefore, the whole trie has constant depth and we can manipulate with the labels stored in the trie in constant time.

During the course of the algorithm, the number of modifications stored in a node increases, but the Q-Heaps have a fixed maximal size. To overcome this problem, we use a variant of global rebuilding [Ove83a]. We start with Q-Heaps capable of holding constant number of elements. When the maximum size of Q-Heaps is equal to $\log^{1/4} m$, we start rebuilding the Q-Heaps. In the first $\log^{1/4} 2m$ steps we perform the precomputation allowing us to create Q-Heaps with maximum size of $\log^{1/4} 2m$. In the following $m/2$ steps we gradually convert all Q-Heaps to the new representation. The whole rebuilding finishes in $\log^{1/4} 2m + m/2$ steps, i.e., before the total number of modifications of the whole structure is $2m$. Therefore, the rebuilding finishes before another rebuild is needed.

2.4 Making Amortized Structures Persistent

The complexity bounds of the discussed methods of making persistent structures do not hold for structures with amortized bounds. That is unfortunate, because for some problems, the time complexity of a worst-case algorithm is provably higher than amortized time complexity, see Section 1.2.1. Also the algorithms with amortized time complexity are often much simpler than the worst-case ones.

There are several equivalent methods for analyzing amortized complexity, the most suitable in our case is the potential method. In this method, a non-negative potential $\mathcal{P}(A)$ is assigned to every state of a structure A and the amortized cost of an operation is defined as the actual running time plus the increase in the potential, i.e., $\textit{amortized cost} = \textit{actual running time} + \mathcal{P}(A_{\textit{after}}) - \mathcal{P}(A_{\textit{before}})$.

As an example, consider an array which supports adding a new element at the back. To accommodate the increasing size, we store the array elements in a memory block of possibly larger size, storing the number of elements currently present in the array. When adding a new element, we check whether the memory block used is larger than the size of the array. If so, we only increase the size of the array and we are done. Otherwise, we allocate a memory block of twice the size, copy the old array to the new memory block and finally add the new element. Although the worst-case complexity of adding a new element is linear, we show that the amortized cost is constant. Consider a potential of *the difference between the numbers of used and unused elements in the memory block.* The amortized complexity of adding a new element when there is unused space in the memory block is $\mathcal{O}(1)$, because the actual running time is $\mathcal{O}(1)$ and the potential increases by 2. When the memory block is full and we copy it to a new block of double size, the actual running time is $\Theta(n)$, but the potential decreases by n, therefore, the amortized cost is $\mathcal{O}(1)$.

The reason why the discussed methods of making persistent structures fail for data structures with amortized bounds is that we can repeat the operations with high running time. Consider the situation in the mentioned example when the array is represented using a memory block of the same size. Adding a new element has $\Theta(n)$ running time. In the original structure, this is compensated by the following n additions which all have $\mathcal{O}(1)$ running time. Nevertheless, in a persistent structure, we can repeatedly add a new element to the array represented in a memory block of the same size, causing each such operation to take $\Theta(n)$ time.

Therefore, we explore the possibilities of making data structures with amortized complexities of operations persistent. We are interested in methods that work without changing the representation of the structure. Naturally, changing the representation of individual structures might lead to better results, but a general method would be useful even if it was less effective. However, we show that such methods in our settings are considerably inefficient and therefore we focus on the lower bounds and only sketch the upper bounds if available.

Consider a data structure A with an operation *update.* Let n denote the size of the structure and suppose that *update* has amortized complexity. The *update* operation can change the size of the structure by a constant amount and there can be several kinds of the *update* operation, e.g., in case of a set, an *update* can be both addition and removal of an element. Our goal is to obtain fully persistent variant P of this structure, with either amortized or worst-case complexity bounds. For the resulting persistent data structure to be effective, we need some way of controlling the potential of the structure. We consider two such methods:

- *rebuild* operation, which rebuilds given A structure and creates an equivalent structure with minimal potential (usually zero),
- *undo* operation, which is an inverse to *update.*

We consider these two methods to be most frequently provided by amortized data structures, notably the *rebuild* operation.

2.4.1 Using the Rebuild Operation

Let A be a data structure with an operation *update* with amortized complexity $u(n)$ and a *rebuild* operation, which rebuilds a given A structure to an equivalent structure with minimal potential.

We assume that the complexity of the *rebuild* operation depends only on the size of the structure. In theory, the *rebuild* operation could also depend on the potential of the structure. Nevertheless, the dependence on the size only is very common (in the shortly mentioned examples, *rebuild* depends only on the size and we do not know of alternative implementations which would depend on the potential). Also, if the *rebuild* depended only on the potential, a worst-case variant of the corresponding structure could be created by calling *rebuild* after every *update*, because the potential is only $\mathcal{O}(u(n))$ after one *update*. Therefore, we consider only the dependence of the *rebuild* complexity on the size of the structure.

We now present several amortized structures with the described *rebuild* operation. The first one is an implementation of a queue using two stacks, a folklore construction (described for example in [Oka99]) frequently used when only singly-linked lists are available in a computer language. The queue is represented as two halves, the front half and the rear half, each stored in a stack with the outermost queue element on top. Adding a queue element is implemented by adding the element to the rear half, removing a queue element is implemented by removing the element from the front half. If the front half is empty, the rear half is reversed and becomes the front half. It is simple to show that if we define the potential of the queue to be the size of the rear half, both adding and removing a queue element has constant amortized complexity.

In our settings, the *update* operation is either adding or removing a queue element, with constant amortized complexity. The *rebuild* operation is a generalization of the operation used in the queue implementation – it reverses the rear half and prepends it to the front half in $\Theta(n)$ time, zeroing the potential.

Another example are the splay trees [ST85]. The splay trees are binary search trees providing operations insert, find and delete, all with $\Theta(\log n)$ amortized complexity. The potential of a splay tree is a sum of potentials of all its nodes, and the potential of a node is a logarithm of the size of the subtree rooted in this node. In our settings, the *update* operation is either adding, finding or deleting an element with $\Theta(\log n)$ amortized complexity, and the *rebuild* operation recreates the tree to be perfectly balanced, in linear time.

The well-known disjoint set union-find data structure [Tar83] fits in this model, if only the path compression is used, not rank by size (otherwise the operations have logarithmic worst case). The *update* operation is either find or union with amortized $\Theta(\log(n))$ complexity, and the *rebuild* operation performs path compression on every node, which can be done in $\Theta(n)$ time.

The final example is a fully persistent list created by the node splitting method of Section 2.2.2. The *update* operation with constant amortized complexity is a modification of a list element and the *rebuild* is a generalization of the node splitting process – it processes the list elements from back to front, storing modifications for every element in the smallest number of nodes with zero potential, i.e., every node is at most half full.

We now show the lower bound of any method which creates a persistent variant of a given amortized structures with *rebuild*. Naturally, a large lower bound holds only for data structures that use amortization heavily. Therefore, we introduce the following definition.

Definition 2.3. Let A be a data structure with an operation *update* with amortized complexity $u(n)$ and a *rebuild* operation with complexity $r(n)$, which rebuilds given A structure to an equivalent structure with minimal potential. For a real $p > 0$, we say that *structure utilizes potential of order* p, if for every size n there exists an instance of the structure of size n, and for every instance of the structure of size n, there exists a sequence S of $\left\lceil \sqrt[p+1]{r(n)} \right\rceil$ *update* operations fulfilling the following properties. Let the sequence S be applied to the data structure, possibly calling *rebuild* operations at any point in the sequence. Then

- *rebuild* called at any point takes $\Omega(r(n))$ time, and
- for every m, if m first *updates* of S are performed without any *rebuild*, then there exists an *update* that decreases the potential by $\Omega(m^p)$. We call such an update a *discharging update*.

Informally, this definition guarantees that there exists a sequence of *updates* that builds large potential, the potential can be discharged at any time and the structure cannot be rebuilt cheaply. This is the case for all mentioned examples except for the first one, the queue using stacks structure. To illustrate, for splay trees, the sequence S consists of *updates* that insert an element greater than all other tree elements. To discharge the potential, the deepest element in the tree is accessed. Therefore, a splay tree utilizes potential of order 1. It is easy to show that also both the mentioned disjoint set union-find data structure and the fully persistent list also utilize potential of order 1.

When a *rebuild* operation is used, a structure equivalent to the original one is created, although it is usually different in order to achieve minimum potential. We extend the notion of equivalence induced by the *rebuild* operation as follows.

Definition 2.4. We say that two A structure instances are *equivalent*, when both are created from the same structure using the same sequence of *update* operations, arbitrarily interleaved by *rebuild* operations for either instance.

We are now ready to present the lower bound of persistent variants of structures with the *rebuild* operation.

Theorem 2.5. *Let A be a data structure with an operation* update *with amortized complexity $u(n)$ and a* rebuild *operation with complexity $r(n)$. Let this structure utilize potential of order p, according to Definition 2.3.*

If a persistent variant P of structure A is created by representing each version of P using any equivalent A structure instance (as defined in Definition 2.4), then the amortized complexity of the update *operation on P is $\Omega\left(r(n)^{p/(p+1)}\right)$. In the common case when $p = 1$ and $r(n)$ is $\Omega(n)$, the lower bound is $\Omega(\sqrt{n})$.*

Proof. Let n be arbitrary, let $r = \left\lceil \sqrt[p+1]{r(n)} \right\rceil$, and let S be the sequence of r operations from Definition 2.3. We start with any A structure of size n and repeatedly perform the following procedure.

We sequentially carry out the *updates* in S and after each one we perform the discharging *update* from Definition 2.3. The resulting version tree is displayed in Figure 2.5.

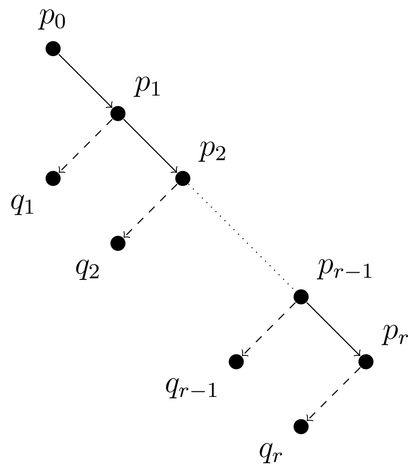


Figure 2.5: The version tree after performing the *updates* from S and discharging *updates*. The p_0 is any A structure of size n, p_i is created from p_{i-1} by the i-th *update* from S and q_i is created from p_i using the discharging *update*.

In total we carry out $\mathcal{O}(r)$ *updates.* To analyze the time complexity, we distinguish two cases depending on whether *rebuild* was executed. If a *rebuild* is performed, it takes $\Omega(r^{p+1})$ time. Otherwise, all the discharging *updates* take $\sum_{i=1}^{r} \Omega(i^p) = \Omega(r^{p+1})$ time.

We can repeat this procedure as many times as we want, with each $\mathcal{O}(r)$ operations taking at least $\Omega(r^{p+1})$. The lower bound is therefore $\Omega(r^p) = \Omega\left(r(n)^{p/(p+1)}\right)$.

□

To illustrate that the conditions of Definition 2.3 are required, consider the queue using two stacks data structure. This structure does not fulfil Definition 2.3, because of the following observation: *When a* rebuild *of size n takes place, the following n* updates *have worst-case constant complexity.* Therefore, the sequence S required by the Definition 2.3 for all instances of the structure does not exist for structures created by the *rebuild* operation.

Motivated by the mentioned observation, we can even create a persistent variant of this structure with constant amortized complexity. Such persistent queue implementation is well known and described for example in [Oka99], we only sketch the algorithm in our settings. We implement the queue using the path copying method of Section 2.1, with the following addition. When the front and rear half of the queue have equal size n, we mark this version. If we ever create a version of the queue by performing n *updates* to a marked version, we rebuild the marked version and replay all *updates* that happened to this version and its descendants, updating all corresponding queue versions. It is simple to show that the cost of the *rebuild* is amortized by the n *updates* that happened before this *rebuild* and that all *updates* have worst-case constant complexity because of the earlier observation.

Nevertheless, it is not easy to generalize this construction for other data structures. For a study of such constructions, see Chapter Amortization and Persistence via Lazy Evaluation of [Oka99].

Apart from the lower bound in the general case, we show an upper bound for linked structures with bound in-degree. The upper bound does not match the lower bound, but it is tight in the common case when $r(n)$ is $\Theta(n)$ (which is true for all mentioned applicable examples) up to a multiplicative factor of $\sqrt{u(n)}$.

Theorem 2.6. *Let A be a linked data structure with bounded in-degree with an operation* update *with amortized complexity $u(n)$ and a* rebuild *operation with complexity $r(n)$, with $r(n)$ being $\Omega(u(n))$.*

If we are able to maintain the value of the potential of the structure during *updates and* rebuilds *and compute values* $r(n)$ *and* $u(n)$*, we can create a persistent variant* P *of this structure, whose* update *operation has* $\mathcal{O}\left(\sqrt{r(n)u(n)}\right)$ *amortized time and space complexity, where* n *is the largest size of the structure. In the common case when* $r(n)$ *is* $\mathcal{O}(n)$*, the* update *complexity is* $\mathcal{O}\left(\sqrt{nu(n)}\right)$*.*

Proof. We obtain the P structure from A using the node splitting method of Section 2.2.2. The versions of P are A structures with defined potential, therefore, we define the potential of P to be the sum of potential of all its versions. Let $b(n) = c_b\sqrt{r(n)u(n)}$ for a constant c_b, whose value we fix later. We maintain two invariants:

I) Every version of P has potential at most $2b(n)$.

II) If q is a child of p in the version tree of P, the potential of q is at most the potential p plus $2c_u u(n)$, where c_u is the constant hidden in the asymptotic complexity of *update.*

When performing an *update* operation on the version p of P and obtaining version q, we check that potential of q is less than $2b(n)$. If so, both invariants are fulfilled and potential of P increases by at most $2b(n)$.

If the potential of q is not less than $2b(n)$, we proceed as follows. We denote the $b(n)/c_u u(n)$-th predecessor of q as r. The invariant II) guarantees that it exists and has potential of at least $b(n)$. We rebuild the version r and for every versions s, t violating the invariant II) we replace t by a structure created by performing an *update* operation to s. We perform these *updates* in a DFS order from the node r.

The potential of any node created by an *update* operation has decreased by at least $c_u u(n)$, thus paying the cost of the *update.* Moreover, the potential of all $b(n)/c_u u(n)$ nodes on the path from r to q decreased by at least $b(n)$, therefore, the resulting potential decrease of P is at least $b^2(n)/c_u u(n) = (c_b^2/c_u)r(n)$, thus amortizing the cost of the *rebuild* operation for large enough c_b.

Therefore, the amortized cost of an *update* operation is $\mathcal{O}\left(\sqrt{r(n)u(n)}\right)$. Unfortunately, this is both time and space complexity – the *rebuild* operation cannot overwrite the original structure, because other versions of P might share parts of the original structure and it could be the case, that these versions will not be modified, because they might not violate invariant II). ☐

2.4.2 Using the Undo Operation

Since the persistent variants of amortized data structures with *rebuild* operation are not very effective, we considered alternative operations that can control the potential and are frequently provided by amortized data structures.

In quite a few cases the amortized data structures provide an operation *undo* that reverts the effect of the *update* operation. This operation has usually the same amortized complexity as the *update* operation. An example of an amortized structure with *undo* operation are the splay trees [ST85] – inverse of an insert is a delete, inverse of a delete is an insert and inverse of a find operation is identity.

The *undo* operation can be used to provide more effective persistent variants of amortized structures – if an *update* decreases potential considerably, the *undo* operation can be used to decrease potential of many versions of the structure. Consider for example the queue using two stacks data structure from Section 2.4.1. This structure can be adapted easily to support the *undo* operation by adding a counter expressing how many elements from the front half should be considered removed. To undo deletion of a queue element, we add it to the front half. To undo insertion of an element which is on the top of the rear half, we remove it. To undo insertion of an element which has been already moved to the bottom of the front half, we increase the counter representing the number of elements considered deleted from the bottom of the front half.

We now sketch how a persistent variant of this structure can be created, with amortized $\mathcal{O}(\sqrt{n})$ complexity of *update*, where n is the largest size of the queue. We assign a potential of $\sqrt{n}$ to a queue version if the rear half is non-empty, otherwise the potential is zero. To implement *update* on the persistent structure, we use the original *update* operation if it has worst-case constant complexity. The only situation in which the original *update* has non-constant complexity is when the front half of the queue is empty. In this case, we distinguish two possibilities. If the current queue version has at least $\sqrt{n}$ predecessor versions with non-zero potential, we perform the required *update* directly in $\mathcal{O}(n)$ time and create a queue version with zero potential. We then use *undo* to replace all mentioned predecessor versions with non-zero potential by equivalent ones of zero potential, decreasing the potential of the whole structure by at least n. If only less than $\sqrt{n}$ predecessor versions with non-zero potential exist, we locate the nearest predecessor version with zero potential and replay all *updates* on the predecessor versions of the current queue version with non-zero potential, decreasing the rear half length of the current queue version to at most $\sqrt{n}$ in $\mathcal{O}(\sqrt{n})$ time, allowing us to perform the required *update* with $\mathcal{O}(\sqrt{n})$ time complexity.

However, analysing *undo* operation in general case is more challenging than analysing *rebuild* – first, *undo* gives no control over the value of the potential, and second, the potential of later versions can influence the potential of earlier versions because of the *undo* operation. We were not able to devise any general method of creating persistent variant of a given structure A with an *undo* operation, which would have better complexity than $\mathcal{O}(b)$ per update operation, where b is the maximum value of the potential of the structure.[4]

Analogously to structures with *rebuild* operation, we define an equivalence of A structure instances.

Definition 2.7. We say that two A structure instances are *equivalent*, when both are created from the same structure using a sequence of *update* and *undo* operations, such that every *undo* operation cancels the nearest uncancelled preceding *update* operation, and the subsequences of uncancelled *updates* are the same for both instances.

We now show that an efficient persistent variant cannot be devised for every amortized structure with *undo* operation. Similarly to structures with *rebuild* operation, we define the properties of structures with *undo* for which the lower bound holds.

Definition 2.8. Let A be a data structure with an operation *update* with amortized complexity and an *undo* operation with amortized complexity, such that *undo* is an inverse of *update*. For an integral b, we say that *structure utilizes potential of* b, if the following holds. There exist an initial instance of the structure A and a sequence S of $\mathcal{O}(b)$ *updates*, such that when the sequence S is applied to the initial instance, it creates a sequence of b versions of the data structure with potential at least b. For every instance of the data structure equivalent to any version created by S, *undo* does not decrease potential, and for every $0 < i \leq \sqrt{b}$ there exists a variant of *update* that decreases potential to $\min(\textit{current potential}, b - i\sqrt{b})$ in time $\Omega(1 + \textit{potential difference})$.

Theorem 2.9. *Let A be a data structure with an operation* update *with amortized complexity and an* undo *operation with amortized complexity, such that* undo *is an inverse of* update. *Let this structure utilize potential of b according to Definition 2.8.*

If a persistent variant P of structure A is created by representing each version of P using any equivalent A structure instance (as defined in Definition 2.7), then the amortized complexity of the update *complexity on P is $\Omega(\sqrt{b})$.*

Proof. Using the initial instance of A and the sequence S from Definition 2.8, we create the data structure versions $p_1, p_2, \ldots, p_b$ of potential at least b, and perform

[4]Creating a persistent structure with $\mathcal{O}(b)$ complexity is trivial, because $\mathcal{O}(b)$ is the worst-case complexity of an *update* operation.

the following $\sqrt{b}$ phases. In the phase x we perform an *update* operation to the versions $p_{b-i\sqrt{b}-x}$ for i from 0 to $\sqrt{b}-1$, creating versions $p'_{b-i\sqrt{b}-x}$. Using the assumptions of Definition 2.8, every *update* during phase x decreases potential to $b - x\sqrt{b}$ and the *undo* operation does not decrease the potential.

One way to perform these operations is as follows. Consider a block B_k which consists of versions $p_{k\sqrt{b}}, p_{k\sqrt{b}+1}, \dots, p_{k\sqrt{b}+\sqrt{b}-1}$. The *update* operations in this block operate on nodes from the last one to the first one. After the *update* operation on p_j, we perform an *undo* operation on the result p'_j and then another *undo* operation to get a structure equivalent to p_{j-1} and we replace p_{j-1} by it. This way, every *update* operation decreases the potential by $\sqrt{b}$, and the time complexity of all the phases is $\Omega(b\sqrt{b})$.

On the other hand, it is obvious that if the version p_j in block B_k is constructed by *undo* operations from a version in another block $B_{i>k}$, its potential is not lower than when p_j is created by an *undo* operation from p_{j+1}. This fact relies on the order in which we process the nodes in the block and also on the fact that the blocks consist of $\sqrt{b}$ nodes.

The same reasoning apply to the case when the version p_j in the block B_k is constructed by *update* operations from a version in another block $B_{i<k}$ – its potential cannot be decreased this way because the *update* operations can increase potential by one and every block consists of $\sqrt{b}$ nodes.

Therefore, it does not help to derive a version p_j from a different block than the one p_j belongs to, and the initial scheme of performing the operations is optimal, needing $\Omega(b\sqrt{b})$ time for $|S| + b = \mathcal{O}(b)$ *update* operations. The whole procedure can be repeated indefinitely. □

2.5 Chapter Notes

The beginning of this chapter (Sections 2.1, 2.2 and 2.3) is a summary of known general methods for creating persistent variants of worst-case linked data structures, compiled from [DSST89, Bro96, Ram92]. Our contribution is Section 2.4, where we explore the efficiency of creating persistent structures from ephemeral structures with amortized bounds. These are original results.

Chapter 3

Navigating the Version Tree

The versions of fully persistent structures form a tree. Therefore, the versions are only partially ordered. The lack of linear order makes it difficult to work with the versions, e.g., to find a predecessor of a version within a given set of versions.

As suggested by [DSST89] or [Die89], we can extend the partial ordering of the versions to a total ordering of the versions, i.e., linearize the version tree and obtain a *version list*. A suitable method of linearizing the version tree is described in Section 3.1.

Having a linear ordering of versions, we need to be able to compare them as fast as possible. This problem, called *the list order problem*, has a long history and has been addressed by several researchers, e.g., [Die82, Tsa84, DS87, BCD$^+$02]. In the list order problem, a linked list is maintained. A new node can be inserted to the list at any place and order of two nodes in the list can be compared. An algorithm similar to [DS87] with worst-case $\mathcal{O}(1)$ complexity for both the operations is described in Section 3.3.

The list order problem is tightly connected to the *list labelling*. In the list labelling, a linked list is also maintained. Nevertheless, instead of an operation comparing two nodes, the goal is to assign integer labels to the nodes, such that the order of the nodes in the list and the order of the node labels is the same. As with the list order problem, asymptotically optimal solutions are known. We elaborate on the list labelling in Section 3.2.

Using the list order problem, we can store an ordered set of nodes in a search tree. When we use the list labelling, we can take advantage of the integer labels and store an ordered set of nodes in an asymptotically more efficient structure for maintaining an integer set. Nevertheless, this does not always pay off – in the list labelling, the lower bound on the number of relabellings after an insertion to the list is $\Omega(\log n)$. In the Chapter 5 we describe a particular situation where this can be circumvented and it is profitable to use the node labels as keys in a fast data structure.

3.1 Linearizing the Version Tree

The most simple way to linearize the version tree is to use its preorder traversal, as in [DSST89].

Definition 3.1. The *version list* is a preorder traversal of the version tree.

We can maintain the version list in the following fashion. When a version w is created from version v, we insert w in the version list just after v. It is simple to prove by induction that the resulting list is the preorder traversal of the tree.

Nevertheless, in some cases the version list is not suitable. The problem is that when inserting a new version to the version list, an update to the versions *following* the version being inserted is performed. Therefore, the number of updates to a version cannot be bounded. If that is an issue, we use an extended version list. Note that although the extended version list is used in several papers, e.g., [Die82, Die89, Ram92], there is no agreement on the term itself.

Definition 3.2. The *extended version list* is a combination of preorder and postorder traversal of the version tree. Specifically, for each version v in the version tree there are two nodes (versions) in the extended version list. We denote these nodes as v_+ and v_-. The v_+ is identified with v and v_- is used for technical purposes. The whole extended version list is a result of a DFS traversal where v_+ is emitted when entering version v and v_- is emitted when leaving version v.

Maintaining the extended version list is similar to maintaining the plain version list. When a version w is being created from version v, we insert w_+ just after v_+ and w_- just after w_+. Once again, we can prove the correctness of this approach by induction.

With this definition, the mentioned problem of unbounded updates of one version is avoided – when inserting version v, we create two versions v_+ and v_- in the version list and perform updates only on these two versions.

3.2 List Labelling

In the list labelling we maintain a linked list. Nodes can be inserted anywhere in the list. The goal is to assign integer labels to all nodes, such that the order of the labels is the same as the order of the nodes in the list. The label of node n is denoted as ℓ_n.

The performance of list labelling depends on the range of the labels we can use. In this section we summarise known results and in detail describe the cases that we will use later. We also present an algorithm solving the worst-case list labelling using polynomial labels, for which no solution has been published.

3.2.1 Exponential Labels

When we can use exponentially many labels and we know the maximum number of the nodes in the list, there exists a simple algorithm which does not require any relabellings. Such an algorithm is useful if the list we are interested in has length $\mathcal{O}(\log n)$ – then the labels have $\mathcal{O}(\log n)$ bits and fit in a constant number of machine words.

Theorem 3.3. *When we can use exponentially many labels and we know the maximum size of the list in advance, we can solve the list labelling without any relabellings.*

Proof. Let n be the maximum size of the list. We create two auxiliary nodes and assign them labels 0 and 2^n. These nodes always stay at the beginning and at the end of the list.

When a new node n is being added between the nodes m and o, we set $\ell_n = (\ell_m + \ell_o)/2$, i.e., the new label is in the middle of the range where it belongs.

It is easy to prove by induction that after adding i nodes (disregarding the two auxiliary ones), the difference of labels of any two adjacent nodes is divisible by 2^{n-i}. Therefore, we can insert n nodes without relabelling. $\square$

3.2.2 Polynomial Labels

Frequently the labels can be chosen from a range with polynomial size with respect to the size of the list. In other words, the labels have $\mathcal{O}(\log n)$ bits, where n is the size of the list.

An algorithm using $\mathcal{O}(n \log n)$ relabellings for n list insertions, i.e., with amortized $\mathcal{O}(\log n)$ relabellings per operation, is described in [Die82] and in [Tsa84]. A much simplified variant is devised in [BCD$^+$02]. We describe the latter algorithm in Theorem 3.4.

A worst-case algorithm needing at most $\mathcal{O}(\log n)$ relabellings per every insert is known to exist – it is implied by the following connection of list labelling to weight-balanced trees, and is also mentioned in [Ram92] and [BCD$^+$02]. Nevertheless, as far as we know, it has never been published. In [Ram92] an unpublished manuscript of Dietz is referenced when talking about the algorithm. In [BCD$^+$02] authors claim to describe the algorithm in the full paper, but according to personal communication,[1] it has not been finished yet. Therefore, we present our worst-case algorithm in Theorem 3.5, and utilize it further in the persistent array construction.

All mentioned algorithms are asymptotically optimal – a lower bound on the number of relabellings needed to perform n list insertions is $\Omega(n \log n)$. A proof appeared in [DSZ05], but is seriously flawed, as acknowledged by one of the authors.[2] Nevertheless, the proof has been fixed in a recent paper [BBC$^+$12].

Connection to Weight-Balanced Trees

As described in [BCD$^+$02], there is a close connection between list labelling and weight-balanced trees. A binary representation of an integer label can be considered a path in a binary tree, 0 meaning *left child* and 1 meaning *right child.* Obviously, a strategy for maintaining tags with $\mathcal{O}(\log n)$ bits with amortized relabel cost $f(n)$ yields a strategy for maintaining a balanced binary tree of height $\mathcal{O}(\log n)$ with amortized insertion time of $\mathcal{O}(f(n))$.

The straightforward converse is false, because a single rotation of the root of the tree can be performed in $\mathcal{O}(1)$ time, but changes paths to all leaves, thus causing $\mathcal{O}(n)$ relabellings. But if we define a *weight* of a node to be the size of its subtree and a *weight cost* of an operation to be the sum of weights of modified nodes, we get following result: Any balanced tree structure of maximum degree d and height h with amortized or worst-case weight cost $f(n)$ for insertions yields a strategy for list labelling with amortized or worst-case cost $\mathcal{O}(f(n))$, respectively. The resulting labels use $h\lceil \log_2 d \rceil$ bits.

Many rotation-based trees like red-black trees or AVL trees have $\mathcal{O}(n)$ weight cost insertions and deletions. Yet some structures with $\mathcal{O}(\log n)$ weight cost insertions exist, like BB[α] trees [NR72], skip lists [Pug89] or weight-balanced B-trees [AV96]. These structures can be used to obtain a $\mathcal{O}(\log n)$ solution to the list order problem, both amortized or worst-case, depending on the structure used.

[1] Personal communication with Martin Farach-Colton and Erik Demaine, July 2013.

[2] According to [BBC$^+$12].

Amortized Solution of List Labelling with Polynomial Labels

We describe an algorithm with amortized complexity from [BCD$^+$02]. The advantage of this algorithm is that it does not explicitly maintain a tree, only the list itself.

Theorem 3.4. *The list labelling with polynomial labels can be solved with amortized $\mathcal{O}(\log n)$ relabellings per insert.*

Proof. We call any interval $j \cdot 2^i, j \cdot 2^i + 1, \ldots, j \cdot 2^i + 2^i - 1$ a *range* of size 2^i. These ranges correspond to nodes of perfect binary tree whose number of leaves is a power of two.

Let $1 < \delta < 2$ be a constant. We say a range of size 2^i is *in overflow* if it contains more than δ^i labels.

Consider inserting a new node in the list between nodes n and m. If $\ell_n + 1 < \ell_m$, we assign it the label $\lfloor (\ell_n + \ell_m)/2 \rfloor$. If $\ell_n + 1 = \ell_m$, we have no label to assign to the new node. We therefore find (by traversing the list in appropriate directions) the smallest range containing ℓ_n which is not in overflow. Then we relabel the labels contained in this range evenly. Consequently, the range of size 2 containing ℓ_n is not in overflow and we have a label to assign to the inserted node (possibly after decreasing ℓ_n by one, if it is last in the discussed range).

It is easy to check that the resulting labels have at most $\lceil \log_\delta n \rceil$ bits. To show that the amortized complexity of insert is $\mathcal{O}(\log n)$, consider relabelling a range of size 2^i. This relabelling takes $\mathcal{O}(\delta^i)$ time and after such relabelling, both child subranges contain $\delta^i/2$ labels. Another relabelling of the whole range occurs when one of these children contains more than δ^{i-1} labels, which happens after at least $\delta^{i-1}(1 - \delta/2)$ inserts. Therefore, it is enough for a given range to charge a constant time for each insert belonging to it. Because a label belongs to $\mathcal{O}(\log_\delta n)$ ranges, the result follows. For experiments with different δ see the original paper [BCD$^+$02]. $\square$

Worst-Case Solution of List Labelling with Polynomial Labels

We now describe an algorithm performing $\mathcal{O}(\log n)$ relabellings in the worst-case.

Theorem 3.5. *The list labelling with polynomial labels can be solved with $\mathcal{O}(\log n)$ relabellings per insert in the worst-case.*

Proof. Our algorithm is based on a *weight-balanced B-tree* described in [AV96], which we adapt for our purposes.

The leaves of the tree are at the same depth and weight of a node of depth i is between 2^{i-1} and 2^{i+1}, where a *weight* of a node is the number of leaves in its subtree. Consequently, the height of the whole tree is at most $1 + \lfloor \log_2 n \rfloor$ and every node has at most eight children, because the weight of a child is at least one eighth of the weight of its parent.

The children of a node are assigned 10-bit labels, such that the order of the children and the order of the labels is the same. The label of a leaf is a catenation of labels of the nodes on the path from the root to the leaf itself. We explicitly maintain both the child labels and the leaf labels.

We show that we can insert leaves to the tree and maintain the leaf labels, such that only $\mathcal{O}(\log n)$ of them are changed during an insertion, and the ordering of the leaves and the labels of the leaves is the same.

When inserting a new leaf, we insert it to the parent of its sibling. Then we check the ancestors of the new leaf, starting from the lowest level, and if we find one with too big weight, we split it. To split a node v of depth i we divide its children in two groups, the *left children* and the *right children*, such that the weights of those two groups are as close as possible. Then we create nodes v_l with the left children and v_r with the right children, insert these nodes to the parent of v instead of v and continue by checking the weight of the parent of v.

If we split the children of v equally, the weight of both v_l and v_r would be 2^i. Nevertheless, there can be a child of weight 2^i, half of it missing in one group and half of it surplus in the other group, so the weight of v_l and v_r is in the range between $2^i - 2^{i-1} = 2^{i-1}$ and $2^i + 2^{i-1} = (3/2)2^i$. Therefore it is in the required range.

If a split has happened, we need to update the child labels and also the leaf labels. We update the child labels immediately, storing both the original and updated labels for every child node. Nevertheless, the leaf labels are updated gradually. Of course, the order of the labels and the order of the leaves is preserved all the time.

We start by describing the child labels. The first child is assigned the label $1\,000\,000\,000_2$. When a child v is split into v_l and v_r, the label of v_l is the average of the label of the left sibling of v (or 0 if it does not exist) and the label of v; the label of v_r is the average of the label of v and the label of the right sibling of v (or $10\,000\,000\,000_2$ if it does not exist). Because a node has at most 8 children, 8-bit labels are enough. We elaborate on the two remaining bits later.

When a node v splits, we cannot update all the leaf labels in its subtree. Instead, we start two relabelling processes. The *left relabel* relabels the leaves in the subtree of v_l and the *right relabel* relabels the leaves in the subtree of v_r. We describe the left relabel in detail, the right relabel is analogous.

To relabel a leaf of v_l, 20 bits of the label have to be changed – the 10 bits corresponding to v change to the label of v_l and the 10 bits corresponding to the label of the child of v containing this leaf change to the label of the child of v_l containing this leaf. When we split v, we associate the left relabel with v_l (and right relabel with v_r) and during following inserts into any subtree of v_l we relabel 4 leftmost not relabelled leaves. Because the weight of v_l is at most $(3/2)2^i$, there must be at least 2^{i-1} inserts into v_l before the next split, so we relabel all leaves in the subtree of v_l before the v_l splits.

Notice that during the left relabel, both the original and the new label of any leaf preserve the ordering – there are no other leaf labels in between. Moreover, because the label changes are local and involve only two levels of the tree, relabellings on different levels do not interfere but in one case – if a split happens to a node whose parent is undergoing a relabel.

If v_l is undergoing left relabel and a child w of v_l splits, we cannot label the leaves in the subtrees of w_l and w_r using the updated child labels of w_l and w_r in v_l, because that would corrupt the ordering of the leaf labels with respect to the not yet relabelled leaves. Instead, for the not yet relabelled leaves, we use the original child label w had as a child of v. Because children of v_l can split at most twice before the left relabel finishes (they split after at least 2^{i-2} inserts), two spare bits of the original child labels are enough to solve this.

We recapitulate the insert operation. After inserting the new leaf to appropriate place, we successively check its ancestors. If there is a left or right relabel associated with an ancestor, we perform 4 its relabellings. If the weight of an ancestor v is too big, we split it and associate the left and the right relabels with v_l and v_r.

The last problem we have to deal with is splitting of the root node. If the root node r splits into r_l and r_r, we add a level to the tree – we create a new root node r_0 and add r_l and r_r as its children. Nevertheless, r has no child label, which is the same as if it has zero label. We therefore have to associate a right relabel with r_l, but only after all the leaves of r_r are relabeled. We do this by associating a special right relabel to r_r working twice as fast as the normal relabel. When it finishes, we associate a right relabel to r_l which also works twice as fast. This way, a relabel is done before any of the r_l and r_r split again.

To conclude, the leaf labels use $10(1 + \lfloor \log_2 n \rfloor)$ bits, preserve the ordering of leaves and, during an insert, at most $\mathcal{O}(\log n)$ leaf labels need to be changed. $\square$

3.2.3 Linear Labels

For completeness, we also present lower and upper bounds of list labelling using labels from range of size linear in the number of list elements, although we do not use these bounds further below.

In some cases, the polynomial labels are too large. Consider the *dense sequential file maintenance* problem – we maintain a file consisting of a sequence of fixed size records, which we want to insert and delete at any place in the file. For these operations to be efficient, we allow the file to be sparse – to store also unused records in the file, provided the size of the file is linear in the number of used records. This corresponds closely to list labelling – the label of the i-th node can be considered an index of the i-th record in the sparse file.

When the labels are bounded by $c \cdot n$ for $c > 1$, an algorithm using $\mathcal{O}(n \log^2 n)$ relabellings for n list insertions, i.e., with amortized $\mathcal{O}(\log^2 n)$ relabellings per operation, is described in [IKR81]. This algorithm was made worst-case by [Wil92], using at most $\mathcal{O}(\log^2 n)$ relabellings per an insertion to the list.

Also the case of labels bounded by $c \cdot n$ for $c = 1$ has been explored. List labelling with this label bound is known as *perfect labelling.* An algorithm using $\mathcal{O}(n \log^3 n)$ relabellings for n list insertions is presented in [BS07]. The worst-case variant of this algorithm does not yet exist, as far as we know.

All mentioned algorithms are optimal – matching lower bounds for labels bounded by $c \cdot n$ for both $c > 1$ and $c = 1$ are proven in [BKS12].

3.3 List Order Problem

As in the list labelling, in the list order problem we maintain a linked list. Nodes can be inserted anywhere in the list. Our goal is to be able to compare two nodes with respect to their order in the list, most preferably in constant time.

The list order problem can be solved easily by list labelling. We maintain labels associated with the nodes of the list and use these labels to perform the comparison of the nodes. Unfortunately, this results in the $\Omega(\log n)$ complexity of the insert operation, as described in the previous section.

Nevertheless, there are better algorithms if we do not require explicit labels and use only the comparison function. A solution with $\mathcal{O}(\log^* n)$ worst-case complexity of insert was described in [Die82]. An improved algorithm with $\mathcal{O}(1)$ amortized complexity of insert was published by [Tsa84]. Ultimately, a solution with worst-case $\mathcal{O}(1)$ complexity of insert was devised in [DS87].

In this section we describe both the amortized and worst-case constant time solution. The algorithm is based on [DS87], but is slightly simplified, because in the worst-case it uses the list labelling with polynomial labels (Theorem 3.5), instead of the list labelling with linear labels from [Wil92]. In both algorithms we also describe in detail how is the ordering of nodes represented. The representation details do not appear in any previous descriptions of the list order problem, and allow us to use the described algorithms in the persistent array construction in Chapter 5, instead of more complicated variants devised by Dietz in [Die89].

Amortized Solution of List Order Problem

Theorem 3.6. *We can solve the list order problem on RAM with amortized $\mathcal{O}(1)$ time complexity of the insert operation.*

Specifically, we can attach labels to the list nodes fulfilling the following properties. Each label consists of two parts, the prefix label and the sublist label, each consisting of $\mathcal{O}(\log n)$ bits. There are $\Theta(n/\log n)$ prefix labels, each shared by $\mathcal{O}(\log n)$ node labels. During an insert, amortized $\mathcal{O}(1)$ prefix labels and sublist labels are changed.

Proof. We solve the list order problem by using the list labelling with polynomial labels. To overcome the $\mathcal{O}(\log n)$ complexity of list labelling, we use a two-level structure.

We split the list into continuous sublists, each consisting of $\Theta(\log n)$ nodes. Specifically, each sublist consists of at least $\log n$ and at most $2\log n - 1$ nodes. Whenever we insert a node, we insert it to appropriate sublist, and if it now contains $2\log n$ nodes, we split the sublist into two, each consisting of $\log n$ nodes.

Each node is assigned a label in its sublist, called the sublist label. The sublist label has $\mathcal{O}(\log n)$ bits and is assigned using algorithm requiring no relabelling described in Theorem 3.3. The sublist labels are updated only during a split of a sublist. On a RAM machine, we can manipulate with these labels in constant time, therefore, we can split a sublist and assign new labels in $\mathcal{O}(\log n)$ time. Because a sublist is split only every $\log n$ insertions, the amortized number of sublist label changes is $\mathcal{O}(1)$ per insertion.

The whole sublists are labelled with polynomial labels, called the prefix labels, using the algorithm from Theorem 3.4 with amortized $\mathcal{O}(\log n)$ relabellings. When a sublist is split, one of the resulting sublists keeps the prefix label and the other sublist gets a new label. During the n insertions, there are $n/\log n$ sublist splits, so we need to label $n/\log n$ sublists, for which we need a total number of $\mathcal{O}((n/\log n)\log(n/\log n)) = \mathcal{O}(n)$ prefix label changes. In other words, there is amortized $\mathcal{O}(1)$ prefix label changes per insertion. Altogether, there is amortized $\mathcal{O}(1)$ number of prefix labels and sublist labels changes per insertion.

To perform a comparison of two nodes, we first compare their prefix labels, and if they are the same, we compare also the sublist labels. This is correct, because sublists always contain continuous sequence of the original list nodes. □

Worst-Case Solution of List Order Problem

The worst-case solution is based on the same idea as the amortized solution – we split the list into $n/\log n$ sublists and label the sublists using prefix labels and the nodes in the sublists using sublist labels. The labelling of sublists is done using the worst-case list labelling described in Theorem 3.5.

The problematic part is maintaining the sublists. The amortised solution performs a split whenever it is needed. Nevertheless, to achieve worst-case bounds, at least $\Omega(\log n)$ insertions must take place between two sublist splits, so that the $\mathcal{O}(\log n)$ relabellings of the prefix labels caused by the first sublist split can be performed.

Therefore, we use the following algorithm for maintaining the sublists. Every $\log n$ insertions, we choose the sublist containing most nodes and split it. This way, we maintain $n/\log n$ sublists and are able to perform the $\mathcal{O}(\log n)$ prefix label changes caused by a sublist split during the following $\mathcal{O}(\log n)$ insertions, changing $\mathcal{O}(1)$ prefix labels at a time.

The question is whether this algorithm gives any upper bound on the size of the sublists. This turns out to be true.

Theorem 3.7 (Theorem 5 from [DS87]). *Consider the following pebble game. In the game, there are n piles, initially empty, represented by nonnegative real numbers $x_1, \ldots, x_n$. The game proceeds by repeating the following steps:*

- *The adversary chooses nonnegative real numbers $a_1, \ldots, a_n$ such that the sum of them $\sum_i a_i = 1$ and sets $x_i \leftarrow x_i + a_i$.*
- *We find index i such that x_i is maximal and set $x_i \leftarrow 0$.*

Then for any strategy of the adversary, no x_i ever exceeds the value $H_{n-1} + 1$, where $H_n = \sum_{i=1}^{n} i^{-1} \leq \ln n + 1$, and this bound is tight.

To apply to our case, we use the following corollary.

Corollary 3.8 (from [Ram92]). *In a variant of pebble game where we can only perform $x_i \leftarrow \alpha x_i$ instead of zeroing, where $0 \leq \alpha < 1$, all x_i are bounded by $(1-\alpha)^{-1}(H_n + 1)$.*

Proof. Let M_α be the yet unknown bound on all x_i. We define another pebble game with $y_i = \max(x_i - \alpha M_\alpha, 0)$. The key observation is that after performing $x_i \leftarrow \alpha x_i$, we have $y_i = 0$. Thus we play zeroing game on y_i, consequently, all y_i are bounded by $H_n + 1$ and the result follows. $\square$

In our case the splitting of the sublists is just a pebble game with halving, where the adversary adds $\log n$ to the piles per turn. Although each split creates new nonempty sublist, we see from the proof of the Corollary that in the corresponding zeroing game, the sublist is represented by an empty pile. Therefore, when adding a sublist, we add an empty pile to the zeroing game and pretend that it was already there, just not used by the adversary. Accordingly, during n insertions, the size of any sublist is bounded by $\mathcal{O}(\log^2 n)$.

We are now ready to present the worst-case algorithm for list order problem.

Theorem 3.9. *We can solve the list order problem on RAM with $\mathcal{O}(1)$ worst-case time complexity of the insert operation.*

Specifically, we can attach labels to the list nodes fulfilling the following properties. Each label consists of three parts, the prefix label and two sublist labels, each consisting of $\mathcal{O}(\log n)$ bits. There are $\Theta(n/\log n)$ prefix labels, each shared by at most $\mathcal{O}(\log^2 n)$ node. During an insert, $\mathcal{O}(1)$ prefix labels and sublist labels are changed.

Proof. We maintain the continuous sublists as described – every $\log n$ insertions, we split the largest one, obtaining the bound $\mathcal{O}(\log^2 n)$ on the size of the sublists.

We label the sublists using so called prefix labels, by using the worst-case list labelling algorithm from Theorem 3.5. It is easy to modify that algorithm to perform an insertion in $\mathcal{O}(\log n)$ phases, each taking $\mathcal{O}(1)$ time and preforming $\mathcal{O}(1)$ relabellings, by processing a constant number of ancestors of the inserted node at a time.

What remains to be done is to label the nodes in the sublists. We cannot use the same sublist representation as in the amortized case, because the sublists contain up to $\mathcal{O}(\log^2 n)$ elements and we cannot operate on numbers with $\mathcal{O}(\log^2 n)$ bits in constant time. Instead, we represent a sublist as a two-level tree with nodes (root and its children) of degree $\mathcal{O}(\log n)$ and leaves representing the list elements. We label the children of a given node using the algorithm requiring no relabelling described in Theorem 3.3. The label of a leaf then consists of the two child labels of its predecessors, each of length $\mathcal{O}(\log n)$.

We maintain the sublists in a similar fashion as in Theorem 3.5. To insert an element into the sublist, we insert it to the corresponding bottom node v. When this node is of size $\log n$, we create two nodes v_l and v_r, which we insert to the root node before and after v. During the next $\frac{1}{3}\log n$ insertions to any of v, v_l and v_r, we move two leftmost children of v to v_l and two rightmost children of v to v_r. After these $\frac{1}{3}\log n$ insertions, v becomes empty and both v_l and v_r are of size at most $\frac{5}{6}\log n$.

Every $\log n$ steps we also have to split the largest sublist. In order to locate the largest one, we maintain a doubly-linked ascending list containing sizes for which there exists a sublist, and for each size, we maintain a doubly-linked list containing sublists of this size. We can easily update this structure during the sublist size changes (i.e., when increasing or decreasing sublist size by one and when adding an empty sublist) in $\mathcal{O}(1)$, and the structure allows us to locate the largest sublist in constant time.

To split the largest sublist, we split the root node the same way as we split its children. Therefore, when splitting sublist s, we create s_l and s_r and during the following $\log n$ insertions (not necessarily to this sublist), we move the children of s to both s_l and s_r.

Nevertheless, we need the prefix labels of s_l and s_r immediately after we start splitting s. Therefore, in the top-level structure containing the prefix labels, we keep spare prefix labels between any two used prefix labels and before the first and after the last used prefix label. When we start splitting s, we already have prefix labels for s_l and s_r prepared. During the next $\log n$ insertions, we insert two spare prefix labels in the structure, one before s_l and one after s_r, and we have all spare prefix labels prepared before the following sublist split.

There are several constants hidden in the asymptotic complexities which we did not explicitly quantify, e.g., the bound on the size of the sublists, but doing so is straightforward and does not help understand the algorithm, so we omit those. $\square$

3.4 Chapter Notes

This chapter defines the well-known list labelling and list order problems and describes optimal solutions for these problems, compiled from [DS87] and [BCD+02]. Our contributions are the following.

- An algorithm for worst-case list labelling in polynomial space from Theorem 3.5, presented in [Str09]. As far as we know, no such algorithm has been published (see beginning of Section 3.2.2 for details).

- Using this algorithm, we provide a slightly simpler solution to the worst-case list order problem in Theorem 3.9, presented also in [Str09]. The original solution from [DS87] uses the worst-case list labelling in linear space from [Wil92], which needs $\mathcal{O}(\log^2 n)$ relabellings per insertion.
- Both our amortized and worst-case solutions to list order problem in Theorems 3.6 and 3.9 provide detailed description of labels associated with the list nodes. This characteristics of the labels structure allows these algorithms to be used in the persistent array construction of Chapter 5, where an ordinary list order problem solution could not be used otherwise.

Chapter 4

Dynamic Integer Sets

Several persistent structures identify versions using bounded integers and we can improve the complexity of these structures, if we can efficiently manipulate sets of bounded integers.

In this chapter we describe several data structures capable of representing a set of n nonnegative integers smaller than a known bound U. The structure must provide *insert*, *search* and *delete* operations. A search for an integer key x should return the maximum key in the set which is smaller than or equal to x. This operation is sometimes called the *predecessor* and its opposite is the *successor* operation.

Quite commonly, some data is associated with the set elements. In this case, the structure is in fact a finite map from the integer keys to associated values. Associating data with set elements is usually trivial (it is so in all structures discussed in this chapter) and does not influence the representation of the set, therefore, we do not distinguish set and maps and talk about sets only in this chapter.

A straightforward implementation of a dynamic integer set is any balanced binary tree, such as an AVL tree [AVL62] or a red-black tree [GS78]. The complexity of all operations is then $\mathcal{O}(\log n)$, which is independent of U.

Lower bound of the *search* complexity in terms of n only is $\Omega(\sqrt{\log n/\log\log n})$, as proved in [BF01]. In [AT07] an implementation is given, which performs insert, search and delete in this time.

For the purpose of using dynamic integer sets in persistent structures, we are interested in implementations performing the required operations in $\mathcal{O}(\log\log U)$ time. Such structures are exponentially faster than binary search trees in case when $U = \mathcal{O}(n^k)$. We describe two such structures in this chapter.

4.1 Van Emde Boas Trees

The Van Emde Boas trees were introduced in [BKZ76] and [vEB77] and have been simplified since. We give a full description of the structure here, because we improve its space complexity in Theorem 4.4.

Definition 4.1. A *Van Emde Boas tree (vEB tree or vEBT)* is a data structure capable of representing a set of integers in range $0 \mathinner{..} U-1$. Let $M = \lfloor\sqrt{U}\rfloor$. The vEB tree for range $0 \mathinner{..} U-1$ consists of a root node containing

- a minimum and a maximum element,
- an array of vEB trees $c_0, c_1, \ldots, c_M$ for range $0 \mathinner{..} M$,
- a summary vEB tree s for range $0 \mathinner{..} M$.

The child tree c_i represents elements in the range $iM \mathinner{..} iM+M-1$ of the original tree, except the minimum and maximum elements of the original tree, which are not represented by any child tree c_i. The summary tree s contains indices of non-empty trees c_i.

Theorem 4.2. *Let U be positive integer. A vEB tree representing range $0 \mathinner{..} U-1$ provides operations insert, search and delete in $\mathcal{O}(\log\log U)$ time and uses $\mathcal{O}(U)$ space.*

Proof. Because the range represented by a vEBT node decreases by a square root each level, the whole vEB tree has depth $\mathcal{O}(\log\log U)$.

Inserting an element to an empty tree or to a tree with one element can be done in constant time by setting the minimum and/or the maximum element appropriately. Otherwise, we first check whether the new element is a new minimum or maximum element and if so, we swap the new element with the previous minimum or maximum element. Then we locate the child tree c_i which should contain the new element, and continue recursively. If the child tree was empty, we also insert i into the summary tree s. Important observation is that when this happens, inserting the element in the empty tree c_i is done in constant time, so there is only one recursive call and the complexity of insert is as requested. Manipulation with the summary tree is the reason why the minimum and maximum elements are not stored in the child trees.

Deleting an element is similar. If the tree contains at most two elements, we perform the deletion by modifying the minimum and/or the maximum element. Otherwise, if we should delete the minimum element of the node (maximum is analogous), we find the first non-empty c_i in constant time using the minimum element of summary tree s, swap the minimum element of the whole node with the minimum of c_i and proceed by deleting the minimum of c_i. In the last case the element being deleted is stored in the child tree c_i. We delete it recursively and if the c_i is now empty, we delete i from the summary tree s. When this happens, deleting from c_i took constant time, so same as with insert, there is only one recursive call.

When performing a predecessor (successor is analogous) search for x, we start by checking the trivial cases – if x is smaller than the minimum, predecessor does not exist, and if x is larger or equal to the maximum, we return the maximum element. Otherwise we find the child tree c_i, where x belongs. If it is non-empty and x is not smaller than its minimum element, we recursively search for the predecessor of x in c_i. If not, the predecessor of x is not in c_i and we use the summary tree s to find the non-empty predecessor tree c_j where the predecessor of x would be. When such tree exists, we return its maximum element, and when c_j it does not exist, we return minimum of the whole vEB tree. As before, there is at most one recursive call on each level, resulting in the claimed $\mathcal{O}(\log\log U)$ complexity.

Let $S(U)$ denote the space needed by a vEBT representing range $0\,..\,U-1$. Using the definition of the vEBT, $S(U) = \mathcal{O}(\sqrt{U}) + (\sqrt{U}+1)S(\sqrt{U}) \le c\cdot\sqrt{U} + (\sqrt{U}+1)S(\sqrt{U})$. We show that $S(U) \le c\cdot(U-2)$ by induction. The base case holds any constant $U > 2$. Otherwise, we use the induction hypothesis to get

$$S(U) \le c\cdot\sqrt{U} + c\cdot(\sqrt{U}+1)(\sqrt{U}-2) = c\cdot(\sqrt{U} + U - 2\sqrt{U} + \sqrt{U} - 2) = c\cdot(U-2).$$

□

Improving The Space Complexity of vEBT

The performance of vEBT is very good when measured using U only. But usually we need to express the complexity using the number of elements n. Assuming $U = \mathcal{O}(n^k)$, the time complexity of vEBT operations is still $\mathcal{O}(\log\log n)$, but the memory complexity is $\mathcal{O}(n^k)$, which is unsatisfactory. If amortized complexity is sufficient, we can do better, as mentioned for example in the Section 3.3 of [Die89].

Theorem 4.3. *Let U be positive integer. The space complexity of a vEBT can be improved to $\mathcal{O}(n)$ at the cost of making the complexity $\mathcal{O}(\log\log U)$ of insert and delete amortized. The unchanged worst-case complexity of search is $\mathcal{O}(\log\log U)$.*

Proof. The reason for excessive space usage of a vEBT is that in every node there is an array of $\sqrt{U}$ child trees. If we stored the child trees using space linear in the number of non-empty child trees, the whole vEBT would fit in $\mathcal{O}(n)$ space. But we need to be able to work with the child trees in constant time, otherwise the time complexity of vEBT operations increases.

We can use dynamic perfect hashing [DKM+94] to represent the non-empty child trees. That allows us to store the non-empty child trees in linear space, find a child tree for a given index in worst-case constant time, and insert and delete a child tree in amortized constant time. □

It is an open problem whether a vEBT can be modified to fit in $\mathcal{O}(n)$ space while retaining worst-case $\mathcal{O}(\log\log U)$ complexity of its operations. When fitting in linear space, the current best complexity of insert, search and delete is $\mathcal{O}((\log\log U)^2/\log\log\log U)$, as we describe in the next section.

Nevertheless, it is possible to retain the $\mathcal{O}(\log\log U)$ worst-case complexity and improve the space complexity.

Theorem 4.4. *Let U be positive integer and let $k = f(U)$ for any function $f(U) \leq \log U$. Assuming we can allocate a block of uninitialized memory of arbitrary size in constant time, we can improve the space complexity of a vEBT to $\mathcal{O}(n \cdot U^{1/k})$, while increasing the worst-case complexity of insert, search and delete to $\mathcal{O}(k + \log\log U)$.*

Proof. Instead of storing one large vEBT, we keep many small vEBTs v_i. Namely, every non-empty range $i{\cdot}U^{1/k} \,..\, i{\cdot}U^{1/k}{+}U^{1/k}{-}1$ of the set is stored in the vEBT v_i. Because there are at most n such trees, all non-empty v_i fit in $\mathcal{O}(n \cdot U^{1/k})$ space.

What remains to be shown is how to store these small vEBTs in order to be able to perform required operations. Conceptually, we store the v_i in a trie (see [Fre60] for description) with branching factor of $U^{1/k}$, such that all the vEBTs v_i are stored at the level k.

In order to save space, we represent only non-empty v_i and use a compressed trie, sometimes also called a Patricia trie [Mor68]. In an ordinary trie, every edge has an one element label. In a compressed trie, the nodes with exactly one child are left out and the edge labels are sequences of elements, see Figure 4.1 for illustration. Because the indices of v_i fit in a machine word, so do the labels of the edges in the Patricia trie, even if the edge is a catenation of $k-1$ edges of the regular trie.

Because there are at most n leaves, the Patricia trie has at most $n-1$ internal nodes, each containing:

- an array of length $U^{1/k}$ containing empty and non-empty children,
- a summary vEB tree s containing indices of non-empty children.

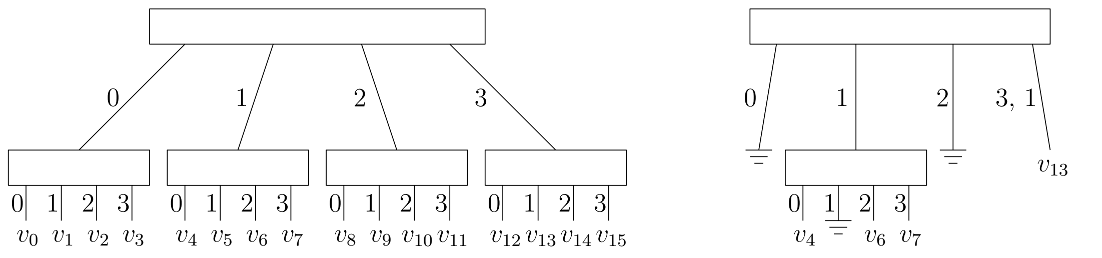

0
1
2
3
0 1 2 3
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14} v_{15}
3, 1
v_{13}
v_4 v_6 v_7

Figure 4.1: Representing $v_0, \ldots, v_{15}$ in a regular and Patricia trie, assuming only v_4, v_6, v_7 and v_{13} are non-empty.

A node occupies $\mathcal{O}(U^{1/k})$ memory, so the whole Patricia trie fits in $\mathcal{O}(n \cdot U^{1/k})$ space. When creating a node, we need to allocate an array of length $U^{1/k}$ in constant time. The array can be uninitialized, because only the elements present in the summary vEB tree are ever accessed, therefore, it is a reasonable assumption.

What remains to be shown is how to perform the required operations. When inserting element x, we start by searching for $v_{\lfloor x/U^{1/k} \rfloor}$ in the trie, inserting x if it is found. Otherwise we create $v_{\lfloor x/U^{1/k} \rfloor}$ containing x, insert it to the trie and update the summary vEBT in the changed trie node.

Deleting an element x is equally simple. We start by locating $v_{\lfloor x/U^{1/k} \rfloor}$ and deleting x from it. If the vEBT is now empty, we remove it from the trie and update the summary vEBT in the changed trie node.

When searching for x, we try locating the vEBT v_j for $j = \lfloor x/U^{1/k} \rfloor$. If it exists and x is at least its minimum, we perform the search in this vEBT. Otherwise we locate the non-empty vEBT v_i with the largest $i < j$ as follows: We start in the deepest visited trie node. Using the minimum element of the summary vEBT, we can check in constant time whether a non-empty vEBT v_i for $i < j$ exists. If it does not, we continue the search recursively in the parent of the trie node. If such v_i exists, we use the summary vEBT to find the non-empty v_i with largest i and we return its maximum.

When performing any described operation, we spend constant time in all but at most one visited trie nodes. In at most one node we perform either insert, delete or search in its summary vEBT tree. We also perform either insert, delete or search in at most one vEBT v_i. Therefore, the total complexity of all described operations is $\mathcal{O}(k + \log\log U)$. □

Corollary 4.5. *Let U be positive integer. We can improve the space complexity of a vEBT to $\mathcal{O}(n \cdot U^{1/(c \cdot \log\log U)})$ for any constant c while preserving the $\mathcal{O}(\log\log U)$ worst-case complexity of insert, search and delete, if we can allocate a block of uninitialized memory of arbitrary size in constant time.*

4.2 Exponential Trees

Exponential trees are a technique by Andersson and Thorup [AT07] for converting static polynomial space integer set structures to dynamic linear space data structures. The resulting dynamic integer set structures are currently the best structures that fit in $\mathcal{O}(n)$ space.

We provide a short description because of the connection of this technique to the vEB trees.

Theorem 4.6 (Theorem 1.1 from [AT07])**.** *Let $k \geq 2$. Suppose a static search structure on d integer keys can be constructed in $\mathcal{O}(d^{k-1})$ time and space so that it supports searches in $S(d)$ time. We can then construct a dynamic linear space search structure that with n integer keys supports insert, delete and searches in time $T(n)$, where*

$$T(n) \leq T(n^{(k-1)/k}) + \mathcal{O}(S(n)).$$

Outline of the proof. The structure of exponential trees is similar to the vEB trees. The exponential tree with n elements can be defined recursively:

- the root has degree $\Theta(n^{1/k})$,
- the subtrees are exponential trees over $\Theta(n^{(k-1)/k})$ keys,
- an S-structure containing *splitter* for each subtree is stored in the root. If a child t has splitter s and its successor has splitter s', all keys in t are greater or equal to s and less then s'.

The difference between exponential trees and vEB trees is how the keys are split between the subtrees. While in vEB trees we uniformly split the key range beforehand, exponential trees assign the key ranges to the subtrees dynamically and store the splitters in a static structure in each exponential tree node. That results in a linear space usage, while increasing the time complexity, because we have to consult $\Theta(\log\log n)$ local S-structures during any operation with the exponential tree.

If during insert, search and delete we spend only constant time and constant number of operations with local S structure in every visited node, the time complexity is

$$\begin{aligned}
T(n) &= \mathcal{O}(S(\mathcal{O}(n^{1/k}))) + T(\mathcal{O}(n^{(k-1)/k})) \\
&= \mathcal{O}(S(\mathcal{O}(n^{1/k}))) + \mathcal{O}(S(\mathcal{O}(n^{(k-1)/k^2}))) + T(\mathcal{O}(n^{(k-1)^2/k^2})) \\
&= \mathcal{O}(S(n)) + T(n^{(k-1)/k})
\end{aligned}$$

In order to get rid of the asymptotic, we applied the recurrence to itself and used the fact that for $n = \omega(1)$ we have $n \geq \mathcal{O}(n^{1/k}) \geq \mathcal{O}(n^{(k-1)/k^2})$ and $n^{(k-1)/k} \geq \mathcal{O}(n^{(k-1)^2/k^2})$.

As the local S structure in a node with n keys can be built in $\mathcal{O}(n^{(k-1)/k})$ time, it occupies at most that much space. Therefore, the space complexity of an exponential tree with n elements is

$$C(n) = \mathcal{O}(n^{(k-1)/k}) + \sum_{n_i} C(n_i) \quad \text{where } n_i = \mathcal{O}(n^{(k-1)/k}) \text{ and } \sum n_i = n$$

and it can be shown by induction that $C(n) \leq c(n - n^{(k-1)/k}) = \mathcal{O}(n)$.

Of course, the fundamental question is how to maintain the static S-structures while inserting and deleting elements. Conceptually, it is a standard task – we use partial rebuilding technique [Ove83b]. Similarly to B-trees [BM72], we can maintain the shape of an exponential tree just by splitting a node into two adjacent nodes, and by joining two adjacent nodes into a new one. When we split or join nodes, we start building S-structures for the new nodes. We build them gradually, performing constantly many steps each time we insert or delete a key in any subtree of the new node. Meanwhile we keep the old S-structures and use them for navigating. It is possible to schedule the splits and joins such that a node containing m keys is split or joined after at least $\Omega(m^{(k-1)/k})$ keys are modified, allowing us to fully build the new local S structure, because the time needed for the creation of the local S structure is $\mathcal{O}((m^{1/k})^{k-1})$. Therefore, we perform the rebuild fast enough to always keep only constant number of local S structures in any node.

Although the technique is a standard one, the details are highly technical and delicate. An interested reader can find an elaborate proof in [AT07]. $\square$

Static Structures for Exponential Trees

There are several suitable static search structures that can be used with exponential trees. As an example, we can use a static variant of a vEB tree. It is similar to the vEBT modification from the Theorem 4.3 – we store the children of every vEBT node using *static perfect hashing*. Static perfect hashing can be thought of as a "perfect sparse array" – it is able to store m integer keys bounded by B in polynomial space with respect to m, and find a given key in constant time. It can be constructed deterministically by several methods, probably the best of them developed by [Ram96], which constructs the static perfect hashing in $\mathcal{O}(m^2 \cdot B)$ time. That allows us to create a static vEB tree in polynomial time – at first we store children of a vEBT node in a balanced binary tree, converting them to static perfect hashing after all keys have been inserted to the vEBT, all in $\mathcal{O}(n \log n \log\log U + n \log\log n \cdot U)$ time.

Using this static variant of vEB tree, we can create linear-space exponential tree with $\mathcal{O}(\log\log n \cdot \log\log U)$ complexity of operations.

In the following theorem we summarize best known static search structures, together with the consequential exponential trees.

Theorem 4.7 (Corollary 1.4 from [AT07])**.** *In polynomial time and space, we can construct static search structure over n keys bounded by U supporting searches in*

$$\begin{array}{ll} (A) & \mathcal{O}\left(\sqrt{\log n / \log\log n}\right) \\ (B) & \mathcal{O}\left(\log\log U / \log\log\log U\right) \\ (C) & \mathcal{O}\left(1 + \log n / \log W\right) \end{array}$$

time on a RAM, where W is the word length.

Using these structures, we can create fully dynamic linear space search structures supporting insert, delete and searches in time

$$\begin{array}{ll} (A) & \mathcal{O}\left(\sqrt{\log n / \log\log n}\right) \\ (B) & \mathcal{O}\left(\log\log n \cdot \log\log U / \log\log\log U\right) \\ (C) & \mathcal{O}\left(\log\log n + \log n / \log W\right). \end{array}$$

Proof. The static structure (A) is described in [BF01]. The static structure (B) is an improved static vEBT just described, also described in [BF01].

The static search structure (C) is based on fusion trees and described in the Lemma 4.2 of [AT07]. It is based on the central part of the fusion tree [FW93], which is a static search structure with the following properties: *For any $d = \mathcal{O}(W^{1/6})$, a static data structure containing d keys can be constructed in $\mathcal{O}(d^4)$ time and space, such that it supports search queries in $\mathcal{O}(1)$ worst-case time.* This structure itself cannot be used in exponential trees because of the limit on the number of elements, but we can create a static B-tree where each node has degree $\Theta(W^{1/6})$ and the children are stored using the described static search structure. Such a B-tree has height of $\Theta(\log_{W^{1/6}} n) = \Theta(\log n / \log W + 1)$.

The resulting exponential trees are just straightforward applications of Theorem 4.6 to the described static search structures. $\square$

4.3 Chapter Notes

This chapter is mainly a summary of known results. Our contribution is the Theorem 4.4, where we improve the space complexity of a vEBT tree while preserving the time complexity of its operations. This theorem is then used during construction of persistent array in Corollary 5.7.

For curious reader, we summarize the relevant lower and upper bounds:

- Corollary 3.10 from [BF01]: Suppose a set of n integers each bounded by U is represented using $n^{\mathcal{O}(1)}$ memory cells of $\log^{\mathcal{O}(1)} U$ bits. Then predecessor search takes $\Omega(\sqrt{\log n / \log\log n})$ in the worst case.

  The authors also describe a data structure matching this lower bound using quadratic memory. Using exponential trees, a dynamic structure fitting in linear space can be created.

- Corollary 3.9 from [BF01]: Suppose a set of n integers each bounded by U is represented using $n^{\mathcal{O}(1)}$ memory cells of $2^{\log^{1-\Omega(1)} U}$ bits. Then predecessor search takes $\Omega(\log\log U / \log\log\log U)$ in the worst case.

  Once again, the authors also describe a data structure matching this lower bound using quadratic memory. Nevertheless, the resulting exponential tree does not match this complexity – when $U = \mathcal{O}(n^k)$, it has complexity of $\mathcal{O}((\log\log U)^2 / \log\log\log U)$.

- The previous lower bound was improved in [PT06], assuming the data structure fits in $n \cdot \log^{\mathcal{O}(1)} n$ bits and $U = \mathcal{O}(n^k)$. The predecessor search then takes $\Omega(\log\log U)$ in the worst case.

  It is an open problem whether such a structure exists. The current best result has the already mentioned $\mathcal{O}((\log\log U)^2 / \log\log\log U)$ complexity.

Chapter 5

Persistent Arrays

Arrays are without any question the most frequently used data structure, present in nearly any algorithm. An array is an object containing n elements, numbered from 1 to n, allowing to both *lookup* or *update* an element using its index in constant time.

Definition 5.1. A *persistent array* a is a structure supporting two operations:

- $\textit{lookup}(i, a_v)$ returns the value of $a_v[i]$, the i-th element of array a_v,
- $\textit{update}(i, x, a_v)$ returns new array a_w obtained from a_v by changing the $a_v[i]$ to x, i.e., $a_w[i] \leftarrow x$.

We call a_v a *version* of the array and we write only v when the array is clear from context. In this chapter, n is always the size of the array and m is the number of updates performed on the array.

However, obtaining efficient persistent array implementation is quite complicated – all described methods of creating efficient persistent data structures work for linked data structures, i.e., for structures with bounded out-degree of their nodes, and therefore these methods cannot be used to obtain an efficient persistent array.

Having a persistent array with constant time operations would have important consequences. The most commonly used model of a computer, the word RAM [Sch80], considers the memory to be just a big array. Therefore, using a persistent array with constant time operations, we could make any data structure with worst-case bounds fully persistent without changing its time complexity. This would include for example graphs and hash tables. In addition, all algorithms with worst-case bounds could be modified to be fully persistent without any increase in complexity, like path finding, game solvers, program simulations detecting data races or deadlocks, and many others.

Nevertheless, in Section 5.2 we show that even partially persistent array lookup complexity has a lower bound of $\Omega(\log\log n)$. That implies that although all worst-case data structures and algorithms can be made fully persistent using persistent arrays, the time complexity is increased by a multiplicative factor of $\Omega(\log\log n)$. Therefore, persistent arrays are not an efficient universal method for creating persistent structures, because for many ephemeral structures the persistent variants have the same time complexity, e.g., for linked data structures as described in Chapter 2.

Still, a persistent array is a very important data structure, because arrays are crucial to many basic algorithms, e.g., depth first search, shortest path algorithms, radix sort or dynamic programming algorithms like longest common subsequence (used to compute difference of two files) and shortest path in acyclic graph. It can also be used to create persistent variants of data structures which cannot be made persistent using the known methods, i.e., structures with unbounded indegree or out-degree. These structures include matrices, hash tables or graphs, to mention a few.

In this chapter we present several fully persistent array implementations. The first implementation of Section 5.3, similar to the one in extended abstract [Die89], achieves amortized complexity of $\mathcal{O}(\log\log m)$ for update and same worst-case complexity for lookup. In Section 5.4 we provide two algorithms with worst-case bounds, the first with $\mathcal{O}((\log\log m)^2/\log\log\log m)$ and the second with $\mathcal{O}(\log\log m)$ complexity for both lookup and update. Nevertheless, the latter algorithm has superlinear memory complexity.

We slightly improve the complexity of persistent array operations in Section 5.5. We modify all our implementations to depend not on the number of array modifications m, but instead on $\min(m, n)$. We also show an implementation with $\mathcal{O}(\sqrt{\log k/\log\log k})$ complexity, if no array element is modified more than k times the average.

The garbage collection of persistent arrays, i.e., freeing of unused array version, is an issue disregarded by other implementations. Usually, some or even majority of array versions become inaccessible over time and could be freed. The problem is that although only one element is modified in a particular version of the array, the other array elements are also accessible via a lookup on this version. Therefore, even if an array version is not directly accessible, it cannot be deleted, because it is referenced by an array version modifying a different element. We describe efficient algorithms for recognizing the removable array versions in Section 5.6.

5.1 Related Work

Because arrays are omnipresent data structures, there have been many attempts to provide efficient persistent arrays. The persistent arrays have been demanded for both the ability to access and modify any version of the array, and also as a functional counterpart to array in functional languages. The ultimate goal was a fully persistent array with constant time lookup and update, although this was naturally never achieved, because of the lower bound presented in Section 5.2.

Probably the most common persistent array implementation is a full binary tree or balanced binary tree with integer keys [Hug85, Mye84]. Such implementations have $\mathcal{O}(\log n)$ worst-case time and space complexity for update and $\mathcal{O}(\log n)$ worst-case time complexity for lookup. These implementations have several advantages – they use only the path copying technique and therefore no explicit version identifiers are necessary. In addition, they can be implemented in a purely functional language, i.e., without side-effects. This implementation is optimal in the pointer machine model, which is sometimes used as a basis for functional languages. We present such implementation in Chapter 6, dealing with the choice of the best branching factor of the tree and also comparing it to other alternatives.

A different approach, a *shallow binding* technique, was devised by [Bak78a, Bak91]. One version of the persistent array is called *active* and the persistent array is represented by an ephemeral array corresponding to the active version and by a version graph containing all modifications to the persistent array. If the persistent array is used single-threadedly, all operations work in constant time. Nevertheless, if a version different from active is accessed and this version is in distance d from the active version in the version tree, it takes $\mathcal{O}(d)$ time to apply/undo the changes to the ephemeral array so that it corresponds to the required version. The same idea appears also in [AHN88] and [Hug85], called *version tree arrays* or *trailer arrays*.

The shallow binding is improved in [Chu92] to keep the version graph of size at most $\mathcal{O}(n)$, by splitting the version tree into two independent persistent arrays. If such a split is performed every $\Theta(n)$ modifications to the array, the amortised cost of the split is constant. Also the lookups which are grouped in voluminous read sequences of size $\Omega(n)$ can be performed in amortised constant time. Nevertheless, both lookup and update still have $\mathcal{O}(n)$ worst-case complexity, if a version far from the active one is required.

Further improvements are presented in [Chu94]. Randomisation is used and the version tree is split not periodically every $\Theta(n)$ updates, but with probability of $1/n$ during every lookup. This improves the average lookup time, however, at the cost of space complexity. If an array with m modifications is read frequently, eventually it is split into m independent arrays, requiring $\mathcal{O}(nm)$ memory. That makes the method highly impractical.

The logarithmic complexity of update and lookup is optimal in pointer machine model. However, in a word RAM model, which is the theoretical model closest to the real computers, we can do better. The asymptotically fastest existing persistent array implementation with amortised $\mathcal{O}(\log \log m)$ complexity of update and lookup is outlined in an extended abstract [Die89]. Independently for each element, a structure containing all its modification is kept. A fast structure described in Theorem 4.3 allowing operations in $\mathcal{O}(\log \log m)$ in the RAM model is used. The array versions are compared using the list order problem solution from Theorem 3.6. In Section 5.3 we describe similar, but simpler persistent array implementation with the same complexity.

The idea of storing modifications separately for each element appeared already in [Coh84]. Here a partially persistent array implementation was suggested which stored a linked list of modifications for each element. Such implementation can perform updates in constant time, but lookup operation complexity is linear in the number of modifications to the element in question.

One of the most practical persistent array implementations was suggested in [OB97]. Again, the array versions are labelled using the list order problem, Theorem 3.6. For each element, all its modifications are stored in a splay tree [ST85], indexed by the version label. The size of the version tree is maintained to be $\mathcal{O}(n)$ by splitting the version tree when it grows too large. Such implementation has $\mathcal{O}(\log n)$ amortized complexity of lookup and update. However, both operations have constant complexity if the array is used single-threadedly or if the elements are accessed in such a way that no element is accessed more than k times the average.

Persistent Arrays Provided in Functional Languages

Considering that there is no persistent array providing constant time operations, designers of functional languages have to decide how to handle persistent arrays in the language.

The simplest solution is to provide no arrays, neither ephemeral nor persistent. This was common in early functional languages like Miranda [Tur86], Standard ML or the first Lisp dialects.

Another approach, taken by Haskell [PJ$^+$03], is to provide *monolithic arrays.* A monolithic array is a standard array stored as a continuous memory block, allowing constant time lookup. Nevertheless, every update creates a fresh copy of the array. Such monolithic arrays are purely functional data structures and are very suitable if lookups and updates do not interleave much, for example when creating histograms or in dynamic programming. But many algorithms like a depth first search cannot be implemented efficiently using monolithic arrays.

Many functional languages with eager evaluation provide ephemeral arrays with destructive updates, as Caml [WF94], its .NET successor F# or newer Lisp dialects like Common Lisp and Scheme. Nevertheless, such structures require side effects and interfere with the type system, causing for example the *value restriction* in Caml [WF94] and its successors.

Another approach was taken by Haskell [PJ$^+$03] and Concurrent Clean [HP92]. In these languages, arrays with destructive updates are also provided, but in such a way, that the compiler ensures every array is used single-threadedly. In other words, there are no side effects to worry about and the type system is not affected. In Concurrent Clean, this is achieved by *unique types*, a concept very similar to linear types of [Wad90b]. The unique types guarantee that there is at most one reference to any structure. Haskell chose a different approach utilizing monads [Wad90a, Wad92], namely the ST monad [LPJ94] in the case of arrays. Computations in a monad are ordered and all array references correspond always to the latest version of the array. Both approaches allow destructive updates of the array to take place, because the original version cannot be accessed any more.

5.2 Lower Bound on Persistent Array Lookup

Consider a partially persistent array with n elements and $m = n^\gamma$ modifications, where γ is a constant fulfilling $1 < \gamma \leq 2$.

Assuming the space complexity of the array is $\mathcal{O}(m \log^k m)$ for a constant k, we show that the lower bound on the lookup complexity in this partially persistent array is $\Omega(\log\log n)$. This lower bound holds in the cell probe model introduced by [Yao81] and the complexity of update is not limited in any way.

The proof of the lower bound is based on a reduction of the predecessor search problem to partially persistent array lookup. The existence of this reduction was first mentioned by Demaine, Langerman and Price [DLP08], stating that "according to personal communication with Mihai Pătraşcu, 2008, persistent arrays have a lower bound of $\Omega(\log\log n)$, based on a predecessor lower bound of [PT07]." As far as we know, the reduction itself has never been published, that is why we present it in Theorem 5.3.

Predecessor Search Problem

In the Predecessor search problem we are given a set Y of n integers of ℓ bits each, and the goal is to answer predecessor queries, i.e., to evaluate, for a given integer x, *predecessor*$(x) = \max\{y \in Y \mid y \leq x\}$.

Theorem 5.2 (from [PT06] and [PT07]). *If we use S w-bit words to represent Y and define $a = \log \frac{S}{n} + \log w$, then the lower bound in the cell probe model on a predecessor query is, up to constant factors,*

$$\min \begin{cases} \log_w n \\ \log \frac{\ell - \log n}{a} \\ \frac{\log(\ell/a)}{\log(a/\log n \cdot \log(\ell/a))} \\ \frac{\log(\ell/a)}{\log(\log(\ell/a)/\log(\log n/a))} \end{cases}.$$

This lower bound applies to both deterministic algorithms (proven in [PT06]) and to probabilistic algorithms (proven in [PT07]). For each case, there is also an algorithm performing predecessor queries with that complexity.

In the important case when $w = \ell = \gamma \log n$ for a constant $\gamma > 1$ and near linear space, i.e., $S = n \cdot \log^{\mathcal{O}(1)} n$, the optimal search time is $\Theta(\log \ell)$.

Reduction to Persistent Array Lookup

The predecessor search problem is tightly connected to persistent array lookup, because *lookup*(i, a_v) returns a value $a_u[i]$ stored by an update operation, where u is the closest predecessor of version v modifying index i. However, this observation by itself results only in a lower bound of $\Omega(1)$, because in the corresponding reduction, ℓ is $1 \cdot \log n$ and the predecessor search problem lower bound is trivial. However, we can use the fact that the array lookups with the same array version behave differently on different indices, which can be used to make the reduction nontrivial.

Theorem 5.3. *The predecessor search problem can be solved using a lookup in a partially persistent array.*

Specifically, let Y be a set of n integers of ℓ bits each and let the number of bits in a word $w = \ell = \gamma \log n$ for $1 < \gamma \leq 2$. Using a partially persistent array with $n^{\gamma/2}$ elements and n modifications, we can answer a predecessor search in Y using a lookup in the partially persistent array and $\mathcal{O}(1)$ additional work.

If the partially persistent array occupies $\mathcal{O}(n \log^k n)$ space after performing n modifications, the Theorem 5.2 implies that the lower bound on the lookup operation is $\mathcal{O}(\log \log n)$ in the cell probe model.

Proof. Let $N = n^{\gamma/2}$. We create a partially persistent array a of N elements and an ephemeral array t of N elements in the following way. Initially, every element of a is set to -1. Then for each $i \in \{0, \dots, N-1\}$ we perform the following:

- For each $y \in Y$ such that $y \bmod N = i$ we modify the current version of a by storing a value y at the index $\lfloor y/N \rfloor$.
- Afterwards, we store the current version of a in $t[i]$.

Finally, let p be another auxiliary ephemeral array of N elements and set $p[i] \leftarrow$ *predecessor*$(i \cdot N)$ for $0 \le i < N$.

The auxiliary arrays occupy $\mathcal{O}(N)$ space. Because there are n modifications performed by the persistent array, the total space complexity is $\mathcal{O}(n \log^k n)$.

An example of this construction is in Figure 5.1.

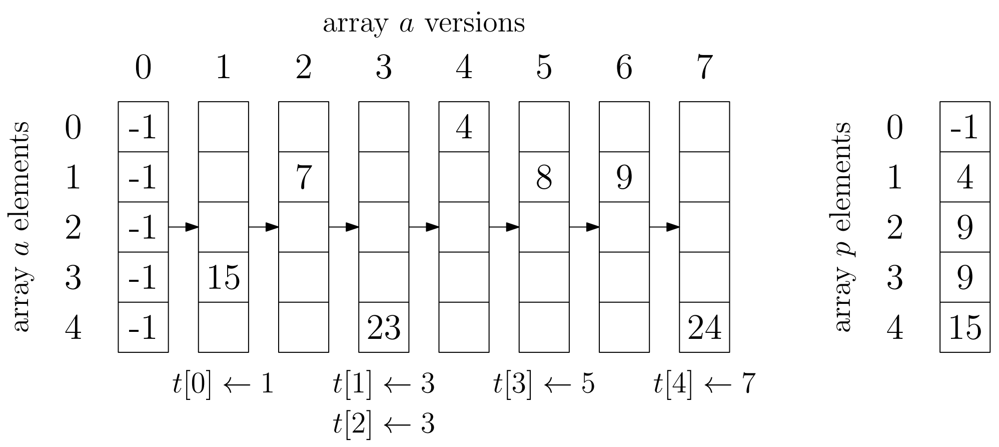


Figure 5.1: Example of a reduction of predecessor search to partially persistent array lookup with set $Y = \{4, 7, 8, 9, 15, 23, 24\}$ and $N = 5$.

To answer a *predecessor(x)* query, we perform a lookup in a at index $\lfloor x/N \rfloor$ and version $t[x \bmod N]$. If the result is not -1, then the predecessor is the result of this lookup. If the result is -1, then the predecessor is stored in $p[\lfloor x/N \rfloor]$. Therefore, we can compute the predecessor using one lookup and $\mathcal{O}(1)$ extra work.

Consequently, the lower bound for predecessor search problem can be applied, showing that under the specified conditions, the time complexity of partially persistent array lookup operation is at least $\Omega(\log \ell)$, which is $\Omega(\log \log n)$. □

5.3 Amortized Persistent Array

We now develop a fully persistent array implementation with amortised time complexity $\mathcal{O}(\log \log m)$. Our implementation is similar to the one in the extended abstract [Die89]. The key difference of our implementation is the carefully chosen connection of the list order problem and the van Emde Boas trees via Invariant 1, together with the representation details provided in the algorithms solving the list order problems (Theorems 3.6 and 3.9). This way, the implementation is simpler, better suited for the worst-case variant in the next section and we are able to use ordinary list labelling, instead of the *weighted* list labelling used in [Die89].

5.3.1 Partially Persistent Array

We first consider a simpler case of partially persistent arrays. For each element, we store the changes to this element separately in a structure indexed by integers. We use the vEB tree defined in Definition 4.1, which allows manupulating a set of integers of size bounded by U and as shown in Theorem 4.2, it supports insertions, deletions and finding predecessors and successors in time $\mathcal{O}(\log \log U)$. We use the space efficient variant of vEBT from Theorem 4.3, which has linear space complexity, at the cost of the complexity of insert and delete becoming amortized.

We implement partially persistent array as an array of vEB trees. We label the array versions by consecutive integers starting from 1. The update operation to the latest version v is implemented as an insertion into the appropriate vEBT with key $v+1$. The lookup operation at version v is implemented by predecessor search in the appropriate vEBT with key v.

This partially persistent array implementation has $\mathcal{O}(\log \log m)$ worst-case lookup complexity and $\mathcal{O}(\log \log m)$ amortized update complexity and requires $\mathcal{O}(m+n)$ space.

5.3.2 Fully Persistent Array

Theorem 5.4. *We can implement a fully persistent array with n elements such that during m modifications of the array, the worst-case lookup complexity is $\mathcal{O}(\log \log m)$ and the amortized complexity of update is $\mathcal{O}(\log \log m)$.*

The initial version of the array can be created in $\mathcal{O}(n)$ time and after m modifications the array occupies $\mathcal{O}(n+m)$ space.

Proof. In contrast to the partially persistent case, where the versions are identified by consecutive integers starting from 1, the representation of the versions and navigation in the version tree is more complicated in fully persistent structures.

We use the solution described in Chapter 3. For navigation in the version tree we use its linearized variant, the version list. Specifically, we use the *extended* version list described in Definition 3.2. In this list, every array version v is represented using two versions v_+ and v_-. When modifying an element i in the version v, we create new versions w_+ and w_- in the extended version list and we set the required value of the element i in the version w_+. With the version w_- we associate the value the element i had in the version v. This is needed, because in the extended version list, the successors of w_- are not successors of w in the version tree, and therefore the version w_- is used to “undo” the effect of w_+ outside the subtree of the version w.

We use the list order problem to compare the versions in the extended version list. Using Theorem 3.6 we can compare the versions in the extended version list in constant time. We also know that the versions are assigned labels of length $\mathcal{O}(\log m)$.

A straightforward solution would be to again use the vEB trees indexed by the labels assigned to the versions in the extended version list. Nevertheless, during an insertion to the extended version list, amortized $\Theta(\log m)$ labels change and this is optimal, as discussed in Section 3.2.2.

Nevertheless, even if $\Theta(\log m)$ labels change, we can exploit the way how the changes are performed. As described in Theorem 3.6, the versions are assigned labels consisting of two parts, the *prefix label* and the *sublist label*, both $\mathcal{O}(\log m)$ bits long. One prefix labels is shared by $\mathcal{O}(\log m)$ versions and we call a sequence of versions sharing the same prefix label a *sublist*. We also know that during each insert to the extended version list, amortized $\mathcal{O}(1)$ prefix labels and sublist labels change.

For every array element, we store all its modification in the following way. If there are less than $\log m$ modifications to this array element, we store the modifications in a balanced binary search tree. If there are more than $\log m$ modifications, we group continuous sequences of modifications into *buckets* of size $\Theta(\log m)$. We call the first version in the bucket a *bucket leader*. The buckets are stored in the vEB tree indexed by the label of the bucket leader and the buckets themselves are represented using a balanced binary search tree.

It is crucial to create a connection between the buckets and sublists:

Invariant 1. *Every sublist contains at most one bucket leader.*

This invariant guarantees that each prefix label is used at most once as an index in any vEBT. Therefore, during the insertion to the extended version list, only amortized $\mathcal{O}(1)$ vEBT keys are changed.

We now describe the resulting implementation of a fully persistent array in detail. The representation is illustrated in Figure 5.2.

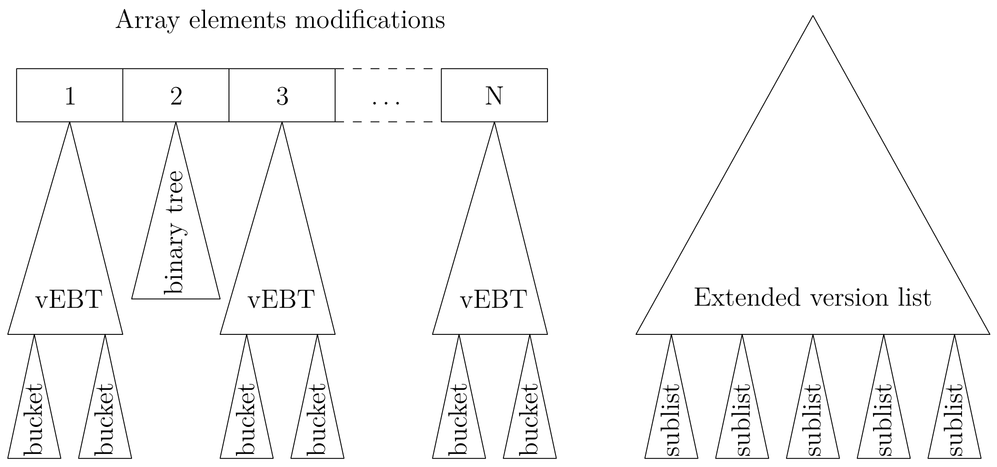


Figure 5.2: Fully persistent array implementation

We maintain an extended version list of versions. Also, for each array index, we store versions which modified this index in the following structure. If there are at most $1 + \log m$ modifications, we store them in a balanced binary search tree. Otherwise we group the versions into buckets of size at most $2 \log m$ and store the buckets in the vEB tree indexed by the label of the bucket leader. There are $\mathcal{O}(m/\log m)$ buckets and each bucket is represented using a balanced binary search tree.

The vEB trees work only for keys bounded by a fixed U. When creating a vEB tree, we choose smallest U in a form $U = 2^{2^i}$ such that all current labels are less than U. We then keep this bound until a label larger than U is assigned. At that moment, we add a level to the vEB tree by creating a new root node containing the current vEB tree as a child with index 0. This can be done in constant time and squares the range the vEB tree can handle. That way, $\log U$ is at most twice the length of used labels, therefore, $\log U$ is $\mathcal{O}(\log m)$.

The *create(n, a)* operation creates a persistent array from specified array a. It constructs an extended version list containing one initial version v and for each array element i, a binary search tree is created, containing value $a[i]$ associated with version v. This can be accomplished in $\mathcal{O}(n)$ time and space and the Invariant 1 holds as there are no buckets yet.

To perform a *lookup*(i, v), we search the modifications of array element i for a predecessor of label of version v (inclusive). If the modifications are stored in a tree, the size of the tree is at most $\log m$ and we can find the predecessor of v in $\mathcal{O}(\log \log m)$ time. Otherwise, we at first find the predecessor in a vEBT, locating the bucket containing the predecessor, and find the predecessor in the bucket. Because the range of vEBT is $m^{\mathcal{O}(1)}$ and the size of bucket is $\mathcal{O}(\log m)$, the lookup has $\mathcal{O}(\log\log m)$ complexity.

To carry out an *update*(i, x, v), we first perform $x' \leftarrow$ *lookup*(i, v) and if $x = x'$, there it nothing to do. Otherwise we insert versions w_+ and w_- in the extended version list just behind v. These inserts take amortized $\mathcal{O}(1)$ and during them there is amortized $\mathcal{O}(1)$ of prefix label changes and sublist label changes. If this causes a bucket leader label to change, we modify the label in the corresponding vEBT too. According to Invariant 1, there is at most one such bucket leader for a prefix label change, therefore, it is enough to perform amortized $\mathcal{O}(1)$ changes in the vEB trees in time $\mathcal{O}(\log\log m)$.

We now insert two modifications of array element i – value x in version w_+ and value x' in version w_-. If the modifications are stored in a balanced search tree, we insert the two new versions into the tree in $\mathcal{O}(\log\log m)$ time. If the resulting tree contains more than $1 + \log m$ versions, we create a bucket out of the whole tree and create a new vEBT containing this bucket indexed by the label of the bucket leader.

If the modifications of array element i are already stored in buckets in vEBT tree, we find the bucket where the versions w_+ and w_- belong using a predecessor search in the vEBT and then insert these versions in the bucket. If the size of the bucket grows larger than $2\log m$, we split the bucket in two equal parts, create a new bucket out of the second part and insert the new bucket to the vEBT with the bucket leader label. All this can be done in $\mathcal{O}(\log\log m)$ time.

In any case, if a new bucket is added to vEBT, we check whether the Invariant 1 still holds. If not, there is now a sublist containing two bucket leaders. We restore the invariant by splitting the sublist into two parts, each containing one bucket leader, creating a new sublist from the second part. This sublist split is performed exactly as the sublist split in Theorem 3.6. Therefore, the split takes $\mathcal{O}(\log m)$ time and causes amortized $\mathcal{O}(\log m)$ prefix label changes and $\mathcal{O}(\log m)$ suffix label changes. According to Invariant 1, these label changes cause at most amortized $\mathcal{O}(\log m)$ changes of bucket leader labels, therefore, we can update the vEB trees in $\mathcal{O}(\log m \log\log m)$ time. Because every bucket splits after $\log m$ insertions, there are at most $\mathcal{O}(m/\log m)$ buckets, thus we have to perform the invariant preserving subtree split at most $\mathcal{O}(m/\log m)$ times, spending a total time of $\mathcal{O}((m/\log m)\log m \log\log m)$ on these splits. Therefore, their amortized cost is $\mathcal{O}(\log\log m)$ per update.

We conclude that the amortized cost of an update is $\mathcal{O}(\log\log m)$. $\square$

5.4 Worst-Case Persistent Array

We now improve the described persistent array implementation to guarantee worst-case bounds instead of only amortized bounds. Note that in the original paper [Str09] we incorrectly used the vEB trees with amortized complexity in the worst-case construction, which we amend here.

Theorem 5.5. *We can implement a fully persistent array with n elements such that during m modifications of the array, the worst-case complexity of lookup and update is $\mathcal{O}(\log\log m + S(m))$, where $S(m)$ is the complexity of operations of a dynamic integer set containing m integers bounded by $m^{\mathcal{O}(1)}$.*

The initial version of the array can be created in $\mathcal{O}(n)$ time and after m modifications the array occupies $\mathcal{O}(n+m)$ space.

Proof. The amortized implementation described in Theorem 5.4 uses amortization in several places:

- The vEB tree requires amortization to be space efficient.
- The list order problem algorithm of Theorem 3.6 has amortized complexity.
- The maintenance of buckets and Invariant 1 has amortized complexity.

We therefore replace the vEB trees with an equivalent structure with worst-case bounds. We can use Theorem 4.4 or Theorem 4.7, although none of these structures has the same bounds as the vEB trees and it is an open problem whether such structure exists. That is why we parametrize the complexity of the resulting persistent array by $S(m)$, the complexity of dynamic integer set operations.

Instead of the amortized solution to the list order problem we use the already described worst-case solution from Theorem 3.9. This algorithm maintains the sublists by splitting the largest sublist periodically every $\Theta(\log m)$ insertions and uses Corollary 3.8 to prove that the sizes of the sublists are bounded by $\mathcal{O}(\log^2 m)$.

We can use the same algorithm to maintain the buckets – if we split a bucket every $\log m$ updates, there will be $\mathcal{O}(m/\log m)$ buckets and the size of the buckets will be bounded by $\mathcal{O}(\log^2 m)$, according to Corollary 3.8.

We now describe the resulting implementation of a worst-case fully persistent array in detail. The representation is very similar to the representation of amortized fully persistent array illustrated in Figure 5.2.

We maintain an extended version list of versions using the algorithm from Theorem 3.9, which is split to $\Theta(m/\log m)$ sublists of size $\mathcal{O}(\log^2 m)$. Also, for each array index we store versions which modified this index in the following structure. Initially we store the modifications in a balanced binary search tree. At some point we may change the representation – we group the versions into buckets of size at most $\mathcal{O}(\log^2 m)$ and store the buckets in a dynamic integer set indexed by the label of the bucket leader, i.e., by the label of the first version in the bucket. We maintain $\Theta(m/\log m)$ buckets and represent each bucket using a balanced binary search tree.

We preserve an invariant which is a slight relaxation of the Invariant 1:

Invariant 2. *Every sublist contains at most one bucket leader, except for one sublist, which may contain two bucket leaders.*

The *create* and *lookup* operations are performed exactly as in the amortized implementation from Theorem 5.4.

To carry out an *update*(i, x, v), we first perform $x' \leftarrow$ *lookup*(i, v) and if $x = x'$, there it nothing to do. Otherwise we insert versions w_+ and w_- in the extended version list just behind v. These inserts take $\mathcal{O}(1)$ worst-case time and cause $\mathcal{O}(1)$ prefix label changes and sublist label changes. If this causes a bucket leader label to change, we modify the label in the corresponding dynamic integer set too. According to Invariant 2, there are at most two such bucket leaders for a prefix label change, therefore, it is enough to perform $\mathcal{O}(1)$ changes in the dynamic integer sets in time $\mathcal{O}(S(m))$.

We then insert two modifications of array element i – value x in version w_+ and value x' in version w_-. If the modifications are stored in a balanced search tree, we insert the two new versions into the tree in $\mathcal{O}(\log\log m)$ time. Otherwise, if the modifications are already bucketed, we find the corresponding bucket in $\mathcal{O}(S(m))$ time and insert the modifications to the bucket in $\mathcal{O}(\log\log m)$ time.

The sublists, buckets and Invariant 2 are maintained in phases, each of length $2\log m$. At the beginning of each phase, we split the largest sublist as described in Theorem 3.9, finishing in $\log m$ array updates. Then we find the largest balanced tree or bucket with array element modifications, split it in $\mathcal{O}(\log\log m)$ time and insert the new bucket in the dynamic integer set in $\mathcal{O}(S(m))$ time using the label of the bucket leader as a key. If the sublist containing the new bucket leader already contains a bucket leader, we split it into two, each containing one bucket leader. We perform this split exactly when splitting the largest sublist, finishing in $\log m$ array updates and thus ending the whole phase. Therefore, there are always $\Theta(m/\log m)$ sublists and buckets and the Invariant 2 holds all the time. All described operations take at most $\mathcal{O}(\log\log m + S(m))$ time during every array update.

Consequently, the worst-case complexity of an update is $\mathcal{O}(\log\log m + S(m))$. □

Corollary 5.6. *Utilizing the variant (B) of exponential tree from Theorem 4.7 as a dynamic integer set, we obtain a fully persistent array implementation with worst-case $\mathcal{O}((\log\log m)^2/\log\log\log m)$ complexity for both lookup and update.*

An array of n elements needs $\mathcal{O}(n+m)$ space after m modifications.

Corollary 5.7. *Utilizing the modified vEBT from Corollary 4.5 as a dynamic integer set, we obtain a fully persistent array implementation with worst-case $\mathcal{O}(\log\log m)$ complexity for both lookup and update.*

However, the space complexity is superlinear. To represent an array of n elements after m modifications, $\mathcal{O}(n + m^{1+1/(c\cdot\log\log m)})$ space is needed, for any fixed constant c.

We conclude this section with the apparent open problem:

Open Problem. It is an open problem whether there exists a fully persistent array implementation with worst-case $\mathcal{O}(\log\log m)$ complexity of both lookup and update, that would use $\mathcal{O}(n+m)$ space to represent an array of n elements after m modifications. The lower bound is $\Omega(\log\log n)$ according to Theorem 5.3, the upper bound is currently $\mathcal{O}((\log\log m)^2/\log\log\log m)$ according to Corollary 5.6.

This open problem is tightly connected to the existence of dynamic integer set with $\mathcal{O}(\log\log U)$ worst-case bound, which is an open problem discussed in Chapter Notes of Chapter 4 and in [AT07].

5.5 Improving Complexity of Persistent Array Operations

We can improve the complexity of array operations in case the number of modifications m is much larger than the number of array elements n. In that case, we can split the version tree of the array into independent pieces, each containing at most $\mathcal{O}(n)$ versions.

Theorem 5.8. *We can improve the amortized fully persistent array implementation from Theorem 5.4, so that the complexity of both lookup and update is $\mathcal{O}(\log\log\min(m,n))$.*

Proof. If there are less than $2n$ modifications of the array, $\min(m,n)$ is $\mathcal{O}(m)$ and the original implementation has the required complexity.

Whenever an array contains $2n$ versions, we split it. We start by finding the version v just in the middle of the version list. We then create a new persistent array containing the versions of the original array up to version v, by inserting these modifications one by one. Finally, we create another new persistent array, whose initial value is the value of the original array in version v. We then insert the modifications introduced by the versions v and all the following into this new array. Therefore, to perform a split, we create two new persistent arrays and perform n updates on both of them. The resulting complexity of a split is $\mathcal{O}(n \log \log n)$. Nevertheless, a split happens only after n updates of the array, making the amortized complexity of split $\mathcal{O}(\log \log n)$ per array update.

Consequently, if a persistent array is modified more than $2n$ times, we represent it using several independent arrays, each containing at most $2n$ modification. The complexity of these array operations is therefore $\mathcal{O}(\log \log n) = \mathcal{O}(\log \log \min(m, n))$, including the amortized cost of the splitting. □

We can augment the worst-case implementation in a similar way.

Theorem 5.9. *We can improve the worst-case fully persistent array implementation from Theorem 5.5, so that the complexity of both lookup and update is* $\mathcal{O}(\log \log \min(m, n) + S(\min(m, n)))$.

Proof. If there are less than $2n$ modifications of the array, $\min(m, n)$ is $\mathcal{O}(m)$ and the original implementation has the required complexity.

Whenever an array contains $2n$ versions, we start splitting it. During the following $n/2$ updates to the array, we create two new persistent arrays, spending $\mathcal{O}(1)$ time during every update. In the next $n/2$ updates, we start moving versions from the original array to the two new arrays, three unmoved leftmost versions to one of the arrays and three unmoved rightmost versions to the other one. When a version is moved to a new array, the lookups and updates with this version are performed in the new array only. After $n/2$ steps, all versions from the original array have been transfered, because there are at most $3n$ versions to be moved and we transfer 6 versions at a time. Also, the size of the new arrays is at most $(2n + n/2)/2 + n/2 = 7n/4$, if the last $n/2$ updates take place in one of the new array.

The time complexity of the split is 6 additional updates during an array update, the asymptotic complexity of update is therefore unchanged.

Consequently, if a persistent array is modified more than $2n$ times, we represent it using several independent arrays, each containing at most $2n$ modification. The complexity of operations of these arrays is therefore $\mathcal{O}(\log \log n) = \mathcal{O}(\log \log \min(m, n))$. □

When we split the version tree of the persistent array into pieces of size $\mathcal{O}(n)$, the persistent array operations have better complexity if the array elements are modified uniformly. That is often the case – for example, during various graph searches like a depth first search, every element of array marking already visited nodes is modified exactly once. Also if a graph has limited degree d, many algorithms modify values associated with every node at most d times, such as the shortest path algorithm. Therefore, improving array operations complexity in this way is quite useful.

Theorem 5.10. *If every array element is modified at most k times or if no array element is modified more than k times the average during every n array updates, the complexity of persistent array operations is $\mathcal{O}(\min(\sqrt{\log k/\log\log k}, \log\log n))$ and $\mathcal{O}(\min(\sqrt{\log k/\log\log k}, \log\log n + S(n)))$ for the amortized and worst-case variant, respectively.*

Notably, if k is a constant, the array operations work in constant time.

Proof. Consider the array implementations from Theorem 5.8 and Theorem 5.9 that split the version tree in order to limit its size to $\mathcal{O}(n)$. If every element is modified at most k times the average, there are at most $\mathcal{O}(k)$ modifications to every array element in any version tree of $\mathcal{O}(n)$ size.

Let k_0 be a value when $\sqrt{\log k_0/\log\log k_0}$ is equal to the original complexity of array operations. A rough estimate is $k_0 \approx \log^{\log\log n} n$. We modify the implementations in the following way. Until there are less than k_0 modifications to an array element, the modifications are stored in the variant (A) of exponential tree from Theorem 4.7. When there are more than k_0 modifications to an array element, we utilize the original representation instead. In the amortized variant we switch to the required representation immediately. In the worst-case variant we start building the required representation gradually in k_0 steps, inserting two modifications at a time, and use the existing representation until the required one is finished. In both cases, the change of the representation does not increase asymptotic complexity of the array update, because for $\Theta(k_0)$ modifications, both representations are equally efficient.

Consequently, the complexity of array operations is the better one out of $\mathcal{O}(\sqrt{\log k/\log\log k})$ and the complexity of the original implementation. □

5.6 Garbage Collection of a Persistent Array

When a data structure is used in a functional language, usually a garbage collector is used to free the unused data. During the garbage collection, structures still reachable by the program are marked and all remaining structures are freed.

At first consider a persistent array containing numbers. During the garbage collection, the garbage collector identifies the *reachable* array versions. Nevertheless, we cannot remove all other array versions, because there are dependencies between the versions – only one element modification is associated with a version, yet all array elements can be accessed using this version, i.e., another $n-1$ array versions are accessible using a single array version. An example is illustrated in Figure 5.3.

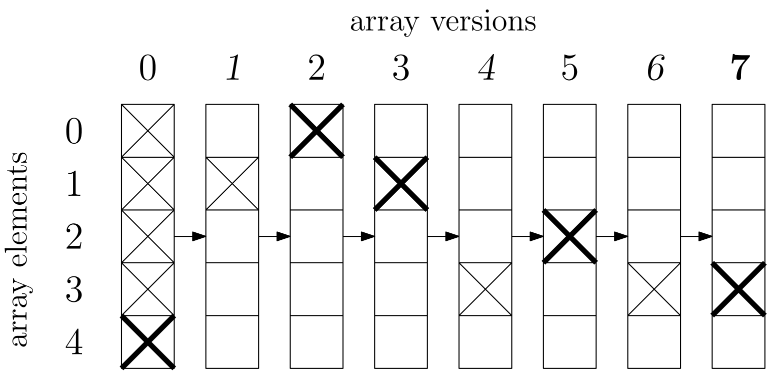


Figure 5.3: Example of reachable array versions. The cross represents an element modification. Even if only version 7 is marked as reachable by the garbage collector, versions 0, 2, 3 and 5 can be accessed using array elements in version 7. However, versions 1, 4 and 6 cannot be accessed any more and can be removed.

We call an array version *purgeable* if it is not marked as reachable by the garbage collector and it cannot be accessed using any reachable array version.

We present an efficient, even if a bit subtle, algorithm for recognizing the purgeable versions.

Theorem 5.11. *If the garbage collector marks the reachable versions of a persistent array of n elements with m modifications, we can identify the purgeable versions in $\mathcal{O}(m)$ time, using $\mathcal{O}(n+m)$ space. The algorithm applies to all described persistent array implementations.*

Proof. Let v be a version modifying array element i. Its descendants in the version tree are the versions between v_+ and v_- in the extended version list.

We call a descendant version u of version v *captured*, if it modifies index i or if there exists a version w modifying index i and w is a descendant of v and ancestor of u. In other words, if the order of these versions in the extended version list is $v_+ \; w_+ \; u_+ \; u_- \; w_- \; v_-$.

It follows that if u is a captured descendant of v, v is not accessible using a lookup on version u, because a lookup of the i-th element in version u returns the modification introduced by either u itself or w. Therefore, version v is purgeable, if all reachable descendant versions are captured.

To recognize purgeable versions, we traverse the version list from left to right, counting for each version v both the number of reachable descendants (nr_v) and the number of captured reachable descendants (nc_v). We use a stack and an array p of n elements, which are initially empty.[1] When we encounter a version v_+ modifying the index i, we

- push v and $p[i]$ to the stack,
- set $p[i] \leftarrow v$.

When we encounter a version v_-, we

- reset $p[i]$ to the value on the top of the stack,
- remove the $p[i]$ and v from the stack,
- classify v as purgeable iff $nc_v = nr_v$,
- add the number of reachable versions between v_+ and v_- (inclusive, i.e., nr_v or $nr_v + 1$, depending on whether v itself is reachable) to the parent of v, if any, which is on the top of the stack,
- if $p[i]$ is not empty, the reachable descendants of v (including v) are captured descendants of $p[i]$, therefore, we increase $nc_{p[i]}$ by the number of reachable versions between v_+ and v_-.

The algorithm works in linear time and space. Its correctness can be established by a straightforward induction on the subtrees of the version tree in the order these trees are left by a depth first search. $\square$

Nevertheless, this algorithm can be applied only when the array elements do not need to be examined by the garbage collector, e.g., if the array elements are non-referential types like numbers or characters.

If the array elements contain references to other structures, we have to augment the garbage collector to be able to recognize versions reachable from a given version. In the context of persistent arrays, if we are given version v, we have to identify all versions that can be returned during a lookup of any array element in the version v, or in other words, the predecessors of v among the modifications for every array element.

[1]After the algorithm finishes, all elements of p are empty again. Therefore, the same array can be reused many times and we do not include its initialization in the time complexity of the described algorithm, because p can be initialized when the persistent array itself is created.

It is straightforward to return all versions reachable from a given array version in time $\mathcal{O}(n \log\log m + nS(m))$. Nevertheless, the resulting garbage collection algorithm would be very inefficient, taking more than $\Omega(nm)$ time to process the whole array. Instead of returning all versions reachable from a given array version, we should return only the versions *not yet processed* by the garbage collector.

Theorem 5.12. *Consider a persistent array of n elements with m modifications. We can implement the following* unreported reachable versions *operation: for a given array version, return all versions reachable from this version (via lookup on any array element), omitting the versions that have been returned previously. The time complexity of d such operations is $\mathcal{O}(d + m \log n)$ and the space usage is $\mathcal{O}(m \log n)$.*

Proof. We start with a preprocessing, whose goal is, for each array version v, to create an *ascendant tree*, a tree of n versions reachable from version v. Specifically, for array version v, the i-th element of the ascendant tree is the version which is the (inclusive) predecessor of v among the versions modifying the i-th array element. In other words, it is the version whose associated element modification is returned during a lookup of the i-th array element in version v.

We represent the ascendant trees as partially persistent complete binary trees with n leaves, created using the path copying method of Section 2.1. We construct the ascendant trees successively for the versions in the version list, from the first to the last. The ascendant tree of the first version v_0 of the version list contains n times the version v_0. To create an ascendant tree of version v modifying index i, we update the ascendant tree of the previous version by changing its i-th leaf to v. Because we modify exactly one ascendant tree element in every step, all the ascendant trees have $\mathcal{O}(n + m \log n)$ nodes.

We can decrease the number of nodes a bit. We create a unique node n_0, which represents an arbitrary large tree with leaves containing version v_0. Therefore, the ascendant tree of the version v_0 is represented using just the node n_0. During the construction, the node n_0 can appear anywhere in the ascendant tree. Because the ascendant trees have a fixed shape, the size of tree represented by n_0 is defined by the position of n_0 within the ascendant tree. Therefore, all the ascendant trees now have a total of $\mathcal{O}(m \log n)$ nodes and the preprocessing takes $\mathcal{O}(m \log n)$ time and space.

In order to facilitate skipping of already reported versions, every ascendant tree node has an associated state, either *unreported* or *reported*. We maintain the following invariant: a leaf is marked as reported iff the version it contains has already been reported, and an internal node is marked as reported iff all leaves in its subtree have been reported.

To perform the *reported reachable versions* operation, we walk through the appropriate ascendant tree, not visiting children of a node marked as reported, and return the versions in unreported leaves. If u such versions are returned, the complexity of this search is $\mathcal{O}(u \log n)$, because every unreported internal node has at least one unreported leaf and there are $\mathcal{O}(\log n)$ predecessors for every leaf. Because every version is reported at most once, time needed to report all versions is $\mathcal{O}(m \log n)$.

After finding the unreported versions, we have to update the status of other ascendant tree nodes to restore the invariant. We do so by marking all leaves containing the unreported versions as reported. For each such leaf, we also recursively check whether the other child of its parent is reported, and if so, we mark the parent as reported and continue with the recursion. Because every node is marked at most once, all the status changes are performed in $\mathcal{O}(m \log n)$ time.

Therefore, the overall complexity of performing the *unreported reachable versions* operation d times is $\mathcal{O}(d + m \log n)$ using $\mathcal{O}(m \log n)$ space. □

We are now able to identify the purgeable versions during the garbage collecting phase, either in $\mathcal{O}(m)$ time by using more effective Theorem 5.11 for arrays not containing references, or in $\mathcal{O}(m \log n)$ time by using Theorem 5.12 for arrays containing references.

Nevertheless, removing the versions from the persistent array causes several complications. Firstly, the list labelling must support deletions, which can be accomplished by requiring a lower bound on the density of used labels in addition to the upper bound. Secondly, the number of sublists in the list order problem must remain $\Theta(m/\log m)$, by merging adjacent small sublists and by splitting sublists which were not added to, but got too big because many versions have been removed. Lastly, the number of buckets must be $\Theta(m/\log m)$. In addition to that, the Invariant 1 must be maintained. Although all these problems can be overcome, it would require solving a lot of technical details.

Instead, we rebuild the whole array, inserting only the reachable versions. This can be done in $\mathcal{O}(m)$ time. Because we already spent $\Omega(m)$ time on recognizing the purgeable versions, the rebuild does not increase asymptotic complexity. Alternatively, if we are performing the version tree splitting described in Section 5.5, we can integrate it with the removal of the unreachable versions – during the garbage collection we only mark unreachable versions and we leave them out during the splitting of the array.

5.7 Chapter Notes

This chapter is based on [Str09] and contains original results, most notably:

- worst-case fully persistent array implementation, Sections 5.4 and 5.5,
- persistent array garbage collection algorithms, Section 5.6.

Part II

Purely Functional Data Structures

Chapter 6

Persistent Array Implementation

Although the theoretical construction of fully persistent array described in Chapter 5 is nearly optimal, there is no guarantee that the resulting implementation is fast in practice. There are many parameters that must be taken into account – what is the real constant of the asymptotic time complexity and how big persistent arrays are usually being used.

In this chapter we try to find the best Haskell implementation of a fully persistent array, using benchmarking to compare speed of various implementations. For the sake of comparison we also include the non-persistent implementation and the standard data structures that can be used as a persistent array.

Our goal is to provide a genuine fully persistent structure which is used when many versions of the array are accessed and updated. Therefore, we provide no optimizations for the case the array is used single-threadedly. In such case, it is better to use an array in the ST monad [LPJ94], which can be updated destructively in-place.

6.1 Fully Persistent Array Implementation

The theoretical construction described in Chapter 5 is very likely not suitable for real implementation, for a combination of two reasons. The constant hidden in the asymptotic complexity is quite high, because of the dynamic integer sets and maintenance of the version labels. This could be compensated by a big difference in asymptotic complexity, but not in our case. The size of persistent arrays used in practice is almost certainly limited by $n \le 2^{32}$ elements. Nevertheless, the ratio of $\log n$ and $\frac{(\log\log n)^2}{\log\log\log n}$ for $n \le 2^{32}$ is at most 2.97, which is too little.

We therefore focus on $\mathcal{O}(\log n)$ implementations using path copying method of Section 2.1. Such implementations can be purely functional, avoid maintaining

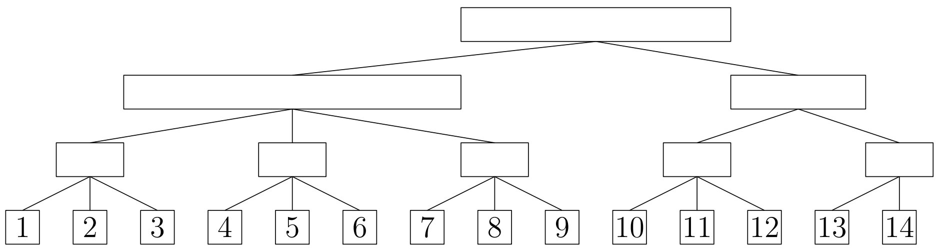


Figure 6.1: Example of persistent array with 14 elements represented using a ternary tree

the version list and can be garbage collected without any additional effort. In addition, the logarithmic complexity is asymptotically optimal if an array is represented using a linked data structure, i.e., using nodes with limited out-degree.

We represent persistent array with n elements using a tree with branching factor b. The array elements are leaves of the tree, they are on the same bottom level and a full tree is built on these leaves, i.e., all internal nodes have b subtrees, with the exception of the right spine of the tree. An example of a ternary tree representing a persistent array with 14 elements is displayed in Figure 6.1.

To implement the `index` operation, we start at the root, navigating in each step to a subtree containing the required element, until reaching the leaf containing the searched element. The complexity of the operation is $\mathcal{O}(\log_b n)$.

In order to `update` an element, we search for it in the same way, create a leaf with the new value and update the internal nodes visited during the search by copying them and updating the pointer to the modified subtree. Therefore, the update operation has $\mathcal{O}(b \cdot \log_b n)$ complexity.

We can also implement growing and shrinking of the array, by adding and removing leaves and suitable internal nodes of the tree.

To achieve good performance, the choice of b is important. Suitable form of b is a power of two, because then we can find the subtree containing the required element using only bit operations, instead of modulus and division.

An exemplary implementation of a persistent array using a tree with branching factor 4 follows. We start by a data type for persistent array, which consists of an array size, the zero based number of levels of the tree and the spine strict tree. Every node in the tree has always 4 elements, but only the first `size` leaves and corresponding internal nodes are guaranteed to be initialized.

```
data Array a = Array { size :: !Int, levels :: !Int, nodes :: !(Nodes a) }
data Nodes a = Nodes { n0 :: !(Nodes a), n1 :: !(Nodes a)
                     , n2 :: !(Nodes a), n3 :: !(Nodes a) }
             | Elems { e0 :: a, e1 :: a, e2 :: a, e3 :: a }
```

We use several helper methods for accessing and updating the Nodes data type. We define them to be able to improve the implementation further ahead.

```
indexNode :: Int -> Nodes a -> Nodes a; {-# INLINE indexNode #-}
indexNode 0 ns = n0 ns; indexNode 1 ns = n1 ns
indexNode 2 ns = n2 ns; indexNode _ ns = n3 ns

indexElem :: Int -> Nodes a -> a; {-# INLINE indexElem #-}
indexElem 0 es = e0 es; indexElem 1 es = e1 es
indexElem 2 es = e2 es; indexElem _ es = e3 es

updateNode :: Int -> Nodes a -> Nodes a -> Nodes a; {-# INLINE updateNode #-}
updateNode 0 ns val = ns {n0 = val}; updateNode 1 ns val = ns {n1 = val}
updateNode 2 ns val = ns {n2 = val}; updateNode _ ns val = ns {n3 = val}

updateElem :: Int -> Nodes a -> a -> Nodes a; {-# INLINE updateElem #-}
updateElem 0 es val = es {e0 = val}; updateElem 1 es val = es {e1 = val}
updateElem 2 es val = es {e2 = val}; updateElem _ es val = es {e3 = val}
```

Using these methods, we define the persistent array methods index and update without knowing the representation of Nodes:

```
index :: Int -> Array a -> a
index i a | i < 0 || i >= size a = error "Out of bounds"
          | otherwise = index’ (levels a) (nodes a)
  where index’ _ es@(Elems {}) = indexElem (i .&. 3) es
        index’ l ns@(Nodes {}) =
          index’ (l-1) $ indexNode ((i ‘shiftR‘ (2*l)) .&. 3) ns

update :: Int -> a -> Array a -> Array a
update i val a | i < 0 || i >= size a = error "Out of bounds"
               | otherwise = a { nodes = update’ (levels a) (nodes a) }
  where update’ _ es@(Elems {}) = updateElem (i .&. 3) es val
        update’ l ns@(Nodes {}) =
          let node = (i ‘shiftR‘ (2*l)) .&. 3
          in updateNode node ns $ update’ (l-1) (indexNode node ns)
```

We also provide a method for creating a new array of given size, all elements initialized to a given value:

```
create :: Int -> a -> Array a
create size val = build 0 (Elems val val val val)
  where build l n
          | size <= 4 ‘shiftL‘ (2*l) = Array { size=size, levels=l, nodes=n }
          | otherwise = build (l+1) (Nodes n n n n)
```

The resulting implementation can be trivially modified to use a different branching factor in a form of power of two and provide additional operations.

6.2 Choosing the Best Branching Factor

To complete our implementation, most suitable branching factor must be chosen to minimize the `update` operation complexity of $\mathcal{O}(b \cdot \log_b n)$. When considering the formula only, it is simple to check that the optimal value of b is e. Nevertheless, there are various different costs in the real implementation and therefore we use benchmarking to find the best value of b.

We used the CRITERION package [PkgCrit], a commonly used Haskell benchmarking framework. All benchmarks were performed on a dedicated machine with Intel Xeon processor and 4GB RAM, using 32-bit GHC 7.4.1. Detailed description of the benchmarking process used by the CRITERION package can be found in Section 8.2.1.

We performed two benchmarks, an `index` benchmark and an `update` benchmark. In the `index` benchmark we sequentially accessed all elements of the array without modifying them. In the `update` benchmark we sequentially modified all elements of the array. One implementation is chosen as a baseline and the execution times are normalized with respect to the selected baseline. For each implementation and each input, the mean time of 100 iterations is displayed, together with 95% confidence interval (which is usually not visible on the graphs as it is nearly identical to the mean). Each benchmark consists of several inputs. The size of input data is always measured in binary logarithms (so the input of size 10 contains 1024 elements). For every implementation a geometric mean of all times is computed and displayed in the legend. The implementations except for the baseline are ordered according to this mean. The detailed results and the benchmark itself are attached to this thesis and also available on the author's website `http://fox.ucw.cz/papers/`.

We benchmarked the following persistent array implementations:

- `Tree_C2`, `Tree_C4`, `Tree_C8`, `Tree_C16`, `Tree_C32`, `Tree_C64`: the implementation from Section 6.1 using a tree with a given branching factor.
- `Tree_A2`, `Tree_A4`, `Tree_A8`, `Tree_A16`, `Tree_A32`, `Tree_A64`: the preceding implementation has one inefficiency – the `indexNode` and `indexElem` methods do not run in constant time.[1] They need to perform the pattern matching according to the given index and then execute one of the method bodies returning the appropriate element. It would be better if there was only one body of `indexElem` returning the element asked for.

[1]At least not in GHC. But even if some smart optimization was performed by a compiler, `indexNode` would still most likely still need to perform some slow conditional jump.

Also the `updateNode` and `updateElem` suffer from the same needless pattern matching problem.

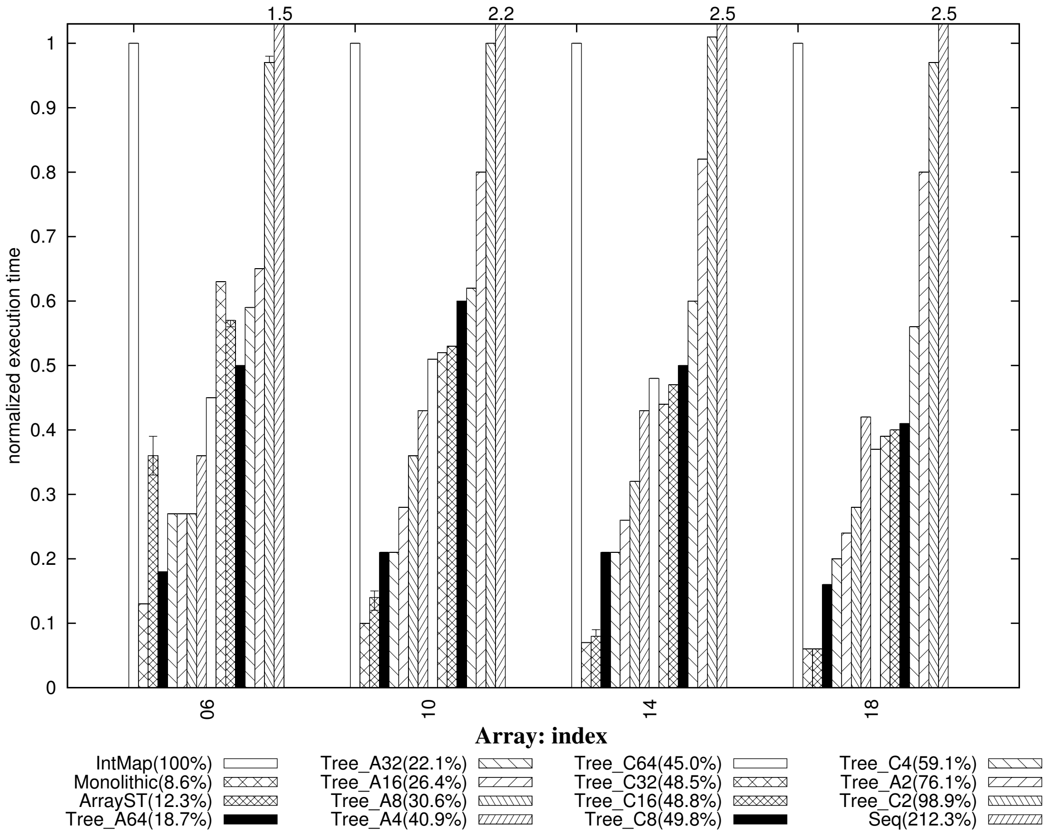
One way to solve this problem is to change the representation of `Nodes a` – the nodes and elements could be stored in an array instead of directly in the `Nodes a` constructor. That way the `indexNode` and `updateNode` methods could access the *i*-th element directly without pattern matching. This implementation indeed improves the `index` benchmark. Nevertheless, the updates of the array are quite slow in GHC and we decided not to include this implementation in the benchmarks because it is always worse then the following implementation.

Our solution to this inefficiency is providing the optimal `indexNode` and `indexElem` as primitives.[2] These primitives can access an *i*-th data constructor element using *i* as memory offset. We also provide improved `updateNode` and `updateElem` as primitives.

- `Monolithic`: a monolithic array [Wad87] from the standard ARRAY package. Although it is persistent, the whole array is copied during every update. Therefore, this array implementation is useful when we perform no or little array modifications. We call this implementation "read-only" further on.
- `ArrayST`: a non-persistent array also from the standard ARRAY package, featuring destructive updates in the ST monad [LPJ94]. Although the array is not persistent, we include it in our benchmark as a reference to see what is the overhead of the persistent implementations.
- `IntMap`: a persistent structure associating `Int` keys with values. The structure is described in Section 8.1.2 and is present in standard libraries. It is chosen as a baseline of the benchmark as it is the best persistent array implementation available in every Haskell installation.
- `Seq`: a persistent structure with broad functionality, also present in the standard library. It is based on 2-3 finger trees annotated with sizes [HP06] and is further described in Section 8.1.3.

The results of the benchmarks are displayed in Figure 6.2. The improved `Tree_A` implementations are superior to `Tree_C` implementations as expected, so we consider only those in the further discussion.

[2]At the moment, these "primitives" are still implemented in Haskell, bypassing the type checker. The implementation is highly experimental and GHC specific. If these methods establish their usefulness, they can be added as primitives to the compiler itself.



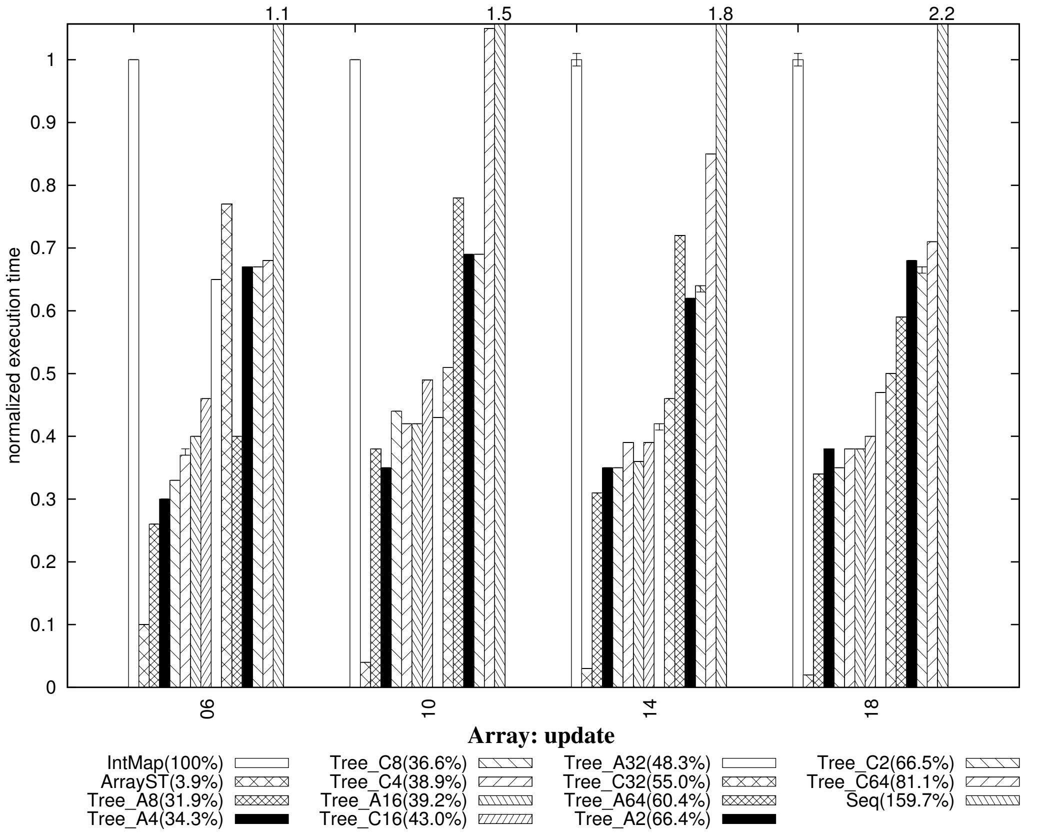


Figure 6.2: Benchmarks of persistent array implementations

In the `index` benchmark, the larger values of b perform better. If the indexing speed if the most pressing concern, indexing using the persistent array implementation with branching factor 64 is 2.2 times slower than indexing a non-persistent read-only array and only 1.5 times slower than indexing a non-persistent modifiable array.

In the `update` benchmark, the effect of different b on performance is nicely visible. The choices of b from the most effective are 8, 4, 16, 32, 64 and 2. Updating the persistent array with branching factor 8 is 8.2 times slower than updating a non-persistent array and indexing is 2.5 times slower.

6.3 Chapter Notes

In this chapter we present a persistent array implementation, which is to our knowledge fastest one available. These are original results.

Our implementation is fully persistent. However, there are many different array implementations which are either non-persistent, partially persistent or fully persistent but require some specific usage pattern to achieve good performance. The implementations of the latter kind include the DIFFARRAY package, a formerly standard library, implementing partially persistent array. The index and update time of the most recent version of the array are constant and accesses to the older versions get gradually slower. Another examples include the shallow binding of Baker [Bak91], which extends this scheme into a fully persistent one. This work was subsequently extended in Chuang [Chu92] and further improved in Chuang [Chu94]. A similar approach was taken by O'Neill and Burton [OB97] which supports constant lookup complexity under the condition of uniformity of access.

Chapter 7

BB-ω Trees

Balanced binary search trees are probably the most important persistent data structure. Any balanced binary tree with worst-case bounds can be made fully persistent using the path copying method of Section 2.1. The resulting persistent search tree has the same time bounds as the original, but its memory complexity is usually increased and matches its time complexity as a result of copying every node on a path to the node being modified.

There are many worst-case balanced binary search trees we can base the persistent variant on, e.g., AVL trees [AVL62], red-black trees [GS78] or B-trees [BM72], to mention the most common ones. Nevertheless, the popular choice for implementing purely functional persistent binary search trees are Adams' trees. Adams' trees, or *trees of bounded balance ω*, shortly *BB-ω* trees, were introduced in [Ada92] and [Ada93]. They are used in Haskell to implement the `Data.Map` and `Data.Set` modules, which are part of the standard data structure library CONTAINERS [PkgCont]. BB-ω trees are also used in data structure libraries in Scheme and SML.

The balance condition of BB-ω trees utilizes subtree sizes. Namely, in every node of BB-ω tree, the sizes of subtrees must differ at most by a factor of ω. This balance condition guarantees logarithmic depth, which is asymptotically optimal.

The BB-ω trees have several properties which make them a suitable choice for implementing persistent search trees:

- The elegant balance condition and simple rebalancing results in simple recursive implementation with very few special cases.

- The sizes of subtrees can be used not only during rebalancing, but also for providing additional functionality, e.g., to access the i-th smallest element in the tree in logarithmic time and to compute the size of the whole tree in constant time.

- The performance of BB-ω trees is slightly better than the performance of AVL trees [AVL62] and red-black trees [GS78], see Chapter 8 for measurements.

The correctness proof (published in [Ada92]) has serious flaws – it wrongly handles restoring balance condition after `delete`. Moreover, if the balance after `delete` is analysed correctly using only the techniques from [Ada92], the outcome is that the BB-ω trees cannot restore balance in all cases, which we show is not true.

Error in the proof manifested in several implementations, most notably in the CONTAINERS package, by violating the balance of the tree after specific `delete` operations. The problem was discovered independently by us (it was mentioned in a personal communication to Simon Peyton Jones in March 2010) and by Taylor Campbell [Cam].

This chapter contains the following:

- We describe BB-ω trees and provide a simple implementation of basic operations in the Haskell programming language [PJ$^+$03]. More complete and more efficient implementation is discussed in next Chapter 8.
- We present a correctness proof of BB-ω trees. In particular, we investigate the space of parameters of all BB-ω trees of size up to one million, and choose several candidates from this space: all parameters that are integral and also parameters guaranteeing trees with smallest depth. We then prove correctness for these candidates. Our analysis guarantees trees with lower depths than the original one and also considers previously ignored `join` operation.
- We show that the depth of BB-ω trees is better than the known upper bound.
- Because the BB-ω trees are parametrized, we perform several benchmarks to find the best choice of parameters.
- In order to save memory, we evaluate the technique of introducing additional data constructor representing a tree of size one. This allows us to save 20-30% of memory and even decreases the time complexity.

7.1 BB-ω Trees

We expect the reader to be familiar with binary search trees, see [Knu98] for a comprehensive introduction.

Definition 7.1 (Original)**.** A binary search tree is a *tree of bounded balance* ω, denoted as *BB-*ω *tree*, if in each node the following balance condition holds:

$$\begin{aligned} \textit{size of the left subtree} &\le \omega \cdot \textit{size of the right subtree}\,, \\ \textit{size of the right subtree} &\le \omega \cdot \textit{size of the left subtree}\,, \\ \textit{if one subtree is empty, the size of the other one is at most 1}\,. \end{aligned} \tag{7.1}$$

Consider a BB-ω tree of size n. According to the balance condition, the size of its left subtree is at most ω times the size of its right subtree, therefore, the size of the left subtree is at most $\frac{\omega}{\omega+1}n$. It follows that the size of a BB-ω tree decreases by at least a factor of $\frac{\omega}{\omega+1}$ at each level, which implies that the maximum depth of a BB-ω tree with n nodes is bounded by $\log_{(\omega+1)/\omega} n = \frac{1}{\log_2(1+1/\omega)} \log_2 n$. Detailed analysis is carried out in Section 7.4.

The exception for empty subtrees in the definition of balance condition is not elegant, but from the implementator's point of view it is of no concern – empty subtrees are usually represented by a special data constructor and are treated differently anyway. Nevertheless, some modifications to the balance condition have been proposed to get rid of the special case – most notably to use the size of a subtree increased by one, which was proposed in [NR72]. We therefore define and use a generalized version of the balance condition, which comprises both mentioned cases.

Definition 7.2 (Generalized)**.** A binary search tree is a *tree of bounded balance* ω, denoted as *BB-*ω *tree*, if in each node the following balance condition holds:

$$\begin{aligned} \textit{size of the left subtree} &\le \max(1, \omega \cdot \textit{size of the right subtree} + \delta)\,, \\ \textit{size of the right subtree} &\le \max(1, \omega \cdot \textit{size of the left subtree} + \delta)\,. \end{aligned} \tag{7.2}$$

The parameter δ is a nonnegative integer and if it is positive, the special case for empty subtrees is no longer necessary. Notice that the definition with sizes increased by one is equivalent to the generalized balance condition with $\delta = \omega - 1$.

An implementation of a BB-ω tree needs to store the size of a subtree of every node, which results in the following data-type:

```
data BBTree a = Nil          -- empty tree
             | Node          -- tree node
                 (BBTree a)   -- left subtree
                 Int          -- size of this tree
                 a            -- element stored in the node
                 (BBTree a)   -- right subtree
```

We also provide a function `size` and a smart constructor `node`, which constructs a tree using a left subtree, a key, and a right subtree. The balance condition is not checked, so it is upon the caller to ensure its validity.

```
size :: BBTree a -> Int
size Nil = 0
size (Node _ s _ _) = s

node :: BBTree a -> a -> BBTree a -> BBTree a
node left key right = Node left (size left + 1 + size right) key right
```

7.1.1 BB-ω Tree Operations

Locating an element in a BB-ω tree works as in any binary search tree:

```
lookup :: Ord a => a -> BBTree a -> Maybe a
lookup k Nil = Nothing
lookup k (Node left _ key right) = case k `compare` key of
                                     LT -> lookup k left
                                     EQ -> Just key
                                     GT -> lookup k right
```

When adding and removing the tree elements, we need to ensure the validity of the balance condition. We therefore introduce another smart constructor `balance` with the same functionality as `node`, which in addition ensures the balance condition. To achieve efficiency, certain conditions apply when using `balance`. We postpone further details until Section 7.2.

With such a smart constructor, the implementation of `insert` and `delete` is straightforward. Assuming the `balance` smart constructor works in constant time, `insert` and `delete` have logarithmic time complexity.

```
insert :: Ord a => a -> BBTree a -> BBTree a
insert k Nil = node Nil k Nil
insert k (Node left _ key right) = case k `compare` key of
                                     LT -> balance (insert k left) key right
                                     EQ -> node left k right
                                     GT -> balance left key (insert k right)

delete :: Ord a => a -> BBTree a -> BBTree a
delete _ Nil = Nil
delete k (Node left _ key right) = case k `compare` key of
                                     LT -> balance (delete k left) key right
                                     EQ -> glue left right
                                     GT -> balance left key (delete k right)
  where glue Nil right = right
        glue left Nil = left
        glue left right
          | size left > size right = let (key', left') = extractMax left
                                     in node left' key' right
```

```
    | otherwise             = let (key', right') = extractMin right
                              in node left key' right'

extractMin (Node Nil _ key right) = (key, right)
extractMin (Node left _ key right) = case extractMin left of
  (min, left') -> (min, balance left' key right)

extractMax (Node left _ key Nil) = (key, left)
extractMax (Node left _ key right) = case extractMax right of
  (max, right') -> (max, balance left key right')
```

When representing a set with a binary search tree, additional operations besides inserting and deleting individual elements are needed. Such an operation is join. The join operation is also a smart constructor – it constructs a tree using a key and left and right subtrees. However, it poses no assumptions on the sizes of given balanced subtrees and produces a balanced BB-ω tree. The join operation is useful when implementing union, difference and other set methods.

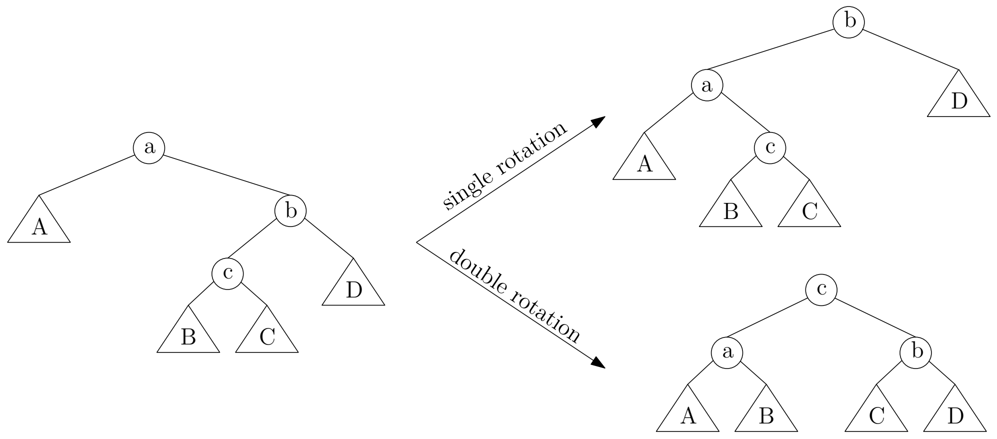
By utilizing the balance smart constructor once more, it is straightforward to implement the join operation. Again, assuming balance works in constant time, join runs in logarithmic time.

```
join :: BBTree a -> a -> BBTree a -> BBTree a
join Nil key right = insertMin key right
  where insertMin key Nil           = Node Nil 1 key Nil
        insertMin key (Node l _ k r) = balance (insertMin key l) k r

join left key Nil = insertMax key left
  where insertMax key Nil           = Node Nil 1 key Nil
        insertMax key (Node l _ k r) = balance l k (insertMax key r)

join left@(Node ll ls lk lr) key right@(Node rl rs rk rr)
  | ls > omega * rs + delta = balance ll lk (join lr key right)
  | rs > omega * ls + delta = balance (join left key rl) rk rr
  | otherwise               = node left key right
```

7.2 Rebalancing BB-ω Trees

We restore balance using well-known single rotations and double rotations. These are depicted in Figure 7.1. The code for these rotations is straightforward, the L or R suffix indicates the direction of the rotation (both rotations illustrated in Figure 7.1 are to the left).

Because we want the balance function to run in constant time, we introduce the following assumption – the balance can be used on subtrees that previously fulfilled the balance condition and since then one insert, delete or join



```
singleL l k (Node rl _ rk rr) = node (node l k rl) rk rr
singleR (Node ll _ lk lr) k r = node ll lk (node lr k r)
doubleL l k (Node (Node rll _ rlk rlr) _ rk rr) =
  node (node l k rll) rlk (node rlr rk rr)
doubleR (Node ll _ lk (Node lrl _ lrk lrr)) k r =
  node (node ll lk lrl) lrk (node lrr k r)
```

Figure 7.1: Single and double rotations

operation was performed. So far all implementations fulfilled this condition.

Using this assumption, the balance function restores balance using either a single or a double rotation – but a question is which one to choose. If we perform a left rotation as in Figure 7.1, a double rotation split the left child of the right subtree into two subtrees, but a single rotation keeps it unaffected. Therefore, we choose the type of a rotation according to the size of the left child of the right subtree.

Formally, we use a parameter α,[1] which we use as follows: When we want to perform a left rotation, we examine the right subtree. If the size of the left subtree is strictly smaller than α-times the size of the right subtree, we perform a single rotation, and otherwise a double rotation. The implementation follows:

```
balance :: BBTree a -> a -> BBTree a -> BBTree a
balance left key right
  | size left + size right <= 1 = node left key right
  | size right > omega * size left + delta = case right of
      (Node rl _ _ rr) | size rl<alpha*size rr -> singleL left key right
                       | otherwise             -> doubleL left key right
  | size left > omega * size right + delta = case left of
      (Node ll _ _ lr) | size lr<alpha*size ll -> singleR left key right
                       | otherwise             -> doubleR left key right
  | otherwise = node left key right
```

[1]Our α is the inverse of α from [Ada92].

7.3 Choosing the Parameters ω, α and δ

We call the parameters (ω, α, δ) *valid*, if `balance` can always restore the balance condition after one `insert`, `delete` or `join` operation.

Ideally we would classify all parameters (ω, α, δ) as either valid or not valid, but it is difficult to come up with complete characterization. The reason is that when dealing with small trees, rebalancing relies on the fact that all subtrees have integral sizes – i.e., it is fine that node with subtrees of sizes 1.5 and 2.5 cannot be rebalanced, because it does not exist.

Instead of a complete characterization, we therefore rule out parameters which are definitely not valid and then prove the validity only for several chosen parameters. It is easy to see that $\omega \geq 5$ and $\omega = 2$ are not valid for any α in the sense of the original balance condition, i.e., with $\delta = 0$: In the situation in Figure 7.2 neither single nor double rotation can restore balance.

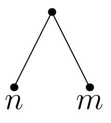
To get a more accurate idea, we evaluated validity of parameters on all trees up to size of 1 million – the results are displayed in Figure 7.3. The code used to generate this figure is listed in Attachment A.1.

When choosing the parameters, the value of ω is the most important, because it defines the height of the tree. On the other hand, the value of α is quite unimportant – it affects only the internal implementation of `balance`. The value of δ is kept as low as possible, since higher values of δ increases imbalance of BB-ω trees, especially the small ones.

After inspection of Figure 7.3 we have chosen integer parameters $(\omega = 3, \alpha = 2, \delta = 0)$ and $(\omega = 4, \alpha = 2, \delta = 0)$ and also parameters $(\omega = 2.5, \alpha = 1.5, \delta = 1)$, where the value of ω is the smallest possible. The last parameters are not integral, but we can perform multiplication by ω or α using right bit shift.

7.3.1 Validity of $\omega = 2.5$, $\omega = 3$ and $\omega = 4$

We now prove the validity of chosen parameters $(\omega = 2.5, \alpha = 1.5, \delta = 1)$, $(\omega = 3, \alpha = 2, \delta = 0)$ and $(\omega = 4, \alpha = 2, \delta = 0)$. Because the values of α and δ are determined by ω, we identify these sets of parameters only by the value of ω.



Consider performing `balance` after balance is lost. Without loss of generality, assume the right subtree is the bigger one and denote n and m the sizes of the left and right subtrees, respectively. We use the notation of the tree size and the tree itself interchangeably.

Because balance is lost, we have $\omega n + \delta < m$. The `insert` operation causes

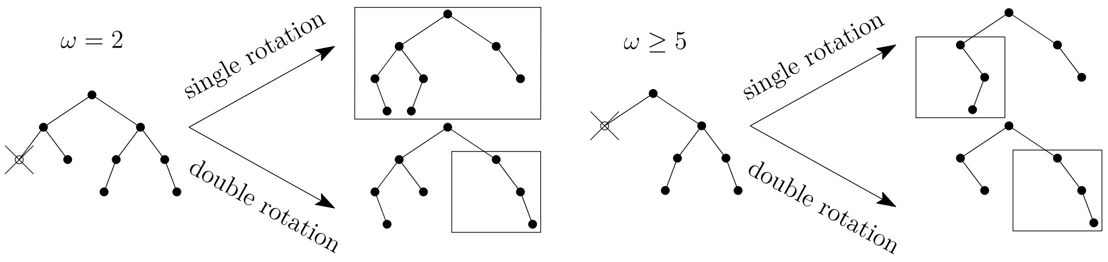


Figure 7.2: Parameters $\omega = 2$ and $\omega \geq 5$ are not valid for any α and $\delta = 0$

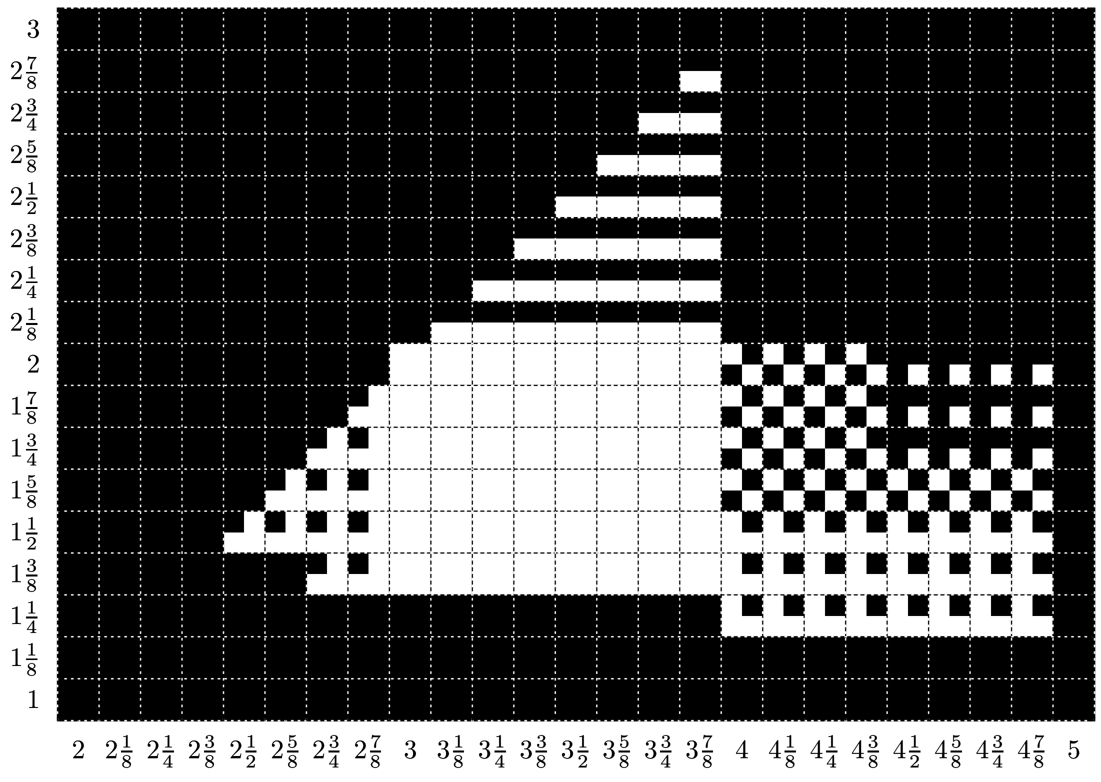


Figure 7.3: The space of (ω, α, δ) parameters. The values of ω and α are displayed on the x and y axis, respectively. Every dashed square consists of four smaller squares, which correspond to the δ values $\begin{smallmatrix}0&1\\2&3\end{smallmatrix}$. Black denotes non-valid parameters, white denotes parameters which are valid for trees of size up to 1 million. For example, when $\omega = 4$ and $\alpha = 2$, $\delta \in \{0, 3\}$ is valid and $\delta \in \{1, 2\}$ is not valid.

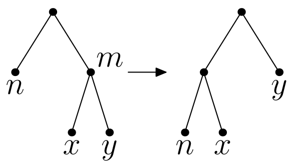
The code used to generate this figure is listed in Attachment A.1.

imbalance by exactly one element, so it is never worse than imbalance caused by a `delete` operation. Therefore, we have to consider only two possibilities how the imbalance was caused – by `delete` or `join` operation. If the last operation was `delete`, we know that $\omega n + \delta \geq m - \omega$. If the last operation was `join` with the subtree of size z, we know that $\omega n + \delta \geq m - z$. During the `join` operation the tree z was small enough to be recursively joined with subtree m, so we have $\omega z + \delta < n + 1 + (m - z)$, so $z < \frac{n+1+m-\delta}{\omega+1}$ and therefore $m - \frac{n+m+1-\delta}{\omega+1} < \omega n + \delta$, $m < \frac{\omega+1}{\omega}\left(\omega n + \delta + \frac{n+1-\delta}{\omega+1}\right)$, $m < \frac{\omega+1}{\omega}\left(\omega n + \frac{n+\omega\delta+1}{\omega+1}\right)$, $m < \left(\omega + 1 + \frac{1}{\omega}\right) n + \delta + \frac{1}{\omega}$. To summarize:

$$m \overset{(A)}{>} \omega n + \delta\,, \quad m - \omega \overset{(B_{del})}{\leq} \omega n + \delta\,, \quad m \overset{(B_{join})}{<} \left(\omega + 1 + \tfrac{1}{\omega}\right) n + \delta + \tfrac{1}{\omega}\,.$$

7.3.2 Correctness of a Single Rotation

Let x and y denote the subtrees of the tree m. We perform a single rotation iff $x < \alpha y$ and in that case we have the following inequalities:



$$\omega x + \delta \geq y \Rightarrow (\omega + 1)x + \delta \overset{(C)}{\geq} m - 1\,,$$
$$x < \alpha y \Rightarrow x \overset{(D)}{<} \tfrac{\alpha}{\alpha+1}(m - 1),\ y \overset{(E)}{>} \tfrac{1}{\alpha+1}(m - 1)\,.$$

At first we need to solve the cases where n, x or y are zero, as the balance condition is different in that case. All such cases are shown in Figure 7.4.

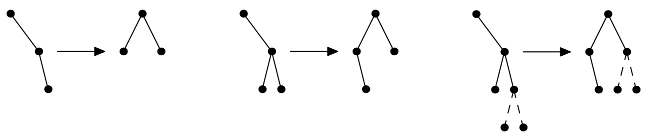


Figure 7.4: Cases when n, x or y are zero and a single rotation is performed

In the case when all subtrees are nonempty, we need to validate the balance condition in each of the two modified nodes:

- $\omega n + \delta \geq x$ after `delete`: $x \overset{(D)}{<} \frac{\alpha}{\alpha+1}(m - 1) \overset{(B_{del})}{\leq} \frac{\alpha}{\alpha+1}(\omega n + \delta + \omega - 1)$
- $\omega n + \delta \geq x$ after `join`: $x \overset{(D)}{<} \frac{\alpha}{\alpha+1}(m-1) \overset{(B_{join})}{<} \frac{\alpha}{\alpha+1}\left((\omega + 1 + \frac{1}{\omega})n + \delta + \frac{1}{\omega} - 1\right)$
- $\omega x + \delta \geq n$: $n \overset{(A)}{<} \frac{m-\delta}{\omega} \overset{(C)}{\leq} \frac{\omega+1}{\omega}x + \frac{1}{\omega}$
- $\omega(n + 1 + x) + \delta \geq y$: $y \leq \omega x + \delta$

- $\omega y+\delta \geq n+1+x$: $n+1+x = n+m-y \overset{(A)}{\leq} \frac{m-1}{\omega}+m-y = m\frac{\omega+1}{\omega}-y-\frac{1}{\omega} \overset{(E)}{<} ((\alpha+1)y+1)\frac{\omega+1}{\omega}-y-\frac{1}{\omega} = \frac{(\alpha+1)(\omega+1)-\omega}{\omega}y+1$. Here we used the fact that when ω is an integer, $m \overset{(A)}{\geq} \omega n+\delta+1$, so we have $m \overset{(A)}{\geq} \omega n+1$.

The third and the fourth inequalities obviously hold. To see that also the first, second and fifth inequalities hold, we evaluate the inequalities using chosen values of ω.

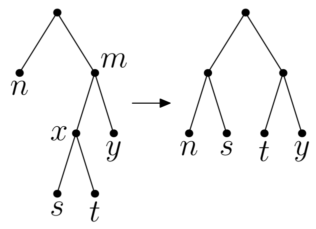
	$\omega n+\delta \geq x$ after `delete`	$\omega n+\delta \geq x$ after `join`	$\omega y+\delta \geq n+1+x$
$\omega = 2.5$	$x < \frac{3}{2}n+\frac{3}{2}$	$x < \frac{117}{50}n+\frac{6}{25}$	$n+1+x < \frac{5}{2}y+1$
$\omega = 3$	$x < 2n+\frac{4}{3}$	$x < \frac{26}{9}n-\frac{4}{9}$	$n+1+x < 3y+1$
$\omega = 4$	$x < \frac{8}{3}n+2$	$x < \frac{7}{2}n-\frac{1}{2}$	$n+1+x < \frac{11}{4}y+1$

Table 7.1: Single rotation inequalities of BB-ω trees with $w = 2.5, 3$ and 4

As you can see in Table 7.1, the linear coefficients in the inequalities are always less or equal the required ones. In some cases, the inequality itself does not hold because of a large positive additive coefficient. Nevertheless, it is simple to manually check that for such small n, no tree exists which would be unbalanced after the call to `balance`. This is caused by the fact that the counterexamples use non-integral subtree sizes.

7.3.3 Correctness of a Double Rotation

When performing a double rotation, we have the following inequalities:



$$\text{any child } a \text{ of } b \Rightarrow (\omega+1)a+\delta \overset{(C)}{\geq} b-1\,,$$
$$\text{any child } a \text{ of } b \Rightarrow (\omega+1)a \overset{(D)}{\leq} \omega(b-1)+\delta\,,$$
$$x \geq \alpha y \Rightarrow x \overset{(E)}{\geq} \frac{\alpha}{\alpha+1}(m-1),\ y \overset{(F)}{\leq} \frac{1}{\alpha+1}(m-1)\,.$$

Once again we need to solve the cases when n, y, s or t are zero – we enumerate these cases in Figure 7.5.

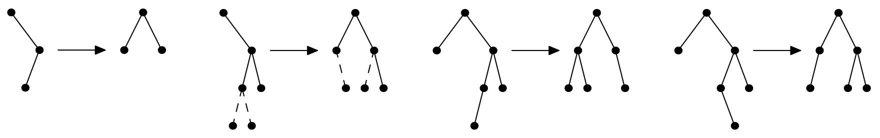


Figure 7.5: Cases when n, y, s or t are zero and a double rotation is performed

When all subtrees are nonempty, we modify three nodes, thus we have to verify six inequalities:

- $\omega n+\delta \ge s$ after `delete`: $s \overset{(D)}{\le} \frac{\omega}{\omega+1}(x-1+\frac{\delta}{\omega}) \overset{(D)}{\le} \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}(m-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega}) \overset{(B_{del})}{\le} \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}(\omega n+\delta+\omega-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega}) = \frac{\omega^3}{(\omega+1)^2}n+\frac{\omega^3+\delta\omega^2-\omega^2+\delta\omega}{(\omega+1)^2}+\frac{\delta-\omega}{\omega+1}$

- $\omega n+\delta \ge s$ after `join`: $s \overset{(D)}{\le} \frac{\omega}{\omega+1}(x-1+\frac{\delta}{\omega}) \overset{(D)}{\le} \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}(m-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega}) \overset{(B_{join})}{<} \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}((\omega+1+\frac{1}{\omega})n+\delta+\frac{1}{\omega}-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega}) = \frac{\omega^3+\omega^2+\omega}{(\omega+1)^2}n+\frac{\delta\omega^2-\omega^2+\delta\omega+\omega}{(\omega+1)^2}+\frac{\delta-\omega}{\omega+1}$

- $\omega s+\delta \ge n$: $n \overset{(A)}{<} \frac{1}{\omega}(m-\delta) \overset{(E)}{\le} \frac{1}{\omega}(\frac{\alpha+1}{\alpha}x+1-\delta) \overset{(C)}{\le} \frac{1}{\omega}(\frac{\alpha+1}{\alpha}((\omega+1)s+\delta+1)+1-\delta) = \frac{\omega+1}{\omega}\frac{\alpha+1}{\alpha}s+\frac{\delta+1}{\omega}\frac{\alpha+1}{\alpha}+\frac{1-\delta}{\omega}$

- $\omega t+\delta \ge y$: $y \le \frac{x}{\alpha} \overset{(C)}{\le} \frac{\omega+1}{\alpha}t+\frac{\delta+1}{\alpha}$

- $\omega y+\delta \ge t$: $t \overset{(D)}{\le} \frac{\omega(x-1)+\delta}{\omega+1} \le \frac{\omega(\omega y+\delta-1)+\delta}{\omega+1} = \frac{\omega^2}{\omega+1}y+\delta-\frac{\omega}{\omega+1}$

- $\omega(n+1+s)+\delta \ge t+1+y$ after `delete`: $\omega(n+1+s)+\delta \ge \omega(n+1)+t \overset{(B_{del})}{\ge} m-\delta+t \ge x-\delta+1+y+t$

- $\omega(n+1+s)+\delta \ge t+1+y$ after `join`: $t+1+y \le \omega s+\delta+1+y \overset{(F)}{\le} \omega s+\delta+1+\frac{m-1}{\alpha+1} \overset{(B_{join})}{<} \omega s+\delta+1+\frac{(\omega+1+\frac{1}{\omega})n+\delta+\frac{1}{\omega}-1}{\alpha+1} = \frac{\omega^2+\omega+1}{\omega(\alpha+1)}n+1+\frac{\omega(\delta-1)+1}{\omega(\alpha+1)}+\omega s+\delta$

- $\omega(t+1+y)+\delta \ge n+1+s$: $n+1+s \overset{(A)}{<} \frac{m}{\omega}+1+s \le \frac{m}{\omega}+1+\omega t+\delta \overset{(C)}{\le} \omega t+\delta+1+\frac{(\omega+1)y+\delta+1}{\omega} = \omega t+\frac{\omega+\delta+1}{\omega}+\frac{\omega+1}{\omega}y+\delta$

All but the first three inequalities obviously hold for positive integral sizes. In order to prove that the first three inequalities hold, we again evaluate the resulting inequalities using chosen values of ω.

	$\omega n+\delta \ge s$ after `delete`	$\omega n+\delta \ge s$ after `join`	$\omega s+\delta \ge n$
$\omega = 2.5$	$s < \frac{125}{98}n+\frac{103}{98}$	$s < \frac{195}{98}n-\frac{1}{49}$	$n < \frac{7}{3}s+\frac{4}{3}$
$\omega = 3$	$s < \frac{27}{16}n+\frac{3}{8}$	$s < \frac{39}{16}n-\frac{9}{8}$	$n < 2s+\frac{5}{6}$
$\omega = 4$	$s < \frac{64}{25}n+\frac{28}{25}$	$s < \frac{84}{25}n-\frac{32}{25}$	$n < \frac{15}{8}s+\frac{5}{8}$

Table 7.2: Double rotation inequalities of BB-ω trees with $w = 2.5, 3$ and 4

As you can see in Table 7.2, the linear coefficients in the inequalities are always less or equal the required ones. Inequalities with positive additive coefficients hold for the same reason as in the single rotation case – all such inequalities have linear coefficient strictly smaller than required and a manual check concludes that the inequalities hold even for small n using the fact that all tree sizes are integral. This concludes the proof.

7.4 BB-ω Trees Height

If the balance condition holds and $\delta \leq 1$, we know that the size of a tree decreases by at least a factor of $\frac{\omega}{\omega+1}$. Therefore, the maximum height of a tree is $\frac{1}{\log_2(1+1/\omega)} \log_2 n$. But this is merely an upper bound – it is frequently the case that the balance condition is not tight, because the tree sizes are integers.

To get an accurate estimate, we compute the maximum heights of BB-ω trees up to size of 1 million. We can use the following recursive definition:

```
-- Returns the list [ max height of BB-w tree with n elements | n <- [1..] ].
heights :: Ratio Int -> Int -> [Int]
heights w d = result
 where
   result = 1 : 2 : compute_heights 3 1 result
   compute_heights n r rhs@(rhs_head : rhs_tail)
     | w*((n-1-(r+1))%1) + d%1 >= (r+1)%1 = compute_heights n (r+1) rhs_tail
     | otherwise = 1 + rhs_head : compute_heights (n+1) r rhs
```

The function `compute_heights` is given the size of the tree n, the size of the its right subtree r and also a list of maximum heights of BB-ω trees of r and more elements. It constructs the highest tree of size n by using the largest possible right subtree, and then using the highest tree of such size.

The resulting heights are presented in Table 7.3. The heights are divided by $\lceil \log_2 n \rceil$, so the optimal height is 1. Notice that the height of a BB-2.5 tree is always smaller than 2 for less than million elements – such height is better than the height of a red-black tree of the same size.

size of BB-ω tree	height divided by $\lceil \log_2 n \rceil$		
	$\omega = 2.5$	$\omega = 3$	$\omega = 4$
10	1.33	1.33	1.33
100	1.57	1.67	1.86
1 000	1.70	1.90	2.30
10 000	1.84	2.00	2.54
100 000	1.86	2.13	2.63
1 000 000	1.90	2.16	2.70
upper bound	2.06	2.41	3.11

Table 7.3: Maximum heights of BB-ω trees with $w = 2.5$, $w = 3$ and $w = 4$

7.5 Performance of BB-ω Trees

With various possible ω to use, the question of the effect of different ω values arises. Is some value of ω universally the best one or does different usage patterns call for specific ω values?

Evidently, smaller values of ω result in lower trees. That seems advantageous, because the time complexity of many operations is proportional to the tree height.

In order to compare different values of ω, we measured the number of invocations of `balance` function. We inserted and then deleted $10^{\{1..6\}}$ elements, in both ascending and uniformly random order, and measured the number of invocations of `balance` during each phase. The results are displayed in Table 7.4.

	insert			delete		
	$w = 2.5$	$w = 3.0$	$w = 4.0$	$w = 2.5$	$w = 3.0$	$w = 4.0$
consecutive 10 elements	25	25	26	11	12	10
random 10 elements	23	23	23	12	12	12
consecutive 10^2 elements	617	657	769	362	349	302
random 10^2 elements	542	549	562	377	376	413
consecutive 10^3 elements	10245	11439	13997	6554	6116	5500
random 10^3 elements	8700	8753	8953	7162	7177	7377
consecutive 10^4 elements	143685	163261	206406	94865	88487	79938
random 10^4 elements	121192	121623	124204	105251	105854	108362
consecutive 10^5 elements	1852582	2133997	2722419	1251621	1175569	1042398
random 10^5 elements	1554230	1562168	1595269	1395871	1402939	1434371
consecutive 10^6 elements	22701321	26336469	33878677	15492747	14429384	12974950
random 10^6 elements	18956075	19074599	19476673	17367930	17480730	17856278

Table 7.4: The number of `balance` calls during inserting and deleting elements

In case of ascending elements, smaller ω values perform better during insertion, the difference between $\omega = 2.5$ and $\omega = 4$ is nearly 50% for large number of elements. On the other hand, higher ω values perform better during deletion, although the difference is only 18% at most. In case of random elements, lower values of ω are always better, but the difference is less noticeable in this case.

We also performed the benchmark of running time of `insert`, `lookup` and `delete` operations. We used the CRITERION package [PkgCrit], a commonly used Haskell benchmarking framework. All benchmarks were performed on a dedicated machine with Intel Xeon processor and 4GB RAM, using 32-bit GHC 7.0.1. Detailed description of the benchmarking process CRITERION uses is in Section 8.2.1.

The benchmarks are similar to our previous experiment – we insert, locate and delete $10^{\{1..6\}}$ elements of type `Int`, in both the ascending and uniformly random

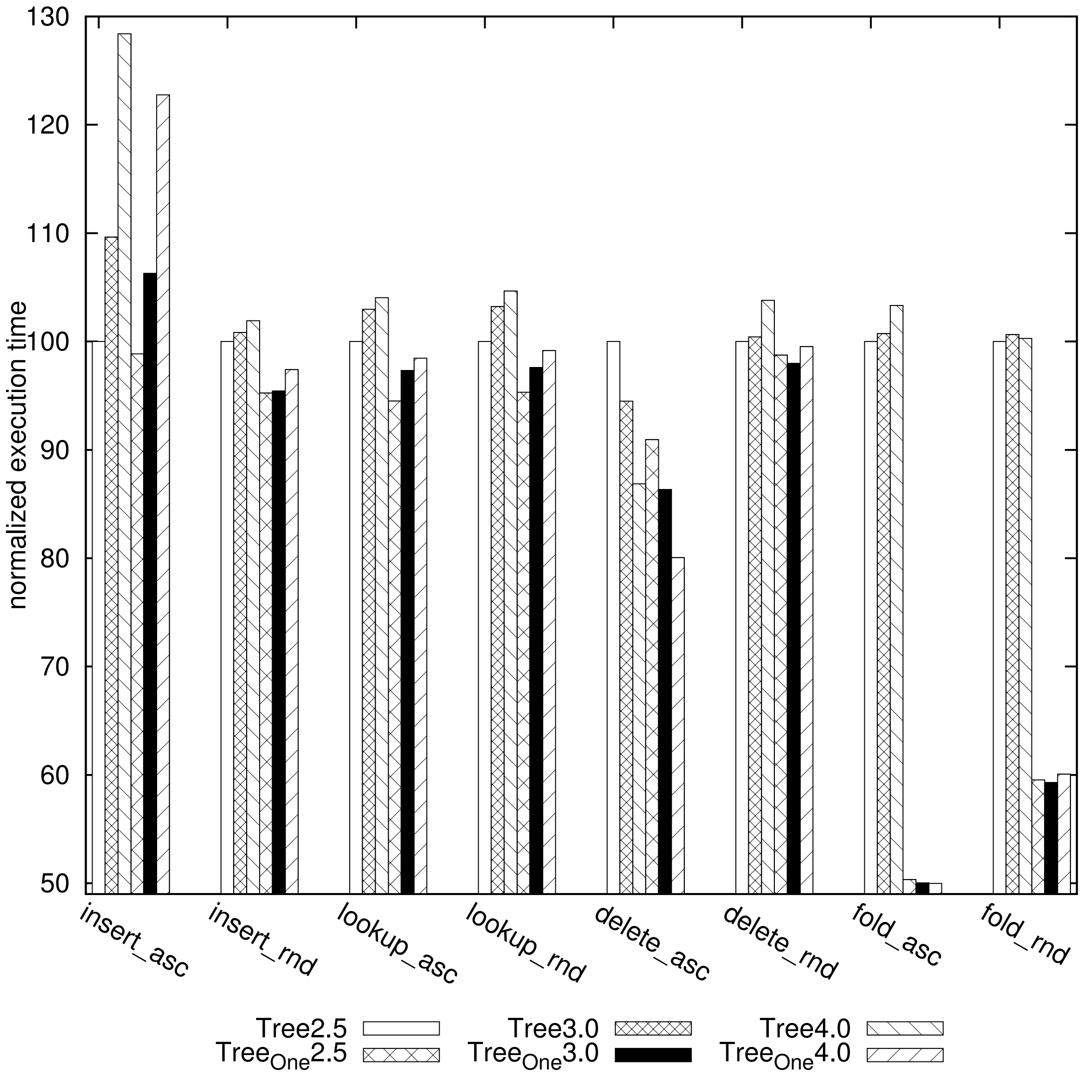


Figure 7.6: The normalized execution times of BB-ω trees with various ω

order. We used the improved implementation of `balance` from the CONTAINERS package which we describe in Chapter 8. The resulting execution times are normalised with respect to one of the implementations and presented as percentages. The overview is in Figure 7.6. (Ignore the trees with `One` subscript for now, they are explained in the next section.) Here the geometric mean of running times for all input sizes 10^1 to 10^6 is displayed. The detailed results for the individual input sizes and also the benchmark itself are attached to this thesis and also available on the author's website `http://fox.ucw.cz/papers/`.

The findings are similar to the previous experiment – if the elements are in random order, the value of ω makes little difference, smaller values performing slightly better. In case of ascending elements, smaller ω are better when inserting and larger when deleting. As expected, the `lookup` operation runs faster for smaller values of ω, independently on the order of elements.

7.6 Reducing Memory by Utilizing Additional Data Constructor

The proposed representation of a BB-ω tree provides room for improvements in terms of memory efficiency – if the tree contains n nodes, there are $n + 1$ `Nil` constructors in the whole tree, because every `Node` constructors contains two subtrees. We can improve the situation by introducing an additional data constructor representing a tree of size one:

```
data BBTree a = Nil          -- empty tree
              | One a        -- tree of size one
              | Node         -- tree node
                  (BBTree a)   -- left subtree
                  Int          -- size of this tree
                  a            -- element stored in the node
                  (BBTree a)   -- right subtree
```

Leaves are represented efficiently with this data-type. However, the trees of size 2 still require one `Nil` constructor.

To determine the benefit of the new data constructor we need to bound the number of `Nil` constructors in the tree. A `Nil` constructor appears in a tree of size 2 and if there are t trees of size 2, there need to be at least $(t-1)$ internal `Node`s for these t trees to be reachable from the root. Therefore, there can be at most $n/3$ `Nil` constructors in the tree. This implies that the number of `One` constructors is between $n/3$ and $n/2$. Experimental measurements presented in Table 7.5 show that a tree created by repeatedly inserting ascending elements contains $n/2$ `One` and no `Nil` constructors, and a tree created by inserting uniformly random elements contains approximately $0.43n$ `One` and $0.14n$ `Nil` constructors.

	$T_{One}2.5$	$T_{One}3.0$	$T_{One}4.0$
any number of consecutive elements	50.0%	50.0%	50.0%
random 10 elements	45.5%	45.5%	45.5%
random 10^2 elements	43.6%	43.6%	43.6%
random 10^3 elements	43.0%	43.0%	42.8%
random 10^4 elements	43.0%	43.0%	43.0%
random 10^5 elements	42.8%	42.8%	42.9%
random 10^6 elements	42.9%	42.9%	42.9%

Table 7.5: The percentage of `One` constructors in a BB-ω tree

Considering the memory representation used by the GHC compiler, the `Node` constructor occupies 5 words and `One` constructor occupies 2 words, so the new representation needs 20-30% less memory, depending on the number of `One` constructors. The time complexity of the new representation is also better as shown in Figure 7.6. Especially note the speedup of the `fold` operation, which is caused by reduced number of `Nil` constructors in the tree. The only disadvantage is the increase of the code size – but this affects the library author only.

We could also add a fourth data constructor to represent a tree of size 2. That would result in no `Nil` constructors in a nonempty tree. The disadvantage is further code size increase ($4 \cdot 4 = 16$ cases in the `join` operation) and also a noticeable time penalty – on 32bit machines the GHC uses pointer tagging to distinguish data constructors without the pointer dereference, which is described in detail in [MYPJ07]. This technique works with types with at most three data constructors (and up to 7 different constructors on 64bit machines), so it is not advantageous on 32bit architectures to add a fourth data constructor.

7.6.1 The Order of Data Constructors

When implementing the data-type with the `One` constructor, we found out that the order of data constructors in the definition of the data-type notably affects the performance. On Table 7.6, the time improvements in the benchmark from the previous section are displayed, when we reordered the constructors to the following order: `Node` first, then `One` and `Nil` last.

	$T_{One}2.5$	$T_{One}3.0$	$T_{One}4.0$
insert_asc	5.1%	6.8%	6.6%
insert_rnd	4.5%	5.2%	5.0%
lookup_asc	7.4%	6.1%	6.2%
lookup_rnd	6.1%	5.4%	5.4%
delete_asc	5.3%	8.4%	8.5%
delete_rnd	4.4%	4.8%	5.0%
fold_asc	8.9%	9.5%	13.1%
fold_rnd	10.1%	10.5%	9.4%

Table 7.6: The improvements of time complexity after reordering the data constructors

We believe the reason for the performance improvement is the following: When matching data constructors, a conditional forward jump is made if the constructor is not the first one from the data-type definition. Then another conditional forward jump is made if the constructor is not the second one from the data-type definition. In other words, it takes $i - 1$ conditional forward jumps to match the i-th constructor from the data-type definition, and these forward jumps are usually mispredicted (forward jumps are expected not to be taken). It is therefore most efficient to list the data constructor in decreasing order of their frequency.

7.7 Chapter Notes

This chapter contains original work and is based on the author's paper [Str12].

Several applications to the Haskell CONTAINERS package [PkgCont] originate from this chapter:

- In accordance with the analysis of valid parameters, the implementation of the `Data.Map` and `Data.Set` modules was modified to use the parameters $(\omega = 3, \alpha = 2, \delta = 0)$.
- The performance of `Data.IntMap` and `Data.IntSet` structures was improved by 10-15% by reordering the data constructors according to Section 7.6.1.
- The representation of `Data.Map` and `Data.Set` is being changed according to Section 7.6.

7.7.1 Related Work

The original weight balanced trees were described in [NR72], with two parameters with values $1+\sqrt{2}$ and $\sqrt{2}$. Because these are not integers, the resulting algorithm is not very practical. Adams created a variant of balanced trees, the BB-ω trees, described in [Ada92] and [Ada93]. Unfortunately, the proof is erroneous – the paper concludes that for $\alpha = 2$ the valid parameters are $\omega \geq 4.646$, even though the valid parameters must fulfil $3 \leq \omega < 4.5$.

The error in the proof manifested in the implementations of BB-ω trees (most notably the `Data.Set` and `Data.Map` in the CONTAINERS package), which was discovered independently by several people. The recent paper [HY11] deals with the correctness of the original weight balanced trees (equivalent to setting $\delta = \omega - 1$ in our definition) and proves in the Coq proof assistant that for $\delta = \omega - 1$, the only integral valid parameters are $\omega = 3$ and $\alpha = 2$. According to personal communication with the authors, the proof depends on the computer proof assistant and cannot be proved without it. In comparison, the proof in this chapter is explicit and covers the both the original weighted trees and also Adams' trees. On the other hand, we prove validity only for several chosen parameters, which we selected according to experiments with balance of all trees of size up to one million.

Binary search trees with similar balance condition are introduced in [Rou01]. Instead of size, $\lfloor \log_2 \mathit{size} \rfloor$ is used, so for each node, logarithms of sizes of its children must differ by at most 1. The correctness proof for such trees is simpler than for BB-ω trees, but the implementation is more complicated because of the computation of the logarithm.

Chapter 8

The Haskell CONTAINERS Package

In almost every computer language there are libraries providing various data structures, an important tool of a programmer. Programmers benefit from well written libraries, because these libraries

- free the programmer from repeated data structure implementation and allow them to focus on the high level development,
- prevent bugs in the data structure implementation,
- can provide high performance.

For many languages, there exist standardized data structure libraries (STL for C++ [SL94], Java Collections Framework, .NET System.Collections, etc.), which provide common and effective options in many cases.

Our goal is to evaluate usability and efficiency of various persistent data structures. To that end we need to choose computer language where the persistent data structure libraries are not only available, but also widely used. We decided to use Haskell [PJ$^+$03], a purely functional language with lazy evaluation. It is one of the most used functional languages and because it is purely functional, persistent structures are commonly used. In addition, there exists a centralized repository called HackageDB where nearly all users release their Haskell packages, which makes it a good source of implementations of persistent data structures.

In Haskell, the CONTAINERS package [PkgCont] is a de facto standard data structure library, being the only data structure package coming with GHC and the Haskell Platform (the standard Haskell development environment). It is used by almost every third package on the HackageDB (1782 out of 5132, 14th May 2013), which is a public collection of packages released by Haskell community.

The CONTAINERS package contains the implementations of

- *sets* of comparable elements,
- *maps* of key and value pairs with comparable keys,

- ordered *sequences* of elements,
- *trees* and *graphs*.

All data structures in this package are fully persistent with purely functional implementation.

In this chapter we present the first comprehensive performance measurements of the widely-used CONTAINERS package, including head-to-head comparisons against half a dozen other popular container libraries (Sections 8.2 and 8.4).

We have been improving the CONTAINERS package since 2010. Therefore, we describe improvements and also changes to the GHC compiler that we made to achieve best performance possible. Our decision to improve the CONTAINERS package was motivated not only by the wide accessibility of the package, but also by our intention to replace the GHC internal data structures with the CONTAINERS package. Therefore we wanted to confirm that the performance offered by the package is the best possible, both for small and big volumes of data stored in the structure.

Last but not least, we describe a new container data structure that uses hashing to improve performance in the situation where key comparison is expensive, such as the case of strings. Hash tables are usually thought of as mutable structures, but our new data structure is fully persistent. Compared to other optimised containers, performance is improved up to three times for string elements, as we describe in Section 8.4.

8.1 The CONTAINERS Package

In this section we describe the data structures available in the CONTAINERS package. We tried to cover the basic and most frequent usage, for the eventual performance boost to be worthwhile. Focusing on basic usage is beneficial for the sake of comparison too, as the basic functionality is offered by nearly all implementations.

8.1.1 Sets and Maps

A *set* is any data structure providing operations `empty`, `member`, `insert`, `delete` and `union`, as listed in Figure 8.1. Real implementations certainly offer richer interface, but for our purposes we are interested only in these methods.

A *map* from keys to values is a set of pairs (key, value) which are compared using the key only. To prevent duplication we discuss only sets from now on, but

```
data Set e
empty   ::  Set e
member  ::  Ord e => e -> Set e -> Bool
insert  ::  Ord e => e -> Set e -> Set e
delete  ::  Ord e => e -> Set e -> Set e
union   ::  Ord e => Set e -> Set e -> Set e
```

Figure 8.1: A *set* implementation provided by the CONTAINERS package

everything applies to maps too.[1]

The set and map implementations provided by the CONTAINER package are based on BB-ω trees described in Chapter 7.

8.1.2 Intsets and Intmaps

A set of `Int`s, or a map whose key type is `Int`, is used very frequently, therefore the CONTAINERS package offers specialised implementations. By an *intset* and *intmap*[2] we denote a specialised implementation of a set of `Int`s and a map with `Int` keys, respectively. These implementations should of course be faster than a regular set or map, otherwise there would be no point in using it.

We now describe the implementation of intset provided by the CONTAINERS package, because we improve the implementation further in the chapter and we also devise its variant based on hashing in Section 8.4. This structure was first described in [OG98].

The implementation is based on the so called *big-endian Patricia tries*. The integers are stored as bit sequences in a trie [Fre60], the most significant bit first. Furthermore, the trie is compressed (such a trie is also called a Patricia trie [Mor68]) by contracting the nodes with exactly one child. Therefore, the resulting structure consists of leaves representing values and nodes with exactly two children representing branchings. Each branching is specified by a mask containing the one bit separating the subtrees, and by a prefix containing bit values common to both subtrees of the branching. An example can be found in Figure 8.2.

[1]In reality it works the other way around – a set is a special case of a map that has no associated value for a key. We could use `Map e ()`, where `()` is a unit type with only one value, instead of `Set e`. But the unit values would still occupy space, which is why a `Set e` is provided.

[2]When the GHC compiles a source file, it spends 5-15 times more performing intmap operations than map operations, depending on the code generator used. These results were measured with the GHC-head on 26th March 2010.

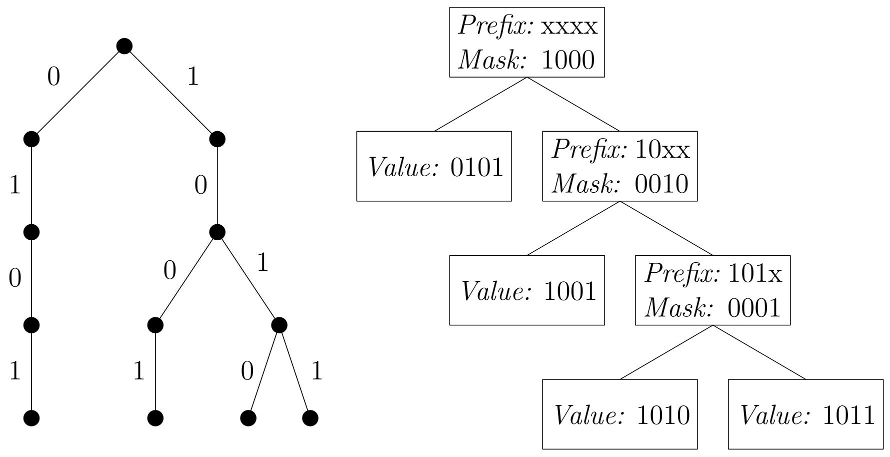


Figure 8.2: Classic trie (on the left) and compressed trie (on the right) containing numbers 5 (0101_2), 9 (1001_2), 10 (1010_2) and 11 (1011_2).

The trie structure has several advantages – there is no need to perform rebalancing and the given trie structure allows set operations like union to execute rapidly in many cases. The trie compression makes the structure memory efficient, because there are exactly $n-1$ branchings in a compressed trie containing n values. The downside is the height of the trie – the only limit on the height is 32 (or 64 on 64-bit systems). Nevertheless, when storing n consecutive elements or n random elements, the height of the trie is logarithmic (in average in the case of random elements).

8.1.3 Sequences

The CONTAINERS package also provides an implementation of a *sequence* of elements called a `Seq` with operations listed in Figure 8.3.

```
data Seq a
data ViewL a = EmptyL | a :< (Seq a)
data ViewR a = EmptyR | (Seq a) :> a
empty  :: Seq a
(<|)   :: a -> Seq a -> Seq a
(|>)   :: Seq a -> a -> Seq a
viewl  :: Seq a -> ViewL a
viewr  :: Seq a -> ViewR a
index  :: Seq a -> Int -> a
update :: Int -> a -> Seq a -> Seq a
```

Figure 8.3: Interface of a *sequence* of elements provided by the CONTAINERS package

A `Seq` is similar to a list, but elements can be added (`<|` and `|>`) and removed (`viewl` and `viewr`) to the front and also to the back of the sequence in constant time, allowing to use this structure as a *double-ended queue*. Elements can be also indexed and updated in logarithmic time and two sequences can be concatenated in logarithmic time.

8.1.4 The Rest of the CONTAINERS Package

The CONTAINERS package also contains a data type of multi-way trees. Aside from the definition of this type, only trivial methods are provided (the folds), therefore, there is no point in benchmarking those.

The last data structure offered by the package is a graph, which is built on top of the ARRAY package and offers some simple graph algorithms. We perform no graph benchmarks, as the similar FGL package is very different in design. We only describe some simple performance improvements.

8.2 Benchmarks

Our goal is to benchmark the CONTAINERS package against other popular Haskell libraries with similar functionality.

8.2.1 Benchmarking Methodology

Benchmarking a program written in a language performing lazy evaluation is a tricky business. Luckily there are powerful benchmarking frameworks available. We used the CRITERION package [PkgCrit], a commonly used Haskell benchmarking framework.

All benchmarks were performed on a dedicated machine with Intel Xeon processor and 4GB RAM, using 32-bit GHC 6.12.2. All Cabal packages were compiled using default compiler switches (except for the CONTAINERS package, where we adopted the switches of the precompiled GHC version). We tried to benchmark all available implementations on the HackageDB. The list of packages used, together with their versions, can be found in Appendix A.2.

The benchmarking process works by calling a benchmarked method on given input data and forcing the evaluation of the result. The evaluation forcing can be done conveniently using a DEEPSEQ package. However, because the representation of the data structures is usually hidden, we could not provide `NFData` instances directly and had to resort to a fold which performs an evaluation of all elements in the structure. The running time of the fold could affect the overall running time, but it is not the case in our benchmarks – we make sure that the running time of the fold is asymptotically less than the benchmark itself. We also compared our method with evaluation to weak head normal form and the difference for spine strict structures was negligible.

Because the benchmarked method can take only microseconds to execute, the benchmarking framework repeats the execution of the method until it takes reasonable time (say, 100ms) and then divides the elapsed time by the number of iterations.

This process is repeated 100 times to get the whole distribution of the time needed, and the mean and confidence interval are produced using the bootstrapping technique.

The results are displayed as graphs, one for each benchmark (Figures 8.5 to 8.18). One implementation is chosen as a baseline and the execution times are normalized with respect to the selected baseline. For each implementation and each input, the mean time of 100 iterations is displayed, together with 95% confidence interval (which is usually not visible, because it is nearly identical to the mean). For every implementation, a geometric mean of all times is displayed in the legend. The implementations except for the baseline are ordered according to this mean.

Each benchmark consists of several inputs. The size of input data is always measured in binary logarithms (e.g., the input of size 10 contains 1024 elements). This size is always the first part of description of the input, which is displayed on the x axis. The input elements are of type `Int` unless stated otherwise (`Strings` and `ByteStrings` will be used with the `HashSet` in Section 8.4). Where any order of elements in the input data could be used, we tried ascending and random order (`asc` and `rnd` in the description of the input) to fully test the data structure behaviour. The random data are uniformly distributed, generated using standard Haskell random generator with fixed seed, and duplicates are allowed.

All graphs together with the numerical data are available on the author's website `http://fox.ucw.cz/papers/`.

8.2.2 Benchmarking Sets

The `Set` interface is polymorphic in the element type, which must be an instance of `Ord`. Since the only element operation available is a comparison, nearly all implementations use some kind of a balanced search tree. We will not describe all the algorithms used, but will provide references for interested readers.

We benchmarked the following set implementations:

- `Set` and `Map` from the CONTAINERS package, which use BB-ω trees described in Chapter 7,
- `FiniteMap` from the GHC 6.12.2 sources, which also uses BB-ω trees,
- `AVL` from AvlTree package, which uses well-known AVL trees [AVL62],
- `AVL` from TreeStructures package, denoted as `AVL2` in the benchmarks, also using AVL trees,
- `RBSet` based on red-black trees [GS78], implemented by the author.

We performed these benchmarks:

- *lookup benchmark*: perform a `member` operation on every element of the given set, either in ascending order (`asc` in the input description) or in random order of elements (`rnd` in the input description). For example, the results for "`08/rnd`" are for a randomly generated input of size 2^8.
- *insert benchmark*: build a set by sequentially calling `insert`, either in ascending (`asc` in the input description) or in random (`rnd` in the input description) order of elements,
- *delete benchmark*: sequentially `delete` all elements of a given set, either in ascending (`asc` in the input description) or in random (`rnd` in the input description) order of elements,
- *union benchmark*: perform a `union` of two sets of given sizes (the sizes are the first and second part of the input description). The input description `asc` means the elements in one set are all smaller than the elements in the other set. The description `e_o` stands for an input, where one set contains the even numbers and the other the odd numbers. The last input description `mix` represents an input whose n elements are grouped in $\sqrt{n}$ continuous runs of $\sqrt{n}$ elements, and there runs are split between the two sets.
- *tree union benchmark*: given a tree with elements in the leaves, perform `union` in all internal vertices to compute the `union` of all the elements. The *tree union* benchmark models a particularly common case in which a set is generated by walking over a tree – for example, computing the free variables of a term. In this situation, many `union` calls operate with very small sets, which is a very different usage pattern compared to the *union benchmark*.

  The input description `asc` and `rnd` specify the order of the elements in the leaves. The shape of the tree is specified by the last letter of the input description. The letter b stands for perfectly balanced binary tree, u denotes unbalanced binary tree (one subtree is six times the size of the other subtree) and p stands for a centipede, see Figure 8.4.

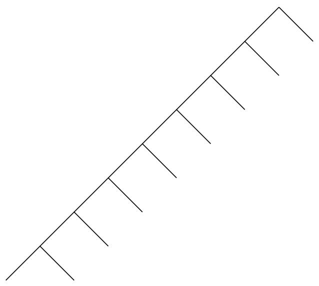
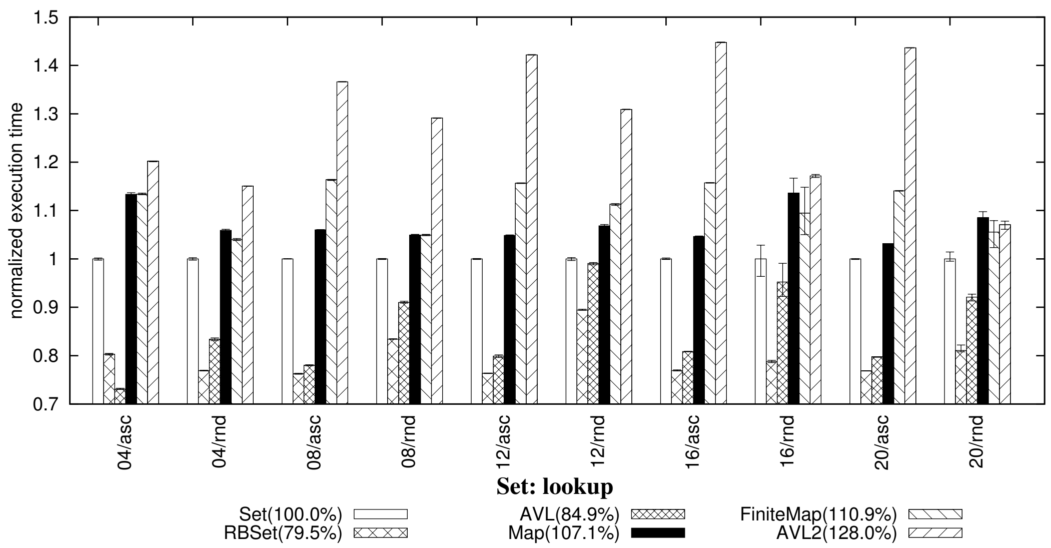


Figure 8.4: A highly unbalanced tree called the *centipede*



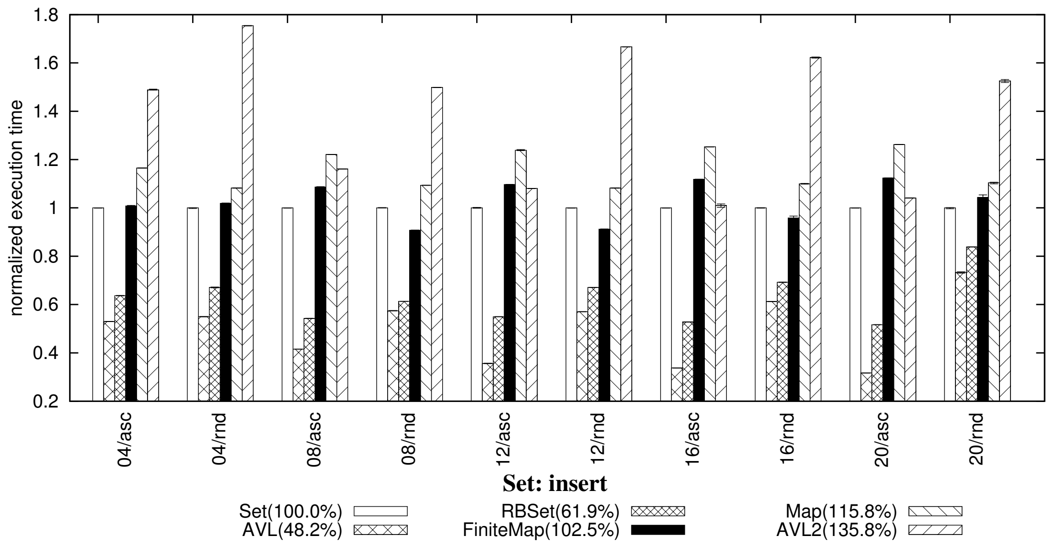


Figure 8.5: Benchmark of sets operations I

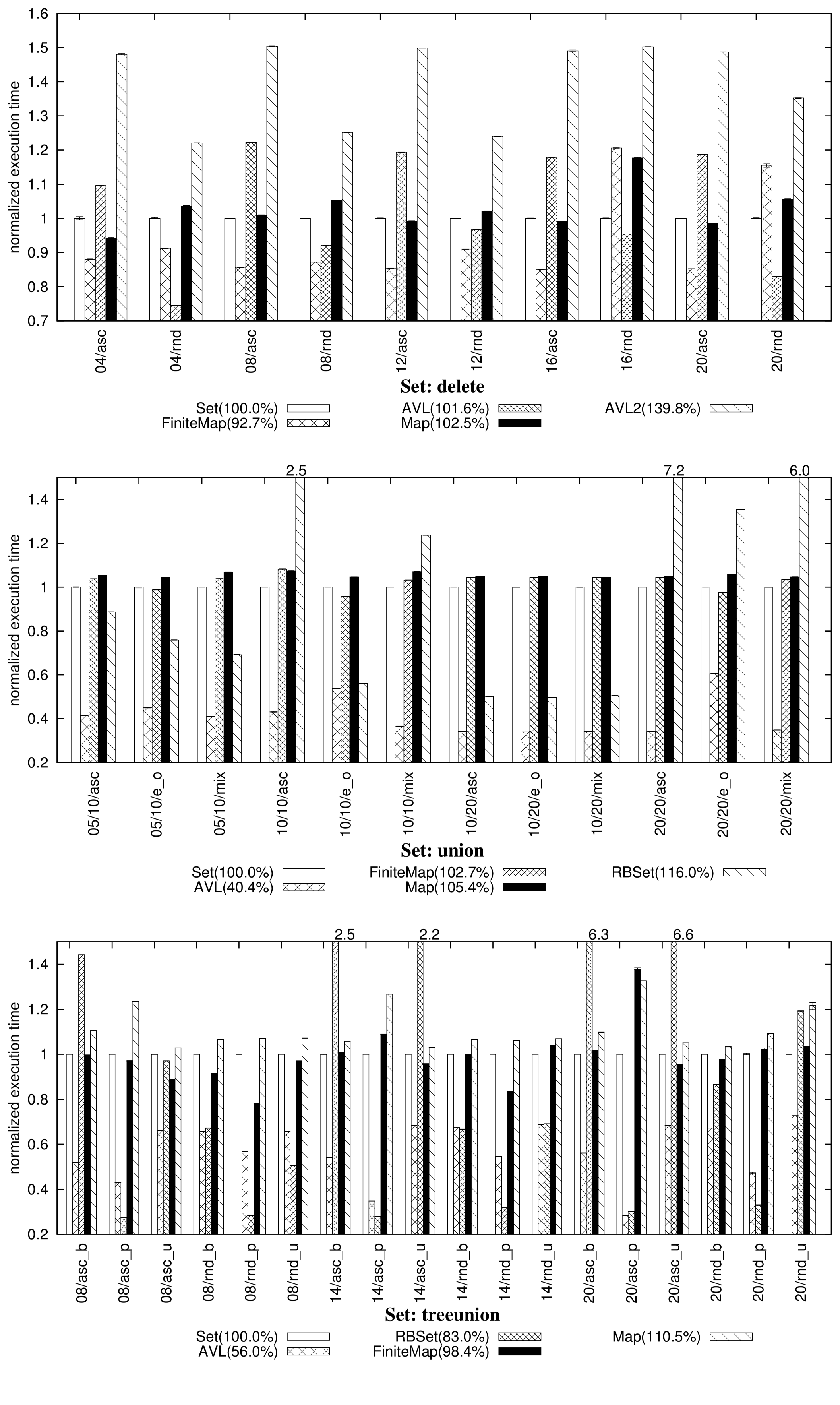


Figure 8.6: Benchmark of sets operations II

The results of the benchmarks are presented in Figures 8.5 and 8.6. The performance of the `Set` is comparable to the `FiniteMap`, but it is significantly worse than `AVL` and `RBSet`. The improvements to the `Set` implementation bringing it on par with `AVL` and `RBSet` are described in Section 8.3.

8.2.3 Benchmarking Intsets

The purpose of an intset implementations is to outperform a set of `Int`s. This can be achieved by performing additional operations with `Int`s in addition to a comparison. All mentioned implementations exploit the fact that an `Int` is a sequence of 32 or 64 bits.

We have benchmarked following intset implementations:

- `IntSet` from the CONTAINERS package which implements big-endian Patricia trees described in Section 8.1.2,
- `UniqueFM` from GHC 6.12.2 sources which also implements big-endian Patricia trees,
- `PatriciaLoMap` from `EdisonCore` package, called `EdisonMap` in the benchmark, which implements little-endian Patricia trees [OG98].

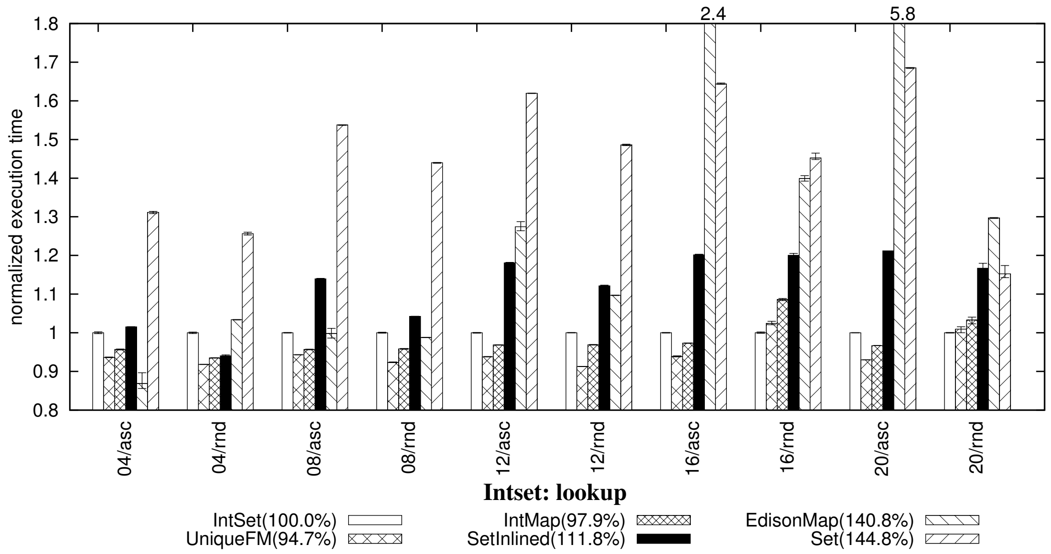
We also include ordinary `Set Int` from the CONTAINERS package in the benchmarks. For comparison, we also manually specialised the `Set` implementation by replacing overloaded comparisons with direct calls to `Int` comparisons, a process that could be mechanised.[3] By comparing to this implementation, called `SetInlined`, we can see the effect of the *algorithmic* improvements (rather than mere specialisation) of other intset implementations.

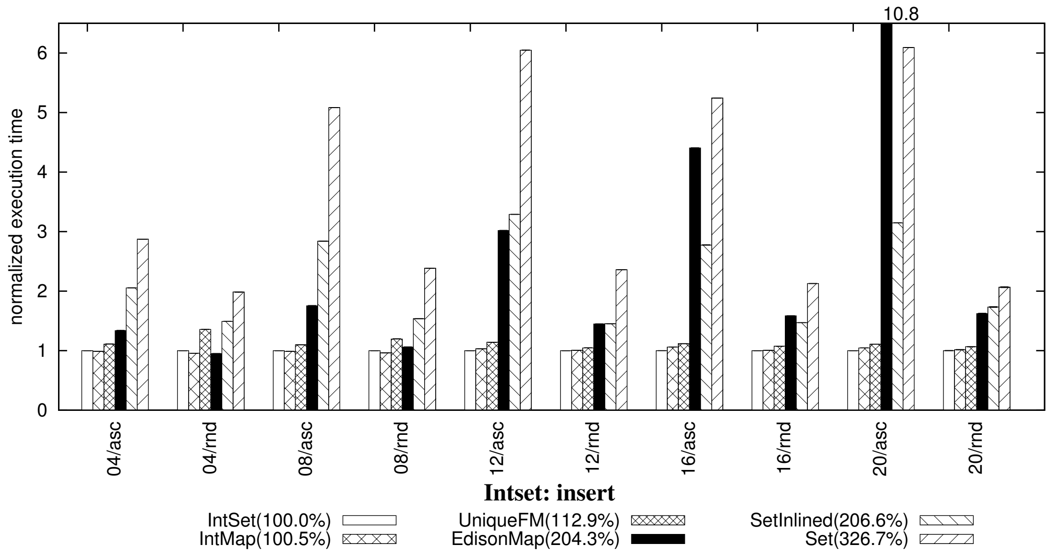
The benchmarks performed are the same as in the case of generic set implementations. The results can be found in Figures 8.7 and 8.8.

The `IntSet` outperforms all the presented implementations, except for the lookup and delete benchmark, where the `UniqueFM` is slightly faster. The `IntSet` is considerably faster than a `Set Int`, especially in the tree union benchmark, where it runs more than four times faster.

Although the performance of `IntSet` is quite good, we describe several improvements in Section 8.3.

[3]After discussions with GHC developers, this process is now automatically performed in the so called specialise pass, provided that the unfolding of a function is known. That can be assured by using the `INLINABLE` pragma added for this purpose in GHC 7.0.1.





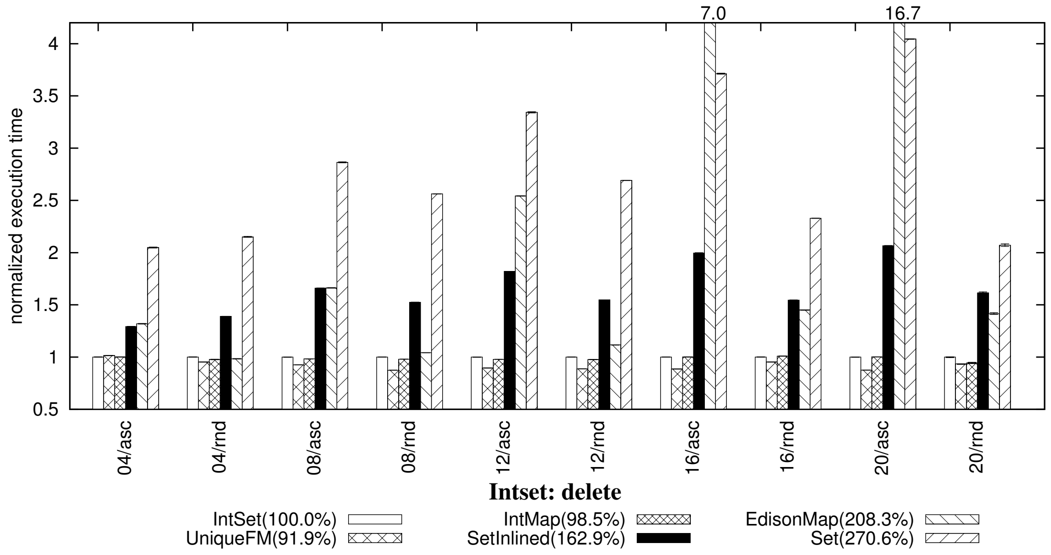
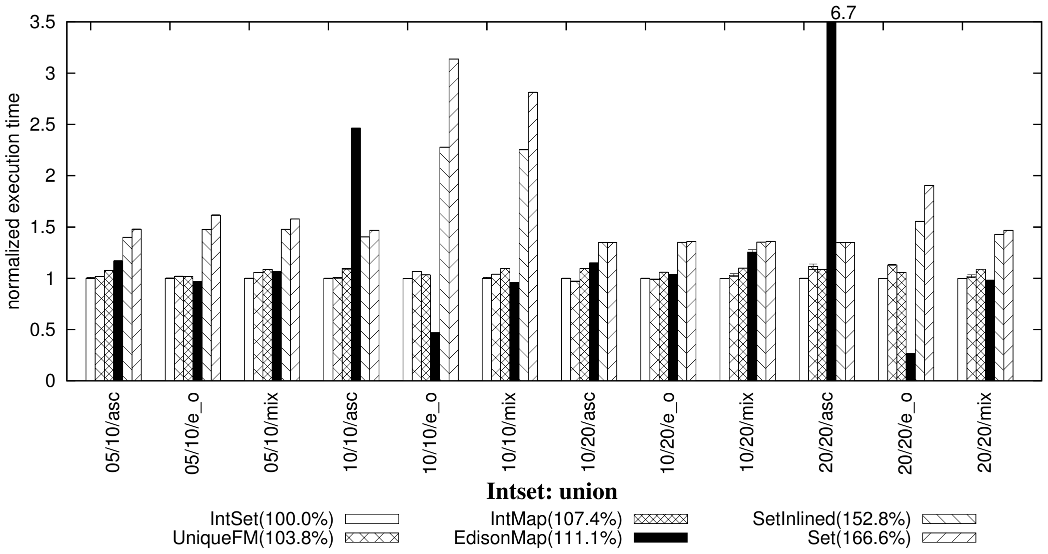


Figure 8.7: Benchmark of intsets operations I



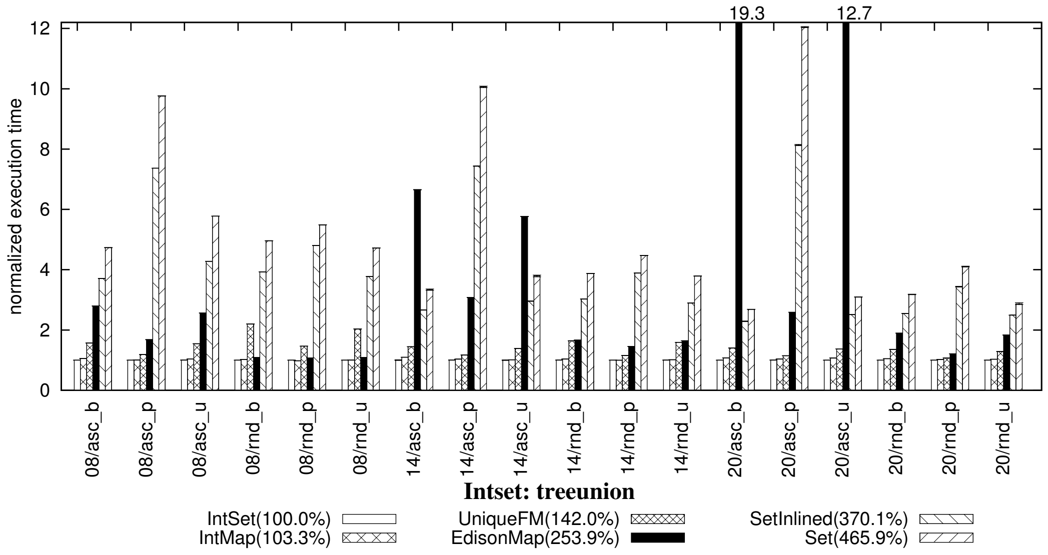


Figure 8.8: Benchmark of intsets operations II

8.2.4 Benchmarking Sequences

The Seq type in CONTAINERS supports beside others both (a) deque functionality (add and remove elements at beginning and end), and (b) persistent-array functionality (indexing and update). We compare the Seq to several other libraries, most of which support only (a) or (b) but not both, and which are therefore expected to outperform the Seq.

Queue Functionality

The queue functionality performance is significant, because there are no other implementations of queues and deques in standard Haskell libraries and so the Seq is the "only" choice when a queue is needed.

The *queue* benchmark consists of two phases: first a certain number of elements is added to the queue (the number of the elements added is the first part of the input description) and then some of the previously added elements are removed from the queue (the second part of the input description). We also tried mixing the insertions and removals, but the differences in performance were negligible, therefore, we do not present these experiments.

In the queue benchmark we tested the following implementations:

- Seq from the CONTAINERS package, which implements 2-3 finger trees annotated with sizes [HP06],
- Trivial, which is a non-persistent queue with amortized bounds, described in Section 5.2 of [Oka99],
- Amortized, which is a persistent queue with amortized bounds, described in Section 6.3.2 of [Oka99],
- Realtime, which is a persistent queue with worst-case bounds, described in Section 7.2 of [Oka99],
- Ed_Simple, Ed_Amortized and Ed_Seq from the EdisonCore package, which implement the same algorithms as Trivial, Amortized and Seq, respectively.

The results are displayed in Figure 8.9. The Ed_Seq is missing, because it was roughly 20 times slower than the Seq implementation. Because the Trivial queue implementation is not persistent, we do not consider it to be a practical alternative. That means the Seq implementation is only 50% slower than the fastest queue implementation available. That is quite a good result, considering the additional functionality it provides.

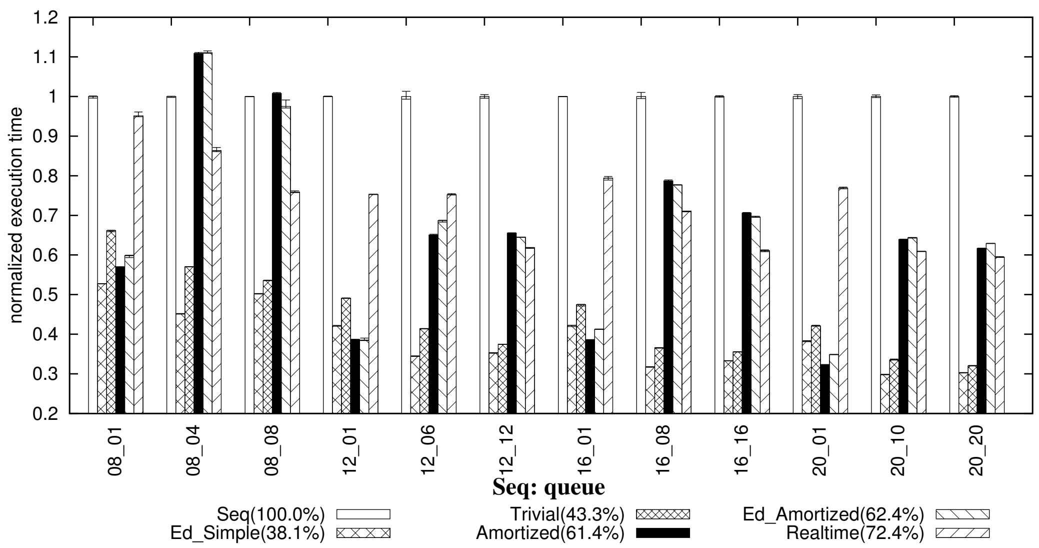


Figure 8.9: Benchmark of queue operations

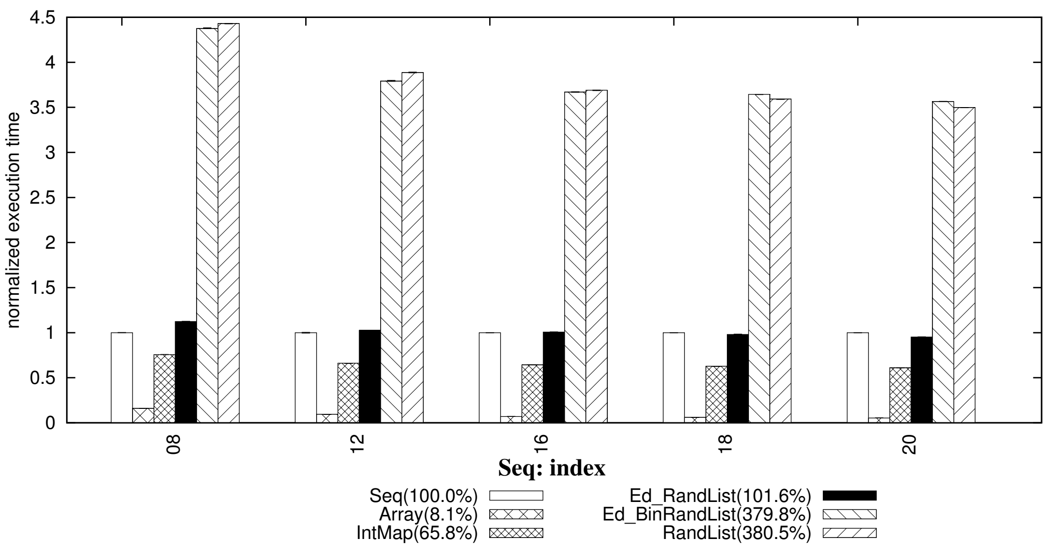
Persistent Array Functionality

The *index* and *update* benchmark perform a sequence of `index` and `update` operations, respectively, one for each element in the structure (the size of this structure is in the input description). The benchmark is nearly identical to the persistent array benchmark of Section 6.2.

We benchmarked the following implementations:

- `Seq` from the CONTAINERS package, which implements 2-3 finger trees annotated with sizes [HP06],
- `Array` from the ARRAY package, a monolithic array [Wad87] providing constant time indexing,
- `RandList` from the RANDOM-ACCESS-LIST package, which implements the skew binary random-access list from Section 9.3 of [Oka99],
- `Ed_RandList` from the EDISONCORE package, which implements the same algorithm,
- `Ed_BinRandList` from the EDISONCORE package, which implements bootstrapped binary random-access list from Section 10.1.2 of [Oka99],
- `Ed_Seq` from the EDISONCORE package,
- `IntMap` from the CONTAINERS package.

The results are presented in Figure 8.10. Again, we do not display `Ed_Seq`, because it was 10-20 times slower than `Seq`. The `IntMap` was used as a map



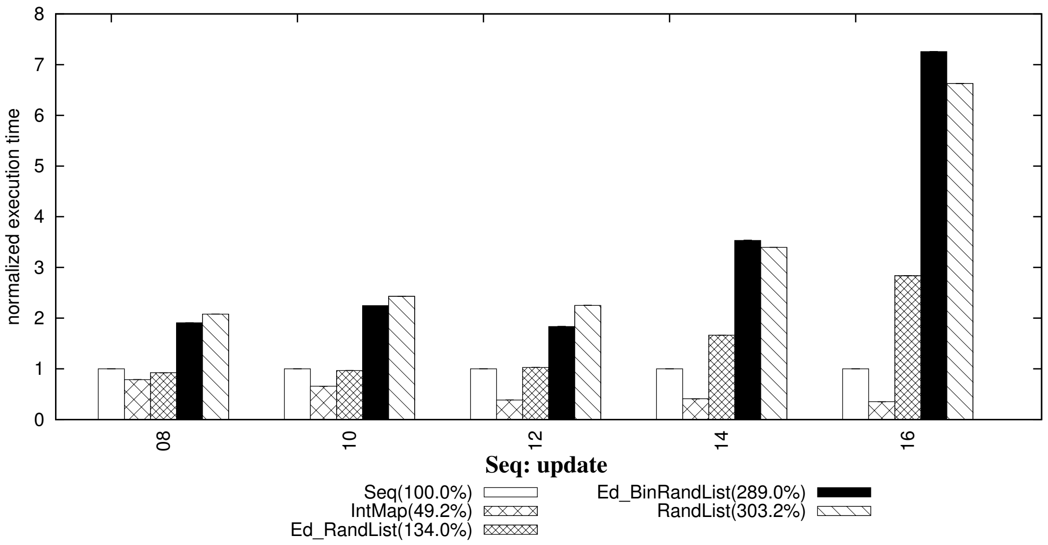


Figure 8.10: Benchmark of sequence operations

from the `Int` indexes to the desired values. Despite the surplus indexes, it outperformed most of the other implementations. The `Array` is present only in the lookup benchmark, because the whole array has to be copied when modified and thus the `update` operation modifying only one element is very ineffective.

Summary

The `Seq` type is neither fastest queue nor the fastest persistent array, but it excels when both these qualities are required.[4] For comparison, when an `IntMap` is used in the queue benchmark, it is 2.5 times slower than `Seq`, and `Ed_RandList` and `Ed_BinRandList` are 5 times and 7 times slower, respectively.

[4]In addition, a `Seq` can also be split and concatenated in logarithmic time.

8.3 Improving the CONTAINERS Performance

We have been improving the performance of the CONTAINERS package since 2010 and the author has been the maintainer of this package since 2011. In this section we summarize our most important improvements. Many of them are general techniques that can be used to improve other data structure implementations.

There are several methods of improving the existing code. The "simplest" method is to improve the (asymptotic) complexity of the algorithm. We say simplest, because such improvements can be done by examining the source code only without requiring benchmarks or binaries, although the necessary theoretical background can be quite complicated.

As an example, consider the following definitions from the `Data.Tree` module:

```haskell
data Tree a = Node a (Forest a)
type Forest a = [Tree a]
```

In the `Data.Graph` module,[5] function for pre-order and post-order `Tree` traversal are provided. The reader is encouraged to find out what is wrong with *both* of these implementations:

```haskell
preorder            :: Tree a -> [a]
preorder (Node a ts) = a : preorderF ts
preorderF           :: Forest a -> [a]
preorderF ts         = concat (map preorder ts)

postorder            :: Tree a -> [a]
postorder (Node a ts) = postorderF ts ++ [a]
postorderF           :: Forest a -> [a]
postorderF ts         = concat (map postorder ts)
```

The `postorder` case is straightforward – the list concatenation is linear in the length of the first list, so the time complexity of `postorder` performed on a path is quadratic.

The `preorder` is a bit more challenging – `concat` has complexity linear in the lengths of all but the last list given. This also results in quadratic behaviour, for example when the `preorder` is executed on a centipede (Figure 8.4). This mistake is also present in the `postorder` function.

It is trivial to reimplement both these functions to have linear time complexity.

However, performance improvements do not originate from algorithmic improvements only. More frequently, the same algorithm can be reimplemented to run faster. Profiling is usually used to find out which parts of the implementation

[5] Up to version 0.4; the functions were corrected in the latter versions.

take long to execute and would be beneficial to improve.

Having two implementations, we can also examine why one is faster. In the simplest case we can do so at the level of Haskell sources. But if the reason for different performance is not apparent, we can inspect the differences at the level of Core Haskell [Tol01] using for example the `-ddump-stranal` GHC flag, which shows the results of strictness analysis. If this is not enough, we can examine the `C--` code [PJRR99] using the `-ddump-cmm` GHC flag. We had to resort to analysis on all these levels when improving the performance of the CONTAINERS.

We briefly describe our most important changes to the CONTAINERS package. All of these changes have already been incorporated into the package and their correctness has been verified using the test suite of the CONTAINERS package. We also present the benchmark results of the new implementations.

8.3.1 Improving Set and Map

According to the benchmarks, we decided to continue using BB-ω tree representation and focus only on its improvements. All of the described improvements apply to both to `Set` and `Map`.

- As already mentioned, a `Set` can contain keys of any comparable type (i.e., any instance of `Ord`). This behaviour is achieved by parametrising all `Set` methods performing comparisons with a comparison function. This overhead can be substantial, most notably when a `Set` method spends a lot of time comparing the elements, like `member` or `insert`.

  One solution to this problem is to mark all such methods `INLINE`. That allows such methods to be inlined to every call site and if the call is not polymorphic, to specialise the method bodies to use a specific comparison method instead of a generic one. The downside of this solution is code growth – we have to repeat the method body in every call site. Nevertheless, this solution is used for GHC before version 7.0. The code grown issue can be alleviated by making sure only the tree navigation is inlined, not rebalancing and other tasks not requiring element comparison.

  Since GHC 7.0, GHC developers together with us came up with a better alternative that allows sharing of the specialised method bodies. A new specialisation pass was added to the optimizer. If a polymorphic method is called with a known type (a particular class instance, to be exact) and if its unfolding is known, it is specialised in this type and placed to the call site module. The specialisation is remembered and reused in this module or in any module depending on this one. To ensure that an unfolding of a method is available, new pragma `INLINABLE` was provided. This solution is currently used in CONTAINERS when compiled with GHC 7.0 or later.

However, we always inline some higher-order methods, notably the folds. They have very small bodies and spend most of the time calling the user supplied function, therefore, inlining improves performance considerably and the code growth is minimal.

- Rebalancing was originally performed by a following balance method:

```
balance :: Set a -> a -> Set a -> Set a
balance left key right
  | size left + size right <= 1 = node left key right

  | size right > omega * size left =
      if size (left_of right) < alpha * size (right_of right)
         then singleL left key right
         else doubleL left key right

  | size left > omega * size right = rotateR left key right
      if size (right_of left) < alpha * size (left_of left)
         then singleR left key right
         else doubleR left key right

  | otherwise = node left key right
```

If the balance condition is broken, a method performing one of the four rotations is called, rebuilding the affected nodes using smart constructors. This results in a repeated pattern matching, which is unnecessary. We rewrote the `balance` function to contain all the logic and to pattern match every node at most once. That resulted in a significant performance improvement in all `Set` methods modifying a given set.

- When a recursive method accesses its parameter at different recursion levels, Haskell usually has to check that it is evaluated each time it is accessed. For a `member` or `insert` this effect causes measurable slowdown. We rewrote such methods so that they evaluate their parameters at most once. For illustration, we changed the original implementation of the `member` method:

```
member :: Ord a => a -> Set a -> Bool
member x Tip = False
member x (Bin _ y l r) = case compare x y of
                           LT -> member x l
                           GT -> member x r
                           EQ -> True
```

to the following one.

```
member _ Tip = False
member x t = x `seq` member' t where
  member' Tip = False
  member' (Bin _ y l r) =
    case compare x y of LT -> member' l
                        GT -> member' r
                        EQ -> True
```

Nevertheless, it is important to check that no heap memory is allocated for the `member'` closure. That would happen if weak head normal form of the result contained a suspended call to `member'`. In that case, allocating the additional closure would probably be more expensive.

Still, creating the `member'` closure is quite fragile and can cause increased heap allocation under specific circumstances. Consider for example the case when `member` is inlined and `member'` is not. Then the evaluated `x` must be somehow passed to `member'`. But if `x` is present only in a register, it needs to be heap-allocated before passed to `member'`. According to our measurements of the GHC performance, this happens rarely and the resulting increased heap allocation is negligible. However, a more predictable optimization should be devised in the long term.

- We improved the set operations, e.g., `union`, to handle more small cases. Originally, the recursion stopped when one tree was empty. We now also handle the case when one of the trees is a singleton. There are many possible small cases which could be handled, so we chose the best ones using a benchmark.[6]

- In all set operations, a comparison with a possibly infinite element must be performed. That was originally done by supplying a comparison function, which was constant for the infinite bound. Supplying a value `Maybe elem` with infinity represented as `Nothing` instead of a comparison function improved the performance.[7] To demonstrate the source code changes, consider the `filterGt` method, which creates a new set from the given set consisting of the elements greater than the given bound, which can be negative infinity.

[6]Only the small cases of size one were helpful, special cases for sets of 2 and 3 elements did not further improve the performance. Also, the small cases did not help in all set operations.

[7]We in fact use strict version of the datatype: `data MaybeS a = NothingS |Just !a`.

```
filterGt :: (a -> Ordering) -> Set a -> Set a
filterGt _   Tip           = Tip
filterGt cmp (Bin _ x l r) = case cmp x of
  LT -> join x (filterGt cmp l) r
  GT -> filterGt cmp r
  EQ -> r
```

We altered the implementation as follows.

```
filterGt Nothing t = t
filterGt (Just b) t = b `seq` filter' t where
  filter' Tip = Tip
  filter' (Bin _ x l r) = case compare b x of
    LT -> join x (filter' l) r
    GT -> filter' r
    EQ -> r
```

- The `fromList` method creates a set of a list of elements. If the elements are already ordered, asymptotically more efficient `fromAscList` method can be used. Nevertheless, the library users must explicitly use the latter method, which is inconvenient – they have to know they have to use it and sometimes they even cannot, if the code in question is present in a code of some other programmer.

  Instead, we modified `fromList` to dynamically choose how to build the set. We start by assuming the input list is ordered and build the set using a faster algorithm. If we detect an unordered element, we keep the already constructed set and add the rest of the input list using a slower algorithm. This dynamic behaviour is much better for the library users, although it is quite complicated to create a implementation which is not slower than any of the original method.

- According to the experiments with reordering data constructors in Section 7.6.1 we reordered the constructors of `Set` from

  ```
  data Set a = Tip | Bin !Size !a !(Set a) !(Set a)
  ```

  to the following.

  ```
  data Set a = Bin !Size !a !(Set a) !(Set a) | Tip
  ```

  However, the performance difference of reordering only two data constructors is minor.

The results are displayed in Figures 8.11 and 8.12. The improved implementations are called `NewSet` and `NewMap` and outperform all other implementations.

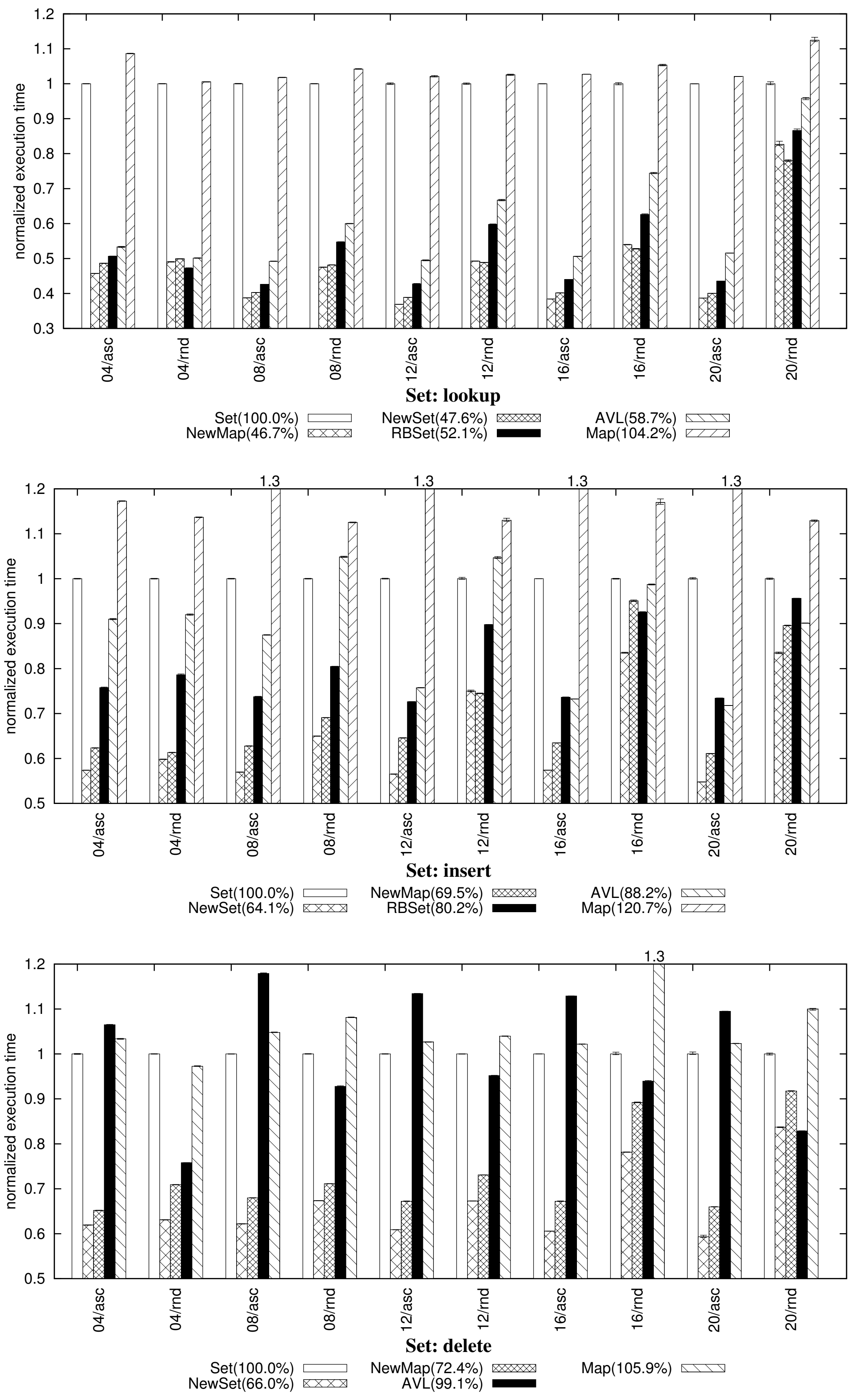


Figure 8.11: Benchmark of improved sets operations I

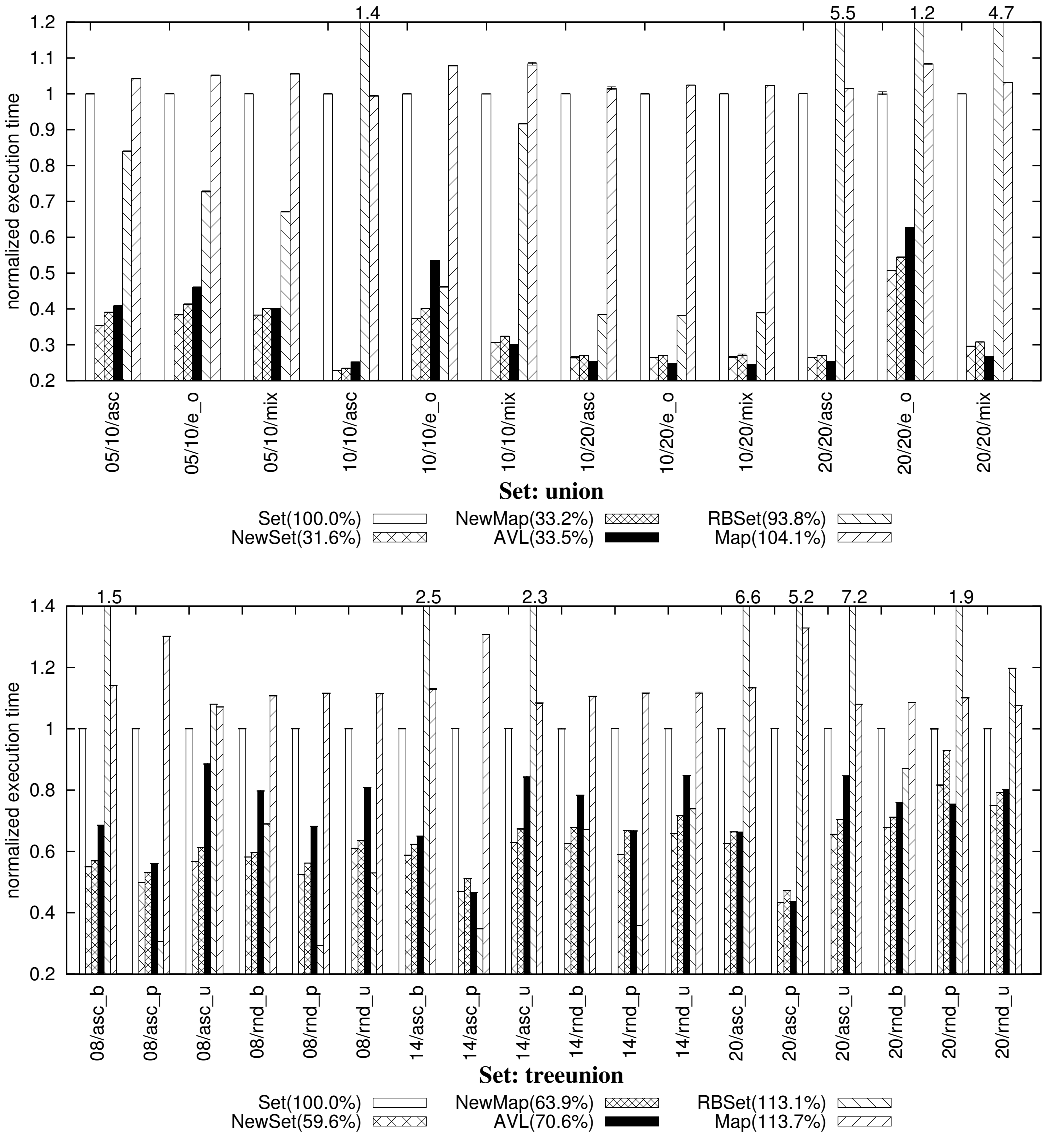


Figure 8.12: Benchmark of improved sets operations II

8.3.2 Improving IntSet and IntMap

The `IntSet` implementation was already performing quite well and was more difficult to improve then the `Set`. The following improvements apply to both `IntSet` and `IntMap`, only the last one applies to `IntSet` only.

- As with `Set`, some recursive functions repeatedly checked whether the parameters are already evaluated. We made sure this check is done at most once. Because some functions were already strict in the key, it was enough to add the `seq` calls to appropriate places. This improved the `lookup` complexity significantly.
- Contrary to `Set`, functions of `IntSet` are not polymorphic and need no specialization. Therefore, we only inline small higher-order functions, notably the folds, as previously.
- According to the experiments with reordering the data constructors in Section 7.6.1 we reordered the constructors of an `IntSet` from

  ```
  data IntSet = Nil | Tip !Int | Bin !Prefix !Mask !IntSet !IntSet
  ```

  to the following

  ```
  data IntSet = Bin !Prefix !Mask !IntSet !IntSet | Tip !Int | Nil
  ```

  The constructors are now introduced from the most frequent one to the least frequent one. This change resulted in approximately 10% performance improvement of all methods.
- We were able to improve the set operations, e.g., `union`, by reordering the pattern matches in the method definition. Consider the original definition:

  ```
  union :: IntSet -> IntSet -> IntSet
  union (Bin p1 m1 l1 r1) (Bin p2 m2 l2 r2) = ...
  union (Tip x) t = insert x t
  union t (Tip x) = insertR x t --right biased insert
  union Nil t     = t
  union t Nil     = t
  ```

  Because the patterns are matched from top to bottom, this order of the patterns is not efficient. Consider `union Nil (Tip _)`. According to the above definition, it must be handled by the third case, not the fourth one. So a `union Nil` must perform a needless pattern match on its second argument to see whether it is a `Tip` or not.

A better ordering is the following:

```
union (Bin p1 m1 l1 r1) (Bin p2 m2 l2 r2) = ...
union t@(Bin _ _ _ _) (Tip x) = insertR x t --right biased insert
union t@(Bin _ _ _ _) Nil = t
union (Tip x) t = insert x t
union Nil t = t
```

This way the first argument is examined and only if it is `Bin`, also the second argument is examined.[8]

- In contrast to `Set` and `Map`, which have nearly the same representation, we changed the representation of the `IntSet`. The idea was suggested by Joachim Breitner, who also created the initial implementation.

  The new representation of `IntSet` is *dense*. Assume `Int` has 32 bits, other sizes are analogous. If we want to represent a set of numbers in range 0..31, we can do it using an `Int`, where i-th bit is zero or one according to whether the number i is in the set. We can therefore modify the `IntSet` to store only the first $32 - 5 = 27$ bits in the big-endian Patricia trie (a so called *prefix*) and for each prefix remember a *bitmap* of elements of the set with this prefix. The resulting representation is therefore:

```
data IntSet = Bin !Prefix !Mask !IntSet !IntSet
            | Tip !Prefix !BitMap
            | Nil
```

  This representation is most useful when the set is dense and the bitmaps contain many ones. But even if the set is not dense and every bitmap contain exactly one bit set, the memory overhead is only 14% considering GHC memory representation.

  It is important that the dense representation does not cause (nearly) any overhead in the case the represented set is sparse. That was the case with nearly all methods. The only problematic operation was enumeration of set elements stored in a bitmap, which is needed during a fold.[9] This enumeration must be fast and linear in the number of values stored in the bitmap.

  We were able to implement the enumeration efficiently using the De Bruijn sequences [dB46]. Firstly, we find the lowest bit set in the bitmap *bm* using the following sequence of operations:

[8]The general method describing optimal order of pattern matches can be found in Chapter 5 of [PJ87].

[9]And also during `filter` and `partition` methods.

`bm`	`101010...10100...0`
`negate bm`	`010101...01100...0`
`x = bm .&. negate bm`	`000000...00100...0`

Secondly, we find the index of the only bit set in x, let it be i. We use the fact that multiplying by x is exactly the same as bit shifting to the left by i. Therefore, if we look at the highest 5 bits of $x \cdot c$, it is a unique quintuple of bits from c for every i. Now the De Bruijn sequences come to the rescue – if every quintuple of bits in c is unique, we can decode i from such a quintuple, using a simple table lookup. The resulting implementation for 32-bit arithmetics is:

```
lowestBitIndex :: Word -> Int
lowestBitIndex bm = table!(((bm.&.negate bm) * 0x077CB531) `shiftR` 27)
  where table = listArray (0,31) [0,1,28,2,29,14,24,3,30,22,20,
                                  15,25,17,4,8,31,27,13,23,21,
                                  19,16,7,26,12,18,6,11,5,10,9]
```

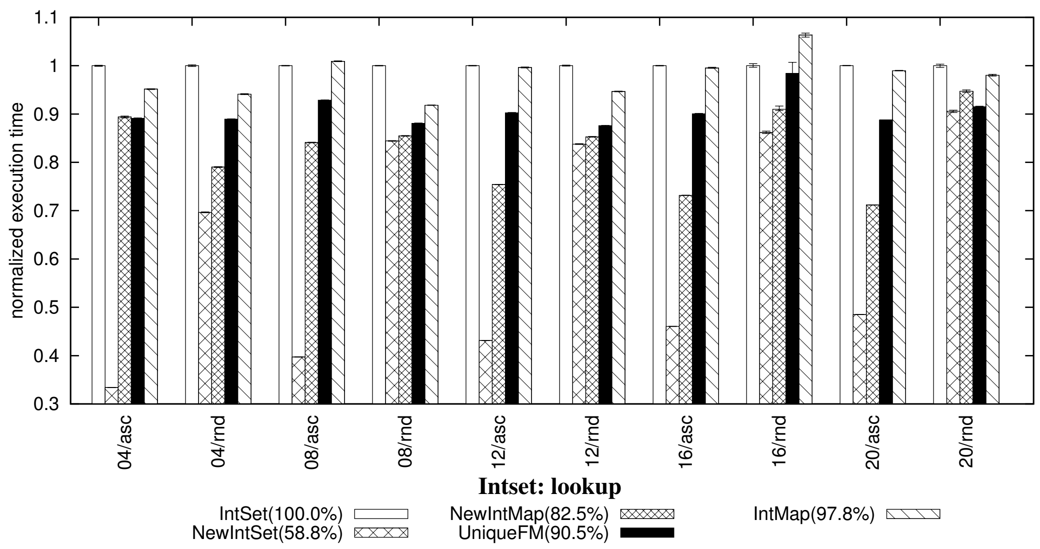
Using this method (and analogous one for 64-bit arithmetics), we can enumerate bits in an `Int` bitmap efficiently enough.

The benchmark results of the improved implementations (called `NewIntSet` and `NewIntMap`) are presented in Figures 8.13 and 8.14. The `NewIntSet` implementation is much faster on all sequential (i.e., dense) sets thanks to the dense representation. On random inputs there seem to be none or minimal overhead. The improvement of `NewIntMap` is smaller, but still significant.

8.4 New Hashing-Based Container

When a comparison of two elements is expensive, using a tree based set representation can be slow, because at least $\log_2 N$ comparisons must be performed to locate an element in the set. In this section we investigate whether we can do better by developing a new implementation for set optimised for the expensive-comparison case.

Two approaches suggest themselves. First, one could use a hash table (Section 6.4 of [Knu98]) to guess the position of an element in the set and performs only one comparison if the guess was correct. Another alternative is a trie (Section 6.3 of [Knu98]), which can also be implemented using a ternary search tree [BS98], which compares only *parts* of the keys in case the keys are sequences of elements (this is what `IntSet` and `IntMap` does).



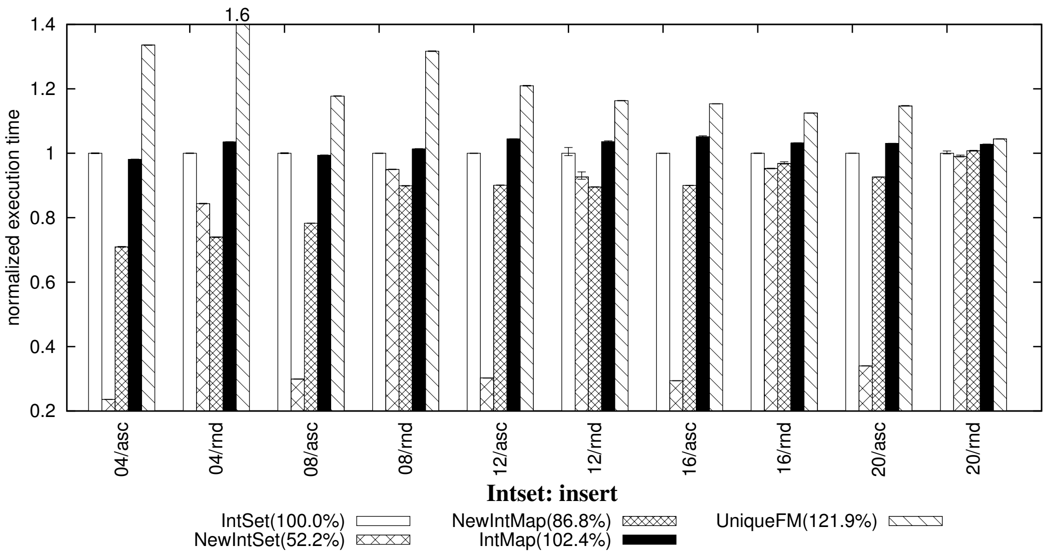
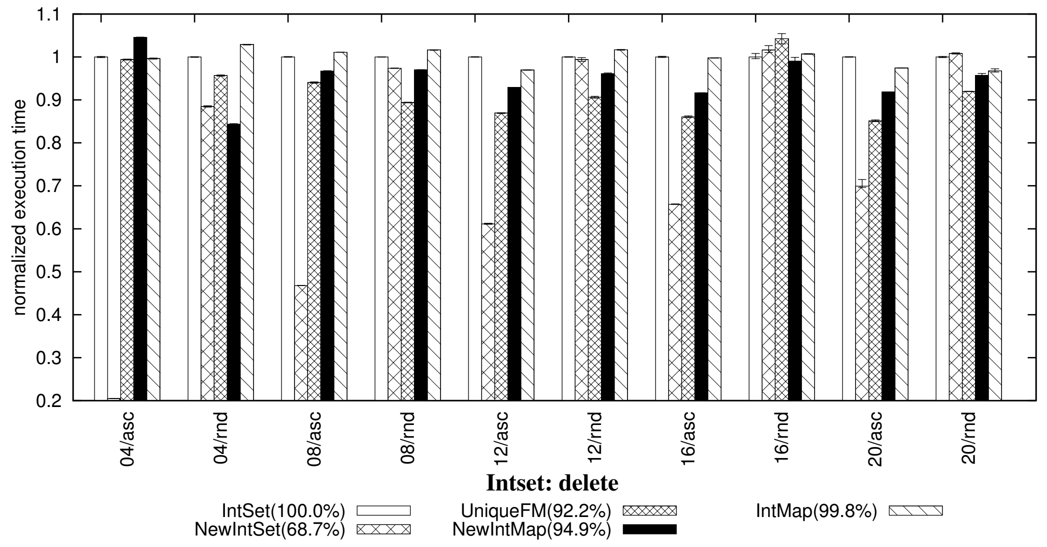
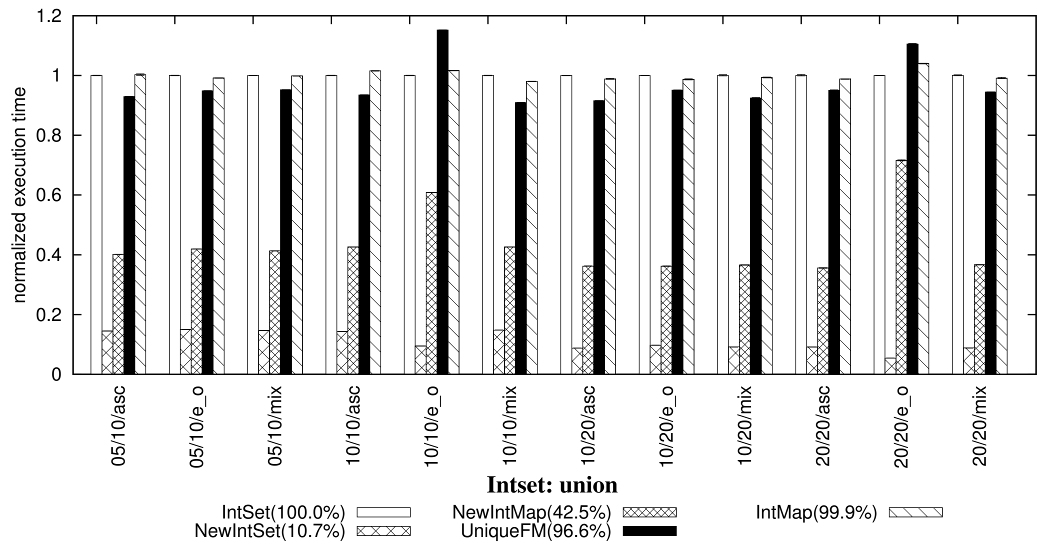


Figure 8.13: Benchmark of improved intsets operations I





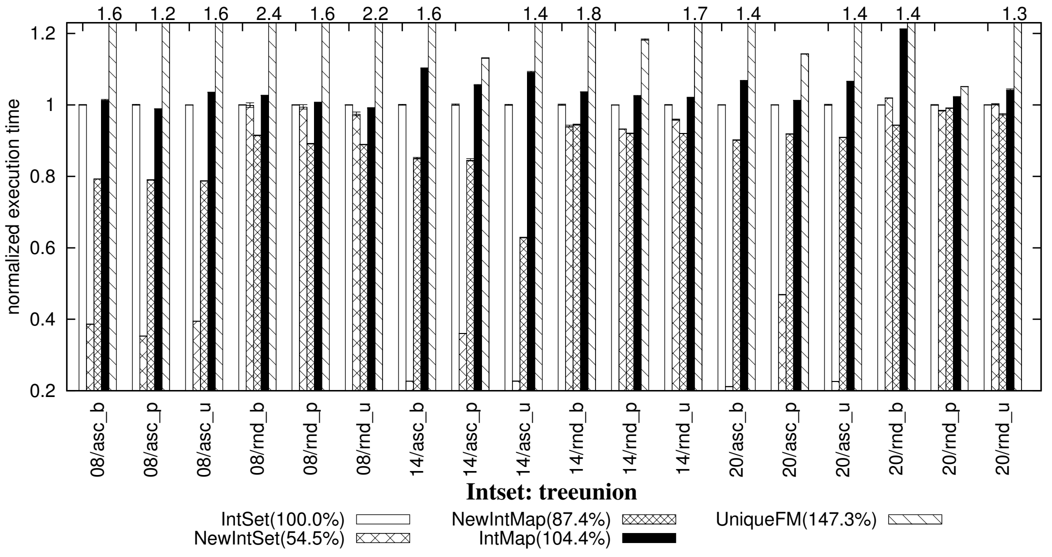


Figure 8.14: Benchmark of improved intsets operations II

In order to implement a hash table, we need an underlying persistent array implementation. Also, we need to take care of rehashing the structure to adjust the size of the persistent array to the size of the hash table. However, the rehashing is not trivial and would probably have to be performed gradually using some kind of global rebuilding [Ove83a].

There is a way to avoid the rehashing. If we had a persistent array implementation that was *sparse*, i.e., able to store only a subset of elements effectively, we could perform no rehashing at all – we could use a full (32-bit or 64-bit) hash without computing the element index modulo persistent array size. Such a persistent array implementation is in fact an `IntMap`.

An important observation is that the most efficient fully persistent array we came up in Chapter 6 is in fact a tree structure, so the overhead of an intmap (or we may call it a sparse persistent array) is justifiable – see the difference of `IntMap` and `Tree_A2` performance in Figure 6.2.

Following this idea, we can implement a `HashSet elem` as:

```haskell
data HashSet elem = HashSet (IntMap (Set elem))
```

The `HashSet` is therefore an `IntMap` indexed by the hash value of an element. In the `IntMap`, there is a `Set elem` containing elements with the same hash value (this set will be of size one if there are no hash collisions). A `HashMap` can be implemented in the same way as

```haskell
data HashMap key val = HashMap (IntMap (Map key val)).
```

Such a data structure is sometimes called a *hash trie* and has already described in [Gou94] or in [Bag01].

This data structure is quite simple to implement, using the methods of an `IntMap` and a `Set` or a `Map`. It offers the subset of `IntMap` interface which does not depend on the elements being stored in ascending order (the elements are stored in ascending order of the hash value only). Namely, we do not provide `toAscList`, `split`, and the methods working with the minimum and maximum element (`findMin`, `findMax` and others). Moreover, the folds and maps are performed in unspecified element order.

We uploaded our implementation to the HackageDB as a package called HASHMAP.

We performed the same lookup, insert and delete benchmark on the `HashSet` as on the `Set` and `IntSet`. We used the original unimproved implementation of the CONTAINERS package – the performance of the `HashSet` further increases with the improvements from Section 8.3 incorporated.

The performance of a HashSet of Ints is displayed in Figure 8.15. It is of course worse than the IntSet, because it uses an additional Set for elements with same hash values.

The HashSet should be beneficial when the comparison of the set elements is expensive. We therefore benchmarked it with Strings and ByteStrings elements. We compared the HashSet implementation to all alternatives present on the HackageDB (mostly trie-like data structures):

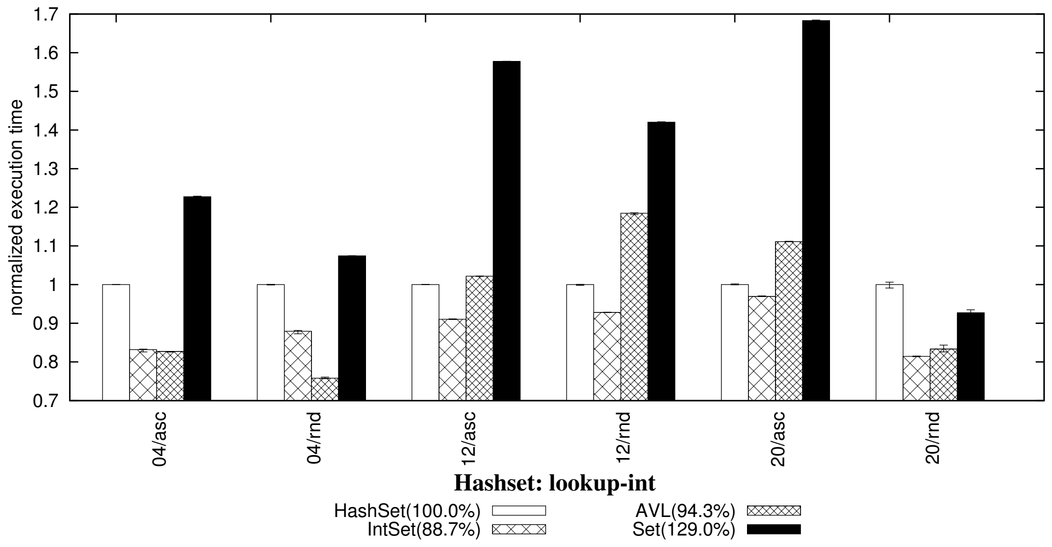
- ListTrie and PatriciaTrie from the LIST-TRIES package implementing a trie and a Patricia trie (Section 6.3 of [Knu98]),
- BStrTrie from the BYTESTRING-TRIE package, which implements a big-endian Patricia tree [OG98] specialised for ByteStrings,
- StringSet from the TERNARYTREES package, which implements a ternary search tree [BS98] specialised for the elements of type String,
- TernaryTrie from EdisonCore also implementing a ternary search tree.

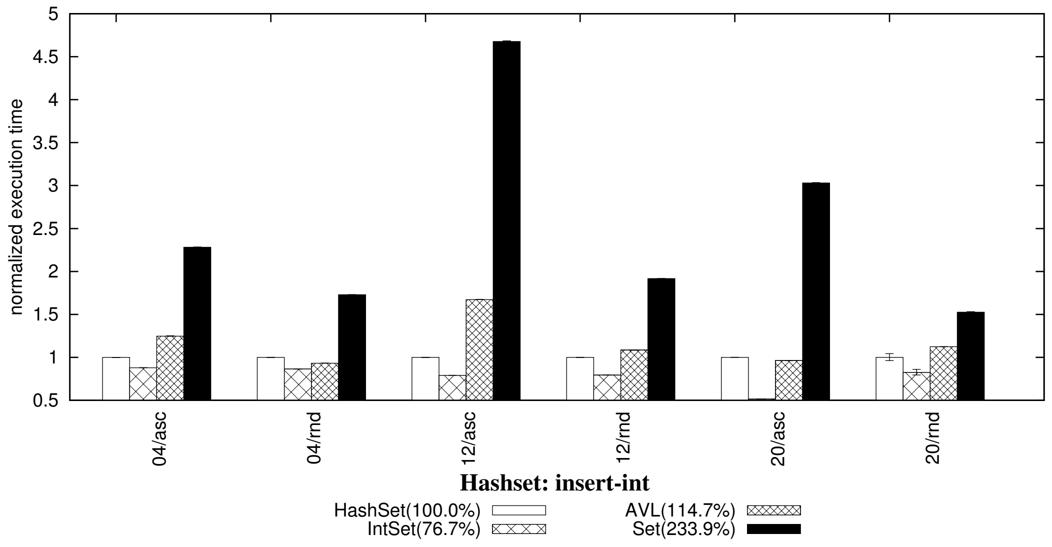
The results are presented in Figures 8.16 and 8.17. The length of the strings used in the benchmarks is the last number in the input description. We used uniformly distributed random strings of small letters (rnd in the input description) and also a consecutive ascending sequence of strings (asc in the input description). In the latter case, the strings have a long common prefix of a's. The ListTrie is not present in the benchmark results, because it was 5-10 times slower than the HashSet.

The HashSetNoC is the same as the HashSet, only the computation of a hash value of a ByteString is done in Haskell and not in C. There is quite significant slowdown in the case Haskell generating the hashing code. We discussed this with the GHC developers and were informed that the problem should be solved using the new LLVM backend [Ter09].

We also performed the union benchmark. We generated a sequence of elements (its length is the first part of the input description) and created two sets of the same size, one from the odd elements and the other from the even elements. We then performed a union of those sets. The results for Int, String and ByteString elements are presented in Figure 8.18.

The performance of a HashSet is superior to trie structures, even those specialised for String or ByteString elements. As mentioned, the performance with the current version of CONTAINERS is substantially improved (the Figures 8.13 and 8.14 show the improvements of the current CONTAINERS version).





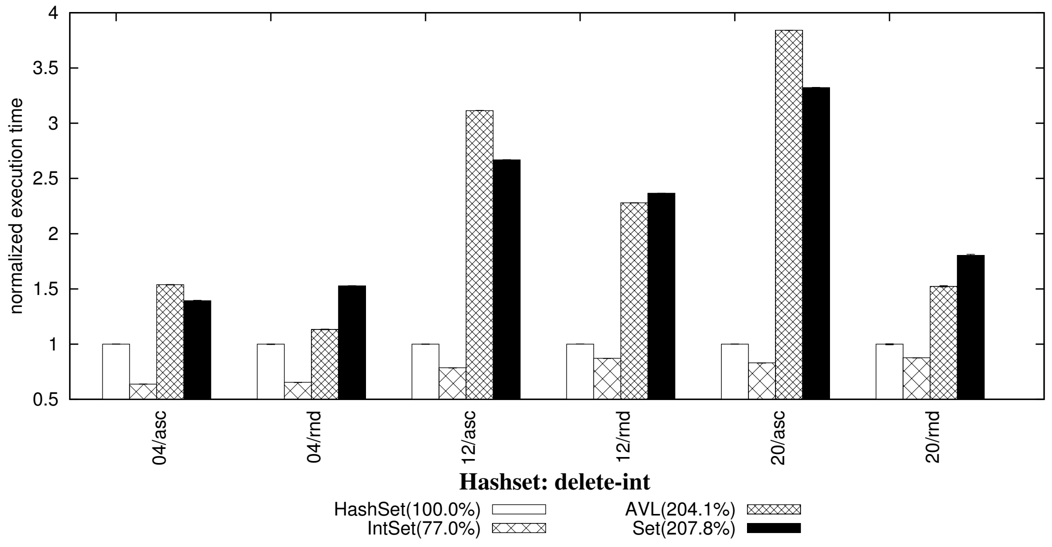
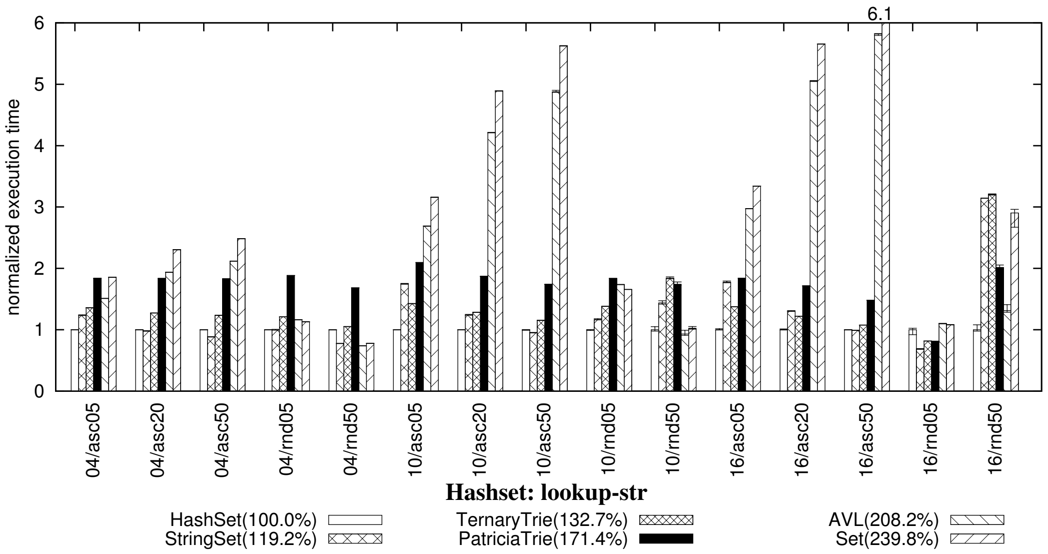
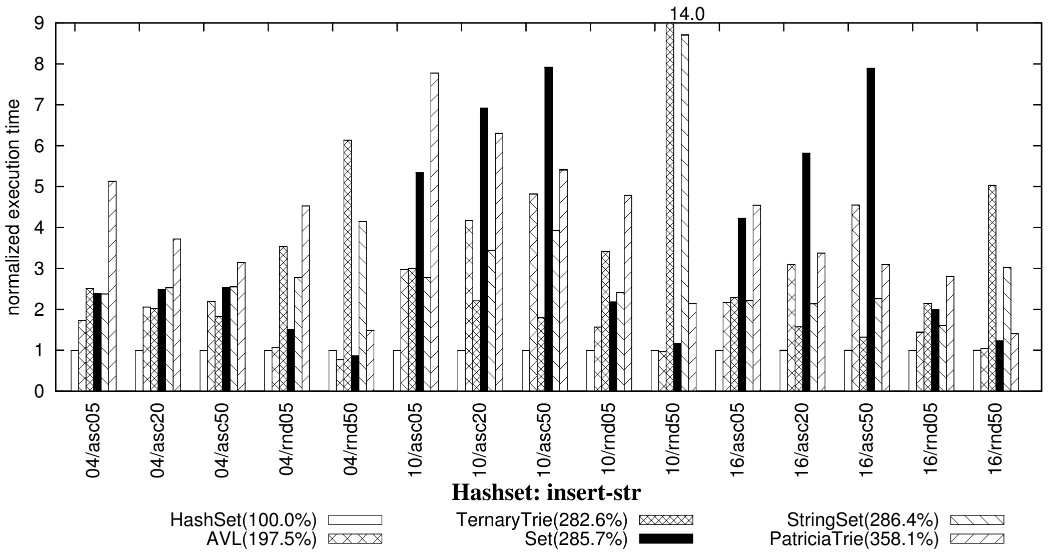


Figure 8.15: Benchmark of hashset operations on `Int`s





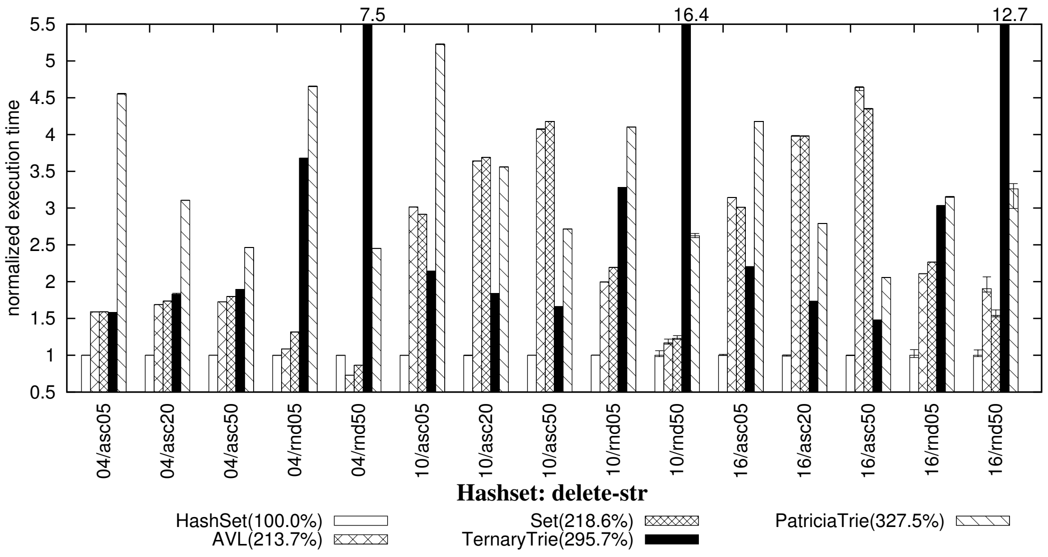


Figure 8.16: Benchmark of hashset operations on `Strings`

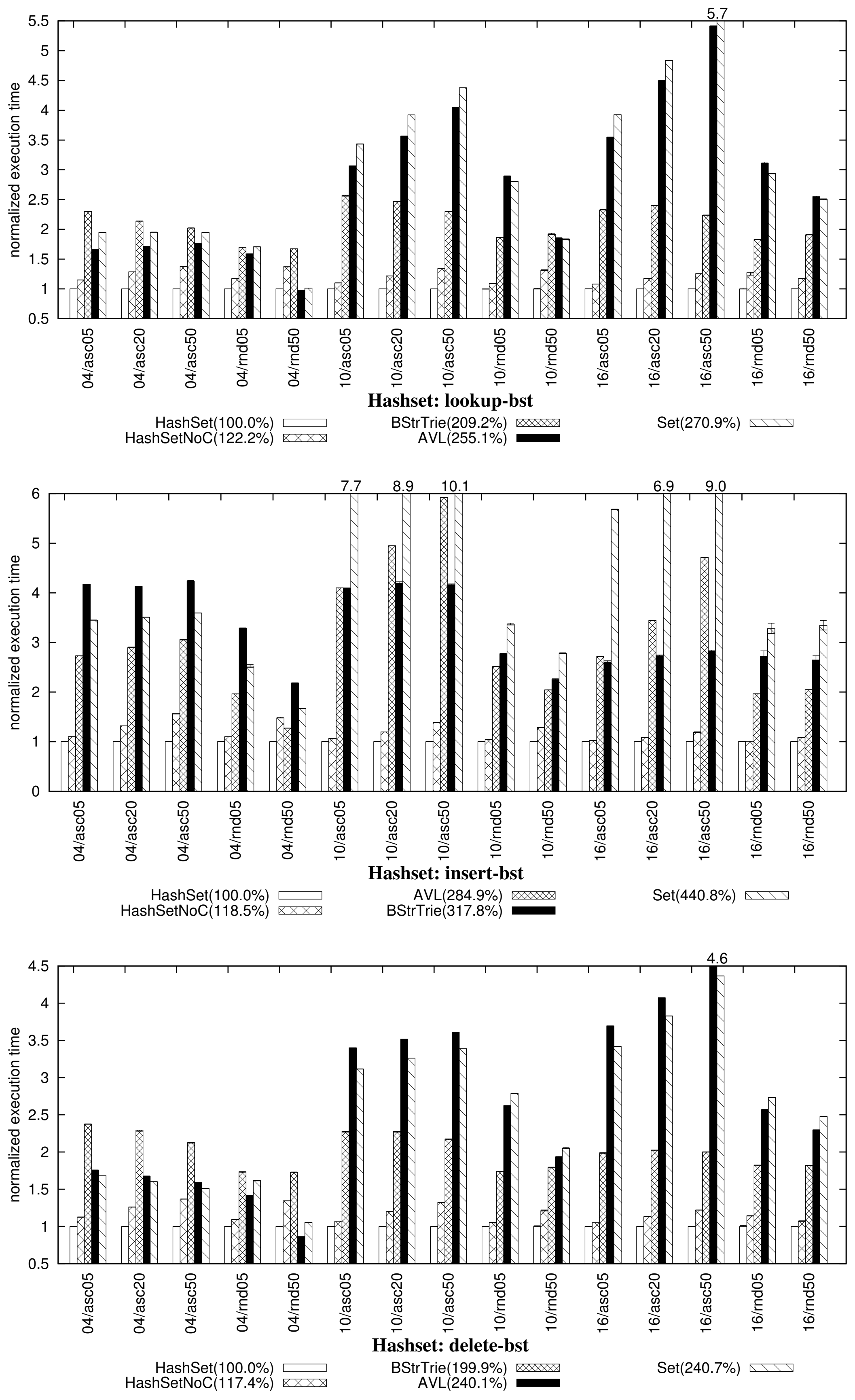
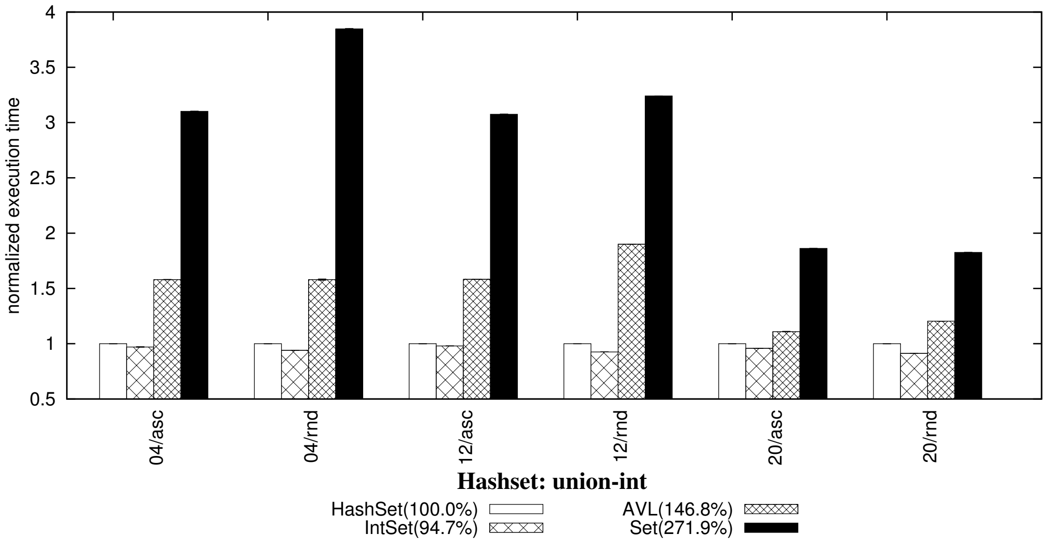
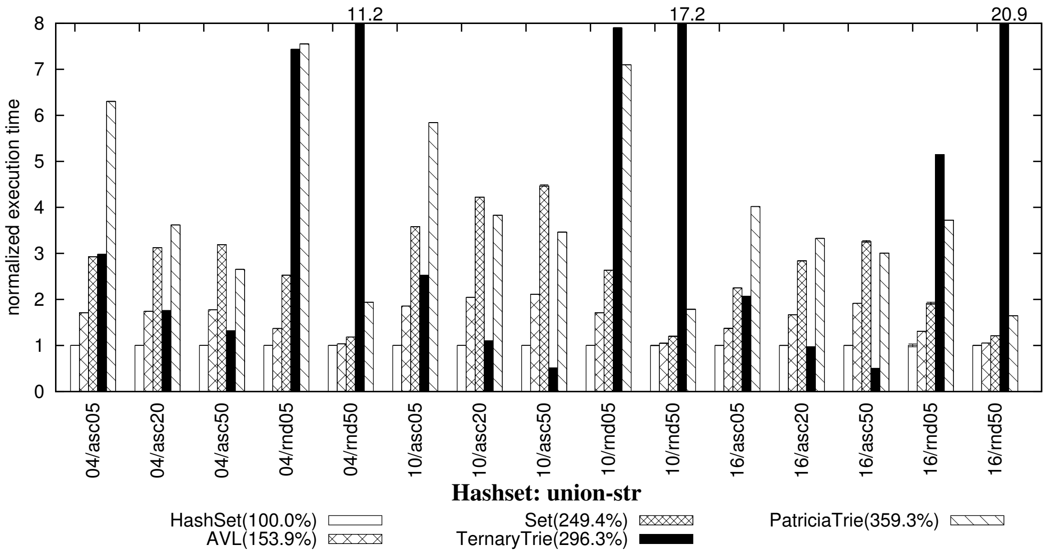


Figure 8.17: Benchmark of hashset operations on `ByteStrings`





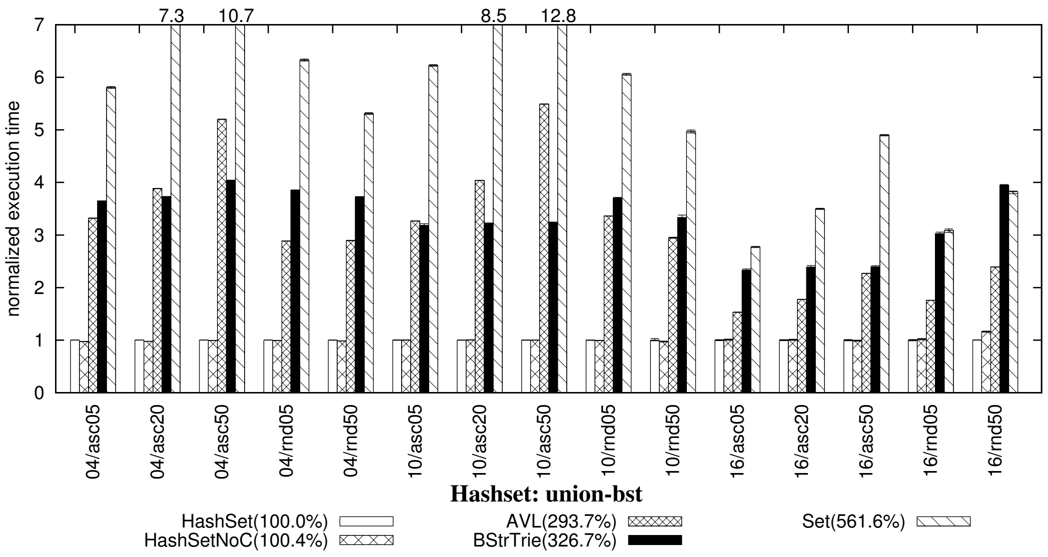


Figure 8.18: Benchmark of `union` operation on hashset

8.5 Chapter Notes

This chapter contains original work and is based on the author's paper [Str10].

Several applications to the Haskell library packages originate from our work on the CONTAINERS package:

- The CONTAINERS package, a widely used standard library package, has been considerably improved, making it comparable and very frequently superior to other implementations available. In addition, the author of this thesis has become a maintainer of this package.
- GHC, a widely used Haskell compiler, has been improved to perform more optimization needed in the CONTAINERS package. Notably, a new specialise pass and an `INLINABLE` pragma were added. Also, the CONTAINERS package is now used in the compiler implementation, instead of in-house data structures.
- People maintaining other data structure packages used in our benchmarks were motivated to improve the performance of their implementations (we know of performance improvements to at least two packages).
- The new HASHMAP package containing an implementation of a new persistent data structure based on hashing has been created. This structure offers supreme performance out of available set implementations with `String` and `ByteString` elements, but should perform well for any element type whose comparison is expensive. This data structure is now available on the HackageDB.

  Our HASHMAP package also inspired Johan Tibell to create another hash trie implementation, with higher branching factors than 2. His package UNORDERED-CONTAINERS is based on [Bag01], which is a generalization of `IntMap` to higher branching factors (usually 16 or 32). Because of the high branching factor, only the nonempty children of a given node are represented, compactly in a continuous block of memory. A bitmap is used to indicate which children are nonempty and provides a mapping between children index and nonempty children index. Because the resulting tree structure has higher branching factor, it has lower depth and `index` is faster then in our HASHMAP package. To compare the `update` and other operations, further benchmarking would be necessary, similar to the experiments in Section 6.2.

Chapter 9

Conclusion

The goal of this thesis was to design new data structures and improve the existing data structures that can be used in functional programming languages.

In the first part of the thesis we focused on persistent data structures, which are structures preserving previous versions of themselves when modified. We started by describing known methods of making data structures persistent. These methods operate effectively on linked data structures of bounded in-degree with worst-case operations. The resulting persistent structures can have either amortized or worst-case bounds. In case of amortized bounds, the complexity of accessing and updating the resulting persistent structure is asymptotically constant. The worst-case complexity of partially persistent structure is also $\mathcal{O}(1)$, however, no method is known which would create fully persistent structures with worst-case constant time operations. The existence of such a method is an interesting open problem.

The main contribution of the first part of the thesis is the study of persistent arrays. We presented simplified implementation of a fully persistent array with optimal amortized complexity. We also presented a worst-case fully persistent array implementation with complexity of $\mathcal{O}(\log\log n + S(n))$, where $S(n)$ is the complexity of operations of dynamic integer set containing n integers bounded by $n^{\mathcal{O}(1)}$. Currently, the best known structure operates in time $\mathcal{O}((\log\log n)^2/\log\log\log n)$ and it is a notable open problem whether this can be reduced to $\mathcal{O}(\log\log n)$. If so, the worst-case persistent array would be optimal.

Utilizing a known reduction of predecessor search problem to persistent array lookup we showed $\Omega(\log\log n)$ lower bound on the lookup complexity of partially persistent array[1] in the cell probe model, where n is the number of array elements. This non-constant lower bound implies several important consequences.

[1]The lower bound and the reduction is mentioned in [DLP08].

Firstly, persistent data structures can have asymptotically worse complexity than ephemeral data structures. Secondly, even though any worst-case data structure can be made fully persistent using fully persistent array to simulate the memory of a RAM, the resulting data structure may be suboptimal.

When using a persistent array in a functional language, versions of the array which cannot be used any more should be removed from the array. However, there are complex dependencies among the array versions in performant persistent array implementations. On that account we designed algorithms that allow efficient recognition and removal of inaccessible versions from the array, making the memory complexity of the persistent array asymptotically optimal in the number of accessible array versions.

In the second part of the thesis we dealt with purely functional data structures. We created a fully persistent array implementation with $\mathcal{O}(b \cdot \log_b n)$ complexity and found the optimal value of the constant b using benchmarking. This complexity is asymptotically optimal when the array is represented by a linked data structure.

The main contribution of the second part of the thesis is the study of Adams' trees, which are fully persistent balanced binary search trees frequently used in functional languages. We corrected the existing proof of Adams' trees balance, which was flawed, causing the existing implementations to sometimes violate the balance condition. We explored the parameter space of Adams' trees in detail and presented correctness proof and performance measurements for the most suitable parameters. In addition, we devised an improved representation leading to reduced memory complexity and faster running times.

Finally, we described in this thesis our contribution to improving the standard Haskell library of purely functional data structures. The library contains implementations of Adams' trees, big-endian Patricia tries (both classic and dense) and 2-3 finger trees annotated with sizes. Currently the library offers the most efficient implementations available and is used by every third Haskell package (as of 14th May 2013, by 1782 out of 5132 packages available on HackageDB, which is a centralized repository of Haskell packages).

To conclude, we devoted this thesis to the study of persistent data structures, discussed their properties and gave bounds on their complexities. We did not limit ourselves to theoretical research, but we also provided efficient implementations of these structures. Persistent data structures remain an interesting field of study with a number of open problems to be solved.

Bibliography

[Ada92] Stephen Adams. Implementing sets efficiently in a functional language. *Technical Report CSTR 92-10*, 1992.

[Ada93] Stephen Adams. Efficient sets – a balancing act. *J. Funct. Program.*, 3(4):553–561, 1993.

[AFK84] M. Ajtai, M. Fredman, and J. Komlós. Hash functions for priority queues. *Information and Control*, 63(3):217 – 225, 1984.

[AHN88] A. Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

[AT07] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.

[AV96] L. Arge and J. Vitter. Optimal dynamic interval management in external memory. In *In FOCS*, 1996.

[AVL62] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 16(2), 1962.

[Bag01] Phil Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.

[Bak78a] Henry G. Baker. Shallow binding in lisp 1.5. *Commun. ACM*, 21(7):565–569, 1978.

[Bak78b] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[Bak91] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26:1991–145, 1991.

[BBC$^+$12] Martin Babka, Jan Bulánek, Vladimír Cunát, Michal Koucký, and Michael Saks. On online labeling with polynomially many labels. In *ESA*, pages 121–132, 2012.

[BCD$^+$02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 152–164, London, UK, 2002. Springer-Verlag.

[BF01] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65:2002, 2001.

[BKS12] Jan Bulánek, Michal Koucký, and Michael Saks. Tight lower bounds for the online labeling problem. In *Proceedings of the 44th symposium on Theory of Computing*, STOC '12, pages 1185–1198, New York, NY, USA, 2012. ACM.

[BKZ76] P. Van Ende Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10(1):99–127, 1976.

[Blu86] Norbert Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM J. Comput.*, 15(4):1021–1024, 1986.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[BO96] Gerth S. Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6:839–857, 1996.

[Bro96] Gerth Stolting Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic J. of Computing*, 3(3):238–255, 1996.

[BS98] Jon Bentley and Robert Sedgewick. Ternary search trees. *Dr. Dobb's Journal*, April 1998.

[BS07] Richard S. Bird and Stefan Sadnicki. Minimal on-line labelling. *Inf. Process. Lett.*, 101(1):41–45, 2007.

[Cam] Taylor Campbell. Reported a bug in the `Data.Map` module of the Haskell CONTAINERS package to the `libraries@haskell.org` mailing list on 3rd August 2010.

[Cha85] Bernard Chazelle. How to search in history. *Information and Control*, 64(1–3):77 – 99, 1985.

[Chu92] Tyng-Ruey Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In *4th European Symposium on Programming*, pages 110–129. Springer–Verlag, 1992.

[Chu94] Tyng-Ruey Chuang. A randomized implementation of multiple functional arrays. In *In Proceedings of 1994 ACM Conference on Lisp and Functional Programming*, pages 173–184. ACM Press, 1994.

[Coh84] Shimon Cohen. Multi-version structures in prolog. In *FGCS*, pages 265–274, 1984.

[Col86] Richard Cole. Searching and sorting similar lists. *J. Algorithms*, 7(2):202–220, June 1986.

[CPT92] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theor. Comput. Sci.*, 106(1):21–60, 1992.

[dB46] N. G. de Bruijn. A Combinatorial Problem. *Koninklijke Nederlandsche Akademie Van Wetenschappen*, 49(6):758–764, 1946.

[DGST88] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, 1988.

[Die82] Paul F. Dietz. Maintaining order in a linked list. In *STOC*, pages 122–127, 1982.

[Die89] Paul F. Dietz. Fully persistent arrays (extended abstract). In *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, pages 67–74, London, UK, 1989. Springer-Verlag.

[DKM$^+$94] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[DLP08] Erik D. Demaine, Stefan Langerman, and Eric Price. Confluently persistent tries for efficient version control. In *SWAT '08: Proceedings* *of the 11th Scandinavian workshop on Algorithm Theory*, pages 160–172, Berlin, Heidelberg, 2008. Springer-Verlag.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[DM85] David P. Dobkin and J. Ian Munro. Efficient uses of the past. *J. Algorithms*, 6(4):455–465, 1985.

[DS87] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 365–372, New York, NY, USA, 1987. ACM.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[DST94] James R. Driscoll, Daniel D. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *J. ACM*, 41(5):943–959, 1994.

[DSZ05] Paul F. Dietz, Joel I. Seiferas, and Ju Zhang. A tight lower bound for online monotonic list labeling. *SIAM J. Discret. Math.*, 18(3):626–637, 2005.

[FK03] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *J. Algorithms*, 48(1):16–58, 2003.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[FS89] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 345–354, New York, NY, USA, 1989. ACM.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[Gou94] Jean Goubault. HimML: Standard ML with fast sets and maps. In *In 5th ACM SIGPLAN Workshop on ML and its Applications*. ACM Press, 1994.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:8–21, 1978.

[HP92] H. Huitema and R. Plasmeijer. *The Concurrent Clean system user's manual, version 0.8.* Technical Report 92-19. University of Nijmegen, Dept. of Informatics, Nijmegen, 1992.

[HP06] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[Hug85] John Hughes. An efficient implementation of purely functional arrays. Technical report, Dept. of Computer Sciences, Chalmers University of Technology, Göteborg, 1985.

[Hug89] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[HY11] Yoichi Hirai and Kazuhiko Yamamoto. Balancing weight-balanced trees. *J. Funct. Program.*, 21(3):287–307, 2011.

[IKR81] 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, UK, 1981. Springer-Verlag.

[KM83] T. Krynen and L.G.L.T. Meertens. *Making B-trees Work for B.* IW 219/83. The Mathematical Centre, Amsterdam, The Nederlands, 1983.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Loe09] Jon Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development.* O'Reilly Media, Inc., 1st edition, 2009.

[LPJ94] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM.

[Mar10] Simon Marlow. Haskell 2010 language report, available online at `http://www.haskell.org/definition/haskell2010.pdf`, 2010.

[Mor68] Donald R. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[Mye83] Eugene W. Myers. An applicative random-access stack. *Inf. Process. Lett.*, 17(5):241–248, 1983.

[Mye84] Eugene W. Myers. Efficient applicative data types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 66–75, New York, NY, USA, 1984. ACM.

[MYPJ07] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 277–288, New York, NY, USA, 2007. ACM.

[NR72] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 137–142, New York, NY, USA, 1972. ACM.

[OB97] Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 7:1–14, 1997.

[OG98] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *ACM SIGPLAN Workshop on ML*, pages 77–86, September 1998.

[Oka99] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, July 1999.

[Ove81a] Mark H. Overmars. Searching in the past I. Technical Report RUU-CS-81-07, Department of Information and Computing Sciences, Utrecht University, 1981.

[Ove81b] Mark H. Overmars. Searching in the past II - general transformations. Technical Report RUU-CS-81-09, Department of Information and Computing Sciences, Utrecht University, 1981.

[Ove83a] Mark H. Overmars. Global rebuilding. In *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*, pages 67–77. Springer Berlin / Heidelberg, 1983.

[Ove83b] Mark H. Overmars. Partial rebuilding. In *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*, pages 52–66. Springer Berlin / Heidelberg, 1983.

[PJ87] Simon Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[PJ$^+$03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[PJRR99] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. `C--`: A portable assembly language that supports garbage collection. In *PPDP*, pages 1–28, 1999.

[PkgCont] The Haskell `containers` package. The latest version is available at `http://hackage.haskell.org/package/containers`.

[PkgCrit] The Haskell `criterion` package. The latest version is available at `http://hackage.haskell.org/package/criterion`.

[PT06] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 232–240, New York, NY, USA, 2006. ACM.

[PT07] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 555–564, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[Pug89] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In *In WADS*, 1989.

[Ram92] Rajeev Raman. *Eliminating amortization: on data structures with guaranteed response time.* PhD thesis, University of Rochester, Rochester, NY, USA, 1992.

[Ram96] Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proceedings of the Fourth Annual European Symposium on Algorithms*, ESA '96, pages 121–137, London, UK, UK, 1996. Springer-Verlag.

[Rou01] Salvador Roura. A new method for balancing binary search trees. In *Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 469–480. Springer Berlin Heidelberg, 2001.

[RTD83a] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, 1983.

[RTD83b] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst.*, 5(3):449–477, 1983.

[Sch80] Arnold Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, 1980.

[SL94] Alexander Stepanov and Meng Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[ST86] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[Str09] Milan Straka. Optimal worst-case fully persistent arrays. Presented at Trends in Functional Programming, 2009.

[Str10] Milan Straka. The performance of the Haskell containers package. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 13–24, New York, NY, USA, 2010. ACM.

[Str12] Milan Straka. Adams' trees revisited - correctness proof and efficient implementation. In *Trends in Functional Programming*, volume 7193 of *Lecture Notes in Computer Science*, pages 130–145. Springer Berlin / Heidelberg, 2012.

[Tar75] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[Tar83] Robert E. Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[Ter09] David A. Terei. Low level virtual machine for Glasgow Haskell Compiler, 2009. BSc Thesis.

[Tol01] Andrew Tolmach. An external representation for the GHC Core Language, 2001. The latest version of the paper is available at `http://www.haskell.org/ghc/docs/papers/core.ps.gz`.

[Tsa84] Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Inf.*, 21(1):101–112, 1984.

[Tur86] D Turner. An overview of miranda. *SIGPLAN Not.*, 21(12):158–166, 1986.

[vEB77] 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.

[Wad76] Philip Wadler. Analysis of an algorithm for real time garbage collection. *Commun. ACM*, 19(9):491–500, 1976.

[Wad87] Philip Wadler. A new array operation. In *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 328–335. Springer Berlin Heidelberg, 1987.

[Wad90a] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[Wad90b] Philip Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[Wil92] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time. *Inf. Comput.*, 97(2):150–204, 1992.

[Yao81] Andrew Chi-Chih Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981.

List of Terms and Abbreviations

BB-ω tree	tree of bounded balance, used in the CONTAINERS package for implementing set and map, Definition 7.2
big-endian Patricia trie	compressed trie storing numbers as bit sequences, used in the CONTAINERS package for implementing intset and intmap, Section 8.1.2
cell probe model	model of computation introduced by [Yao81]
dynamic integer set	structure representing a set of n integers, providing insert, search and delete operations, Chapter 4
exponential tree	dynamic integer set structure, Section 4.2
fat node method	method of creating persistent structures, Sections 2.2.1 and 2.2.2
node copying	method of creating persistent structures, Section 2.2.1
node splitting	method of creating persistent structures, Section 2.2.2
intmap	map with integer keys, Section 8.1.2
intset	set of integers, Section 8.1.2
linked data structure	pointer-based data structure, Definition 2.2
path copying method	method of creating persistent structures, Section 2.1
RAM	random access machine, [Sch80]
vEB tree, vEBT	van Emde Boas tree, Definition 4.1
version list	preorder traversal of version tree, Definition 3.1

List of Figures

Attachments

A.1 Generating Figure 7.3

When generating Figure 7.3 of valid parameters for all trees up to size of 1 million, we used the following code:

```
max_n = 1000000
find_min x p | p x       = last $ x : takeWhile p [x-1, x-2 .. 0]
             | otherwise = head $ dropWhile (not . p) [x+1, x+2 ..]

test w a d = and [delete n m && join n m | n <- [0 .. max_n],
                                           let m = flr $ max 1 (w * n + d)]
 where
  delete n m = n == 0 || rebalance (n-1) m
  join n m = rebalance n (m+increment)
    where increment = max 1 $ ceil ((n+m+1-d) / (w+1) - 1)

  rebalance n m = and [rebalance’ n m x | x <- nub [x_min, x_mid - 1,
                                                    x_mid, m - 1 - x_min]]
    where x_min = find_min (flr $ m     / (w+1)) (\x -> balanced x (m-1-x))
          x_mid = find_min (flr $ m * a / (a+1)) (\x -> x >= a * (m-1-x))

  rebalance’ n m x
    | x < a * y = balanced n x && balanced (n + 1 + x) y
    | otherwise = balanced n s && balanced t y && balanced (n+1+s) (t+1+y)&&
                  balanced n t && balanced s y && balanced (n+1+t) (s+1+y)
   where(y,s,t)=(m-1-x,find_min (flr$x/(w+1)) (\s->balanced s (x-1-s)),x-1-s)

  balanced n m = max 1 (w * n + d) >= m && n <= max 1 (w * m + d)

  flr, ceil :: Double -> Double
  flr = fromInteger . floor
  ceil = fromInteger . ceiling

results = [ (w, a, d, test w a d) | w <- [2, 2.125 .. 5],
                                    a <- [1, 1.125 .. 3], d <- [0 .. 3]]
```

It relies on the fact that when there is a tree which cannot be balanced, there also exists a counterexample with a subtree as large as the balance condition allows. Therefore, for a fixed value of n it is enough to try the largest possible m and for a fixed value of m it is enough to verify that the balance condition is restored when considering the smallest and the largest subtree causing a single rotation and the smallest and the largest subtree causing a double rotation.

A.2 Packages Used in Chapter 8

All packages mentioned in Chapter 8 can be found on the HackageDB, which is a public collection of packages released by the Haskell community. The list of HackageDB packages currently resides at `http://hackage.haskell.org/`.

We used the following packages in the benchmarks:

Package	*Version used*
ARRAY	0.3.0.0
AVLTREE	4.2
BYTESTRING-TRIE	0.1.4
CONTAINERS	0.3.0.0
CRITERION	0.5.0.0
EDISONCORE	1.2.1.3
HASHMAP	1.0.0.3
LIST-TRIES	0.2
PROGRESSION	0.3
RANDOM-ACCESS-LIST	0.2
TERNARYTREES	0.1.3.4
TREESTRUCTURES	0.0.2

Table A.1: Packages used for benchmarks

We also benchmarked internal data structures of the GHC compiler. Their implementation can be found in the sources of GHC 6.12.2, namely the files `FiniteMap.hs` and `UniqFM.hs` in the `compiler/utils` directory.