NTNU – Trondheim
Norwegian University of
Science and Technology

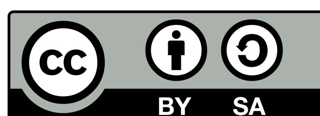
Improving RRB-Tree Performance through Transience

Jean Niklas L'orange

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Copyright © Jean Niklas L'orange, 2014
This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License
http://creativecommons.org/licenses/by-sa/4.0/



Trees sprout up just about everywhere in computer science...

— Donald Knuth

Abstract

The RRB-tree is a confluently persistent data structure based on the persistent vector, with efficient concatenation and slicing, and effectively constant time indexing, updates and iteration. Although efficient appends have been discussed, they have not been properly studied.

This thesis formally describes the persistent vector and the RRB-tree, and presents three optimisations for the RRB-tree which have been successfully used in the persistent vector. The differences between the implementations are discussed, and the performance is measured. To measure the performance, the C library *librrb* is implemented with the proposed optimisations.

Results shows that the optimisations improves the append performance of the RRB-tree considerably, and suggests that its performance is comparable to mutable array lists in certain situations.

Sammendrag

RRB-treet er en sammenflytende persistent datastruktur basert på persistente vektorer, med effektiv konkatenering og kutting, og effektivt konstant tid indeksering, oppdatering og iterasjon. Selv om effektive tilføyinger på enden av et RRB-tre har blitt diskutert, har de ikke blitt studert nærmere.

Denne oppgaven beskriver den persistente vektoren og RRB-treet formelt, og presenterer tre optimaliseringer for RRB-treet som har blitt brukt vellykket i den persistente vektoren. Forskjeller mellom implementasjonene er diskutert, og ytelsen er målt. For å måle ytelsen har C-biblioteket *librrb* blitt implementert med de foreslåtte optimaliseringene.

Målinger viser at optimaliseringene øker ytelsen på tilføyinger i RRB-treet drastisk, og at dens ytelse er sammenlignbar med muterbare arraylister i visse situasjoner.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Magnus Lie Hetland and Ole Edsberg for their support and guidance during my work on this master's thesis. I find it hard to imagine I would be as motivated to work on this topic without the excellent lectures they have provided in *Algorithms and Data Structures* and *Algorithm Construction*.

I would also like to thank my fellow students Christer Bru and Vegard Edvardsen for the participation in numerous programming contests with our team `2b||!2b`, discussions on how to solve different theorems and, the long nights we have spent in the lab together.

I thank my family for their endless love and encouragement.

Last but not the least, thank you Caroline, for always being there for me.

Contents

Abstract a

Acknowledgements c

Contents i

List of Figures iv

List of Listings vi

List of Tables vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Terminology . . . 1
- 1.3 Purpose . . . 2
- 1.4 Organisation . . . 3

I Background 5

2 Persistent Vectors 7
- 2.1 History . . . 7
- 2.2 Introduction . . . 8
- 2.3 Radix Access . . . 12
- 2.4 Update . . . 13
- 2.5 Append . . . 15
- 2.6 Pop . . . 16
- 2.7 Tail . . . 18
- 2.8 Display . . . 21
- 2.9 Performance . . . 24

3 RRB-Trees 33
3.1 Introduction 33
3.2 Relaxed Radix Access 38
3.3 Update 40
3.4 Concatenation 41
3.5 Slicing 49
3.6 Performance 50

4 Transience 53
4.1 Definition 53
4.2 Implementation 56
4.3 Performance 58
4.4 Related Work 59

II Methodology 61

5 Direct Append 63
5.1 Introduction 63
5.2 Implementation 64
5.3 Direct Pop 65
5.4 Performance 66
5.5 Further Optimisations 66

6 RRB-tree Tail 69
6.1 Implementation 69
6.2 Interference with Direct Append 70
6.3 Display 75
6.4 Performance 75

7 RRB-tree Transience 77
7.1 Introduction 77
7.2 Implementation 78

8 librrb and pgrep 81
8.1 librrb 81
8.2 pgrep 84

III Results, Discussion and Conclusion 87

9 Results and Discussion 89
9.1 Practical 89
9.2 Append Time 90
9.3 Concatenation Time 94
9.4 Overall Time 98
9.5 Memory Usage 100

10 Conclusion **105**
10.1 Future Work 105

IV Appendices **107**

A Additional Functions **109**
A.1 Concatenation Functions 109
A.2 Direct Append Helper Functions 111
A.3 Direct Pop 113

B Search Terms **115**

C Additional Plots **117**

D librrb Installation and Use **119**
D.1 Setup 119
D.2 Options 119
D.3 Interface 120

Bibliography **125**

List of Figures

2.1 Persistent vector illustration 9
2.2 Persistent vector operations 14
2.3 Persistent vector pop and tail 19
2.4 No display vs. display 21
2.5 Plot of $h(P)$ for different M. 25
2.6 Overhead ratio for different M. 30
2.7 Persistent vector indexing and update times. 31
3.1 Redistribution of nodes. 36
3.2 Trie containing size tables. 38
4.1 Transient vectors. 57
5.1 Direct appending 64
6.1 RRB-trees breaking and satisfying tail invariant 72
7.1 RRB-tree with tail and transient 79
8.1 Different computation stages in the parallel grep implementation. . . 84
8.2 A parallel concatenation with $P = 8$ threads. 85
9.1 Time used in line search phase 91
9.2 Total time used in search filtering phase 92
9.3 Branching factor effects on line search phase 94
9.4 Branching factor effects on search filter phase 95
9.5 RRB-tree concatenation time 96
9.6 Concatenation time with array 96
9.7 Branching factor effects on concatenation 97
9.8 Total running time for `pgrep`, excluding I/O 98
9.9 Branching factor effects on total runtime 99
9.10 Total memory usage for different optimisation permutations. 101
9.11 Memory overhead ratio for different optimisation permutations. . . . 101
9.12 Memory overhead ratio for different branching factors. 104
9.13 Total memory usage for different branching factors. 104
C.1 Box-and-whisker plot for concatenation phase. 117

List of Listings

2.1 Implementation of PVEC-LOOKUP. 12
2.2 Improved implementation of PVEC-LOOKUP. 13
2.3 Implementation of UPDATE. 14
2.4 Implementation of APPEND. 15
2.5 Implementation of POP. 18
2.6 Implementation of UPDATE, using a display. 22
3.1 Relaxed radix search. 39
3.2 RRB-tree update. 40
3.3 Algorithm to create a concatenation plan. 45
4.1 CLONE function used in transients. 56
4.2 Transient usage in Clojure 59
5.1 RRB-PUSH using direct appending 65
6.1 Code in RRB-CONCAT to satisfy RRB tail invariant. 73
8.1 Example of manual loop unrolling 82
8.2 Concatenation algorithm. 86
A.1 CONCAT-SUB-TRIE. 109
A.2 Rebalancing algorithm for RRB-CONCAT. 110
A.3 Implementation of COPYABLE-COUNT. 111
A.4 Implementation of COPY-FIRST-K. 112
A.5 Implementation of APPEND-EMPTY. 112
A.6 RRB-POP using direct popping 113
D.1 Shell commands for system setup. 120

List of Tables

2.1 Time complexity of persistent vector operations. 32
3.1 Time complexity of RRB-tree operations. 51
9.1 Line search times 90
9.2 Search filter time for 1.5 million matches 93
9.3 Total runtime for 1.5 million matches 99
9.4 Branching factors and memory usage. 103
B.1 List of search terms 115
D.1 List of configurable options 121

Chapter 1

Introduction

1.1 Motivation

With more and more multi-core systems, programmers have to manage more complex environments than previously. Immutable data structures give programmers a tool that is easy to reason around. In addition, they enhance the power of other tools able to cope with the increased complexity, for example software transactional memory[1] and functional programming languages. In many cases, however, the increased overhead of immutable structures compared to mutable structures makes them unattractive for parallelisation and performance-based programs.

Parallel processing of lists is generally easy, with options easily available for the JVM and C/C++. The Fork/Join framework[2] is available for programmers using the JVM, whereas MPI and OpenMP is available for C/C++ developers. If none of these are available, it is usually not difficult to implement a parallel processing task with threads, albeit with an inferior scheduling algorithm compared to the frameworks mentioned above. However, efficient parallel processing of lists where the size of the output list is not known in advance require concatenation, a task that is hard to perform efficiently without paying performance penalties on other operations.

The recently designed Relaxed Radix Balanced Tree (RRB-Tree) by Bagwell and Rompf [3] attempts to be an efficient immutable data structure for parallel processing, and extends the persistent vector implementation from the Clojure programming language[4]. RRB-Trees maintain the near constant time costs of updates and indexing, while providing efficient concatenation and splits. However, the paper does not assess the need for concatenation and splits, in contrast to more efficient appends, updates and indexing. Additionally, the paper discusses constant time additions through a *tail*, but it does not specify how to efficiently insert the tail into the RRB-tree itself.

1.2 Terminology

This thesis uses a different notion of the word *persistence* than commonly used in computer science: Rather than meaning something which is stored in persistent

storage, persistence/persistent data structures refer to data structures that always preserve their original version when “updated”.

An *immutable data structure* is a data structure that does not change its contents after creation. Allowing mutability during creation is sometimes considered to be *effectively immutable*. Although the data structures covered in this thesis are usually implemented that way, we will not distinguish between these terms.

The term *mutable* refers to data structures which has an *identity* with a certain *state* at a specified point in time. All functions modifying this structure take the identity and additional arguments, and modify the current state. Driscoll et al. uses the term *ephemeral* to refer to these structures[5].

The term *transience/transient* refers to a different class of data structures, which is further explained in Chapter 4. It is not related to the definition in Scala/Java, which is used to indicate that a class field is not part of the persistent state (within the context of persistent storage) of an object.

Although $\ll$ and $\gg$ are usually used to denote much less than and much greater than, we will in this thesis use those symbols for the bitshift operations “shift left” (SHL) and “shift right” (SHR).

NIL represents the absence of a value: This is commonly known as the *null* pointer in Java, C and C++, or *Nothing* in the Maybe monad in Haskell.

In algorithms, the iteration “**for** $i \leftarrow a$ **to** b” means that the iteration is done from a up to and including b, incrementing i by one for each step. The iteration “**for** $i \leftarrow a$ **downto** b” means that the iteration is done from a down to and including b, decrementing i by one for each step. If the additional keyword **by** is included, then the variable iterated over is decremented or incremented by the following value.

The expressions T_i and $T[i]$ are equivalent, and denote the ith element in the array T or slot entry in the trie node T. The expression $T_i[j]$ denotes the jth entry in T_i. $|P|$ refers to the length of a trie, vector or RRB-tree.

In garbage collected programming environments, memory leaks may only happen as a result of bugs in the environment or garbage collector. However, data structures may retain pointers to data which can never be accessed by a running program. The data such pointers point to is called *semantic garbage*.

1.3 Purpose

The purpose of this thesis is threefold: First and foremost, it attempts to improve appending for RRB-trees in order to make it a reasonable data structure to use for parallel programming. We assume that three ideas from the persistent vector may result in considerable speed increases:

1. Direct appending
2. Tails
3. Transience

Second, as the persistent vector has not been properly described in an academic context, an attempt to formalise and prove runtimes analytically is done. Additionally, claims by Bagwell and Rompf on the runtime of the RRB-tree are studied analytically. A consequence of this is that the background chapters may be somewhat longer and more detailed than what is considered usual.

Finally, the RRB-tree is implemented as a C library for two reasons: To make it easier for future data structure developers to have some easily available reference implementation in a well-known programming language, and to make the data structure available in programming languages that are able to use C libraries.

1.4 Organisation

This thesis consists of three parts: The background part, the methodology part, and the result part. The background part consists of three chapters: Chapter 2 describes in detail the persistent vector, which the RRB-tree is based upon. Chapter 3 describes the RRB-tree, operations on it and its performance compared to the persistent vector. Chapter 4 describes *transience* a category of effectively immutable data structures that do not “persist”.

The methodology part consists of four chapters, one for each optimisation, and one for the implementation. Chapter 5 describes direct append, an algorithm which inserts elements directly into the RRB-tree. Chapter 6 explains how one adds a tail to the RRB-tree, and how the original RRB-tree algorithms must change to preserve correctness. Chapter 7 elaborates on how transient RRB-trees can be implemented. Chapter 8 explains the implementation of the RRB-tree library *librrb*, along with the benchmark program *pgrep*.

The result part consists of two chapters: Chapter 9, which contains results of benchmarks, along with a discussion, and Chapter 10, which concludes this thesis and presents future work.

Part I

Background

Chapter 2

Persistent Vectors

2.1 History

Functional programming language researchers have continuously studied immutable data structures, in order to try to improve their performance. Finding a good data structure with similar performance guarantees as mutable, growable arrays has been of particular interest, but this has proven to be a hard problem. Using the well known singly linked list gives good performance for operations modifying the head of the list, but worst case $\mathcal{O}(n)$ time for random access and modification. On the other hand, using an immutable array gives optimal random access time, but $\Theta(n)$ time for removal, insertions and updates. Additionally, every update requires a new array to be stored in memory, which is costly.

Okasaki attempts to solve this problem through *random-access lists*[6]. The structure provides lookup and update operations in $\mathcal{O}(\log n)$ time, while retaining $\mathcal{O}(1)$ time for the common singly linked list operations `first`, `rest` and `cons`. Bagwell also attempts to solve this problem with *VLists*[7]. In contrast to Okasaki's random-access lists, the VList focuses more on practical considerations, and attempts to reduce both cache misses and space usage, while providing $\mathcal{O}(n)$ time for update, $\mathcal{O}(\log n)$ access time and $\mathcal{O}(1)$ for the singly linked list operations. Both of these have their use cases, but have in general been superseded by the persistent vector implementation first provided in the programming language Clojure. The persistent vector is a bit-array mapped trie, where the branching factor is very high. As a result, the trie trees are incredibly shallow, and whereas operations on the tree is in theory $\mathcal{O}(\log_{32} n)$, they are effectively constant time as the tree will never have a height over 7.

The persistent vector was pioneered by Rich Hickey in July 2007 for the programming language Clojure, influenced by Bagwell's paper *Ideal Hash Trees*[8]. Initially, the persistent vector functions were not optimised, and consisted only of access, updates, pop and insertions. In March 2008, Hickey invented the *tail*, claiming to improve bulk insertion and removal times with a factor of around 2 to 5[9]. In August 2009, the notion of *transience* was introduced to the structure, providing even faster bulk operations, with a factor of about 10 in specific cases[10].

In September 2009 Tiark Rompf implemented the initial implementation of the immutable vectors in Scala, first available in version 2.8.0. The notion of a *focus* and *display* were invented for this implementation, created as an attempt to further reduce cache misses. The implementation also has a faster insertion algorithm for inserting at the front of the vector. In contrast to the persistent vector implementation in Clojure, Scala's implementation does not provide transients as of this writing. Implementations of the persistent vector also exist for Factor[1], Java[2], Haskell[3] JavaScript[4], and other languages.

2.2 Introduction

A persistent vector P is an immutable variant of an M-way branching radix trie, using element indices as keys. For a refresher on tries, see Knuth [11, pp. 492-507]. As the persistent vector trie differs somewhat from a normal trie, the structural differences will first be explained briefly. To elaborate and prove the performance characteristics of the persistent vector, a more formal definition along with informal explanations will be given afterwards.

A trie used in a persistent vector can either be a leaf node or an internal node. Internal nodes only contain table entries to other tries, and leaf nodes only contain table entries to elements in the vector. Both type of nodes will contain at least one table entry, and at most M table entries: The only exception is the empty trie, which is a single leaf node containing zero table entries. The amount of table entries for a trie node T is denoted $\|T\|$, and table entries are looked up by an integer from 0 to $M-1$. The trie nodes we will work with are contiguous, meaning that if the table entry i exists and is not 0, then table entry $i-1$ also exists. In addition, if a trie has a sibling on its right hand side, then this trie has to be fully populated. Finally, all leaf nodes in a persistent vector will be at the same level, which means we have to walk through the same amount of nodes before we can return or update the value asked for.

Figure 2.1 visualises such a vector P with a branching factor of $M = 2$. The vector contains the values 1, 2, 3, 4 and 5 in that order. The top structure contains the size of the vector (5), along with its *shift* (4) and a pointer to the trie root, denoted P_{root}. The shift is an optimised way to denote the height of the trie for specific persistent vectors, and will be explained in detail later.

These properties are captured formally by the definition of a *leftwise dense* persistent vector, which depend on the definition of a trie's height and a *fully dense* persistent vector:

Definition 2.1. *The height of any trie* T *is defined as:*

$$\begin{aligned} & h(T) = 0 && \textit{if } T \textit{ is a leaf node} \\ \forall\, 0 < i < \|T\|,\ & h(T) = h(T_i) + 1 && \textit{if } T \textit{ is not a leaf node} \end{aligned}$$

[1] http://factorcode.org/
[2] https://github.com/krukow/clj-ds
[3] https://hackage.haskell.org/package/persistent-vector
[4] http://swannodette.github.io/mori/

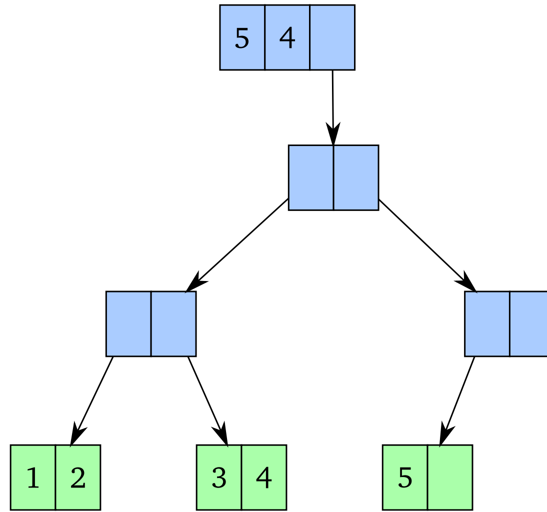


Figure 2.1: Persistent vector illustration

The height of a persistent vector P *is the height of its root trie:* $h(P) = h(P_{root})$.

From Definition 2.1, it follows that leaves in a persistent vector P will only exist exactly $h(P)$ search steps away from the root of the trie.

Definition 2.2. *Let* $d_{full}(T)$ *denote that the trie* T *is* fully dense. *An* M*-way branching trie* T *containing type* τ *is* fully dense *if and only if* $\|T\| = M$ *and*

$$\begin{cases} \bigwedge_{i=0}^{M} T_i : \tau & \text{if } h(T) = 0 \\ \bigwedge_{i=0}^{M} d_{full}(T_i) & \text{if } h(T) \neq 0 \end{cases}$$

Definition 2.3. *Let* $d_{left}(T)$ *denote that the trie* T *is* leftwise dense. *An* M*-way branching trie* T *containing type* τ *is* leftwise dense *if and only if* $0 < \|T\| \leqslant M$ *and*

$$\begin{cases} \bigwedge_{i=0}^{\|T\|} T_i : \tau & \text{if } h(T) = 0 \\ \left(\bigwedge_{i=0}^{\|T\|-1} d_{full}(T_i) \right) \wedge d_{left}(T_{\|T\|-1}) & \text{if } h(T) \neq 0 \end{cases}$$

If a persistent vector P *is leftwise dense, its root trie* P_{root} *must be leftwise dense, and either* $h(P) = 0$ *or* $1 < \|P_{root}\|$ *must be satisfied.*

Informally, this means that all the nodes in a leftwise dense persistent vector will always be fully dense, excluding the rightmost nodes. The additional root constraint ensures that the height of a persistent vector always will be minimal.

It is possible to relax the leftwise density constraint, but we will not focus on such implementations and its implications in this thesis. Therefore, a vector will from now on be considered to be a leftwise dense vector unless otherwise noted.

While these definitions are sufficient to ensure that the persistent vector is structurally correct, they do not convey any relation between the height and size. This is essential in order to reason around the performance characteristics of a persistent vector. The following definitions and theorems prove that $h(P) \in \Theta(\log_M |P|)$.

Definition 2.4. *The size of any trie* T *is defined as:*

$$
\begin{aligned}
|T| &= \|T\| && \textit{if } T \textit{ is a leaf node} \\
|T| &= \sum_{i=0}^{\|T\|} |T_i| && \textit{if } T \textit{ is not a leaf node}
\end{aligned}
$$

The size of a persistent vector P *is the size of its root trie:* $|P| = |P_{root}|$.

Note that $\|T\|$ is the amount of table entries in a trie, whereas $|T|$ refers to the total elements contained in T.

Theorem 2.1. *If* $d_{full}(T)$*, then* $|T| = M^{h(T)+1}$.

Proof. For $h(T) = 0$, $|T| = M^{h(T)+1} = M^{0+1} = M$, as by Definition 2.2.

Next, assume the theorem is true for heights $\leqslant k = h(T')$. We must prove that the hypothesis holds for $k + 1 = h(T)$.

The theorem is true (by the inductive hypothesis) for all the subtries $0 < T_i < M$, since they have the height $h(T_i) = h(T) - 1 = k$ by Definition 2.1. As this trie is not a leaf node, the total size of the trie is

$$
\begin{aligned}
|T| &= \sum_{i=0}^{M} |T_i| = \sum_{i=0}^{M} M^{h(T_i)+1} \\
&= M \times M^{h(T)} \\
&= M^{h(T)+1}
\end{aligned}
$$

□

Corollary 2.1. *If* $d_{full}(T)$*, then* $h(T) = \log_M(|T|) - 1$.

Proof. This follows from the logarithmic definition

$$
|T| = M^{h(T)+1} \Leftrightarrow h(T) + 1 = \log_M(|T|)
$$

□

Additionally, we define the capacity of a vector as follows:

Definition 2.5. *The maximal capacity of a leftwise dense trie* T *is*

$$cap(T) = M^{h(T)+1}$$

A leftwise dense vector may only increase its height when we insert elements. This happens when the original vector size is equal to its maximal capacity, $|P| = cap(P)$, in which it will be fully dense.

Given Theorem 2.1, we can now define a relation between the length of a leftwise dense persistent vector and its height:

Theorem 2.2. *Given a* leftwise dense *persistent vector* P*,*

$$\begin{aligned} h(P) &= 0 && \text{if } |P| \leqslant M \\ h(P) &= \lceil \log_M(|P|) \rceil - 1 && \text{if } M < |P| \end{aligned}$$

Proof. For $|P| \leqslant M$, all elements will be placed inside a single leaf node P_{root}, which satisfies Definition 2.1. The theorem is therefore true for $0 \leqslant |P| \leqslant M$.

Next, assume $M^k < |P| \leqslant M^{k+1}$, $k > 0$. Because $M < |P|$, $h(P) > 0$, the root node $T = P_{root}$ must satisfy the property $1 < |T|$ from Definition 2.3. It follows that $d_{full}(T_0)$. From Corollary 2.1, we have that

$$h(T_0) = \log_M(|T_0|) - 1$$

and as $h(T_0) = h(T) - 1$ from Definition 2.1, $h(T) = \log_M(|T_0|)$. It is evident that $|T_0| = M^k$, consequently $h(T) = k$.

The minimum value of $|P|$ is $M^k + 1$, and the maxium value is M^{k+1}. As

$$\lceil \log_M(M^k + 1) \rceil - 1 = \lceil \log_M(M^{k+1}) \rceil - 1 = k$$

and as $\lceil \log_M(x) \rceil$ is monotonic, it follows that

$$h(P) = \lceil \log_M(|P|) \rceil - 1$$

for all $M^k < |P| \leqslant M^{k+1}$. □

Corollary 2.2. $h(P) = \lceil \log_M(|P|) \rceil - 1 \in \Theta(\log_M(|P|))$

Proof. This is evident from the definition of Θ. □

2.3 Radix Access

The path down to a single element in a persistent vector is calculated through its index. The search step is similar to the big-endian binary tries in [12], but the branching factor is M instead of 2: For each node T, the table entry to continue walking is $i' = \left\lfloor \frac{i}{M^{h(T)}} \right\rfloor \mod M$, until we have found the value. Listing 2.1 represents a more efficient algorithm which returns the same result. It utilises the fact that $\frac{M^{h(T)}}{M} = M^{h(T)-1}$ to avoid recomputing powers of M. Assuming we store $M^{h(T)}$ in the vector head, the lookup still has to perform $h(P)$ modulo operations and $2h(P)$ integer divisions, which are relatively costly assembly operations.

```
1  function PVEC-LOOKUP(P, i)
2      T ← P_root
3      d ← M^h(T)
4      for h ← h(T) downto 0 do
5          i' ← ⌊i/d⌋ mod M
6          T ← T_i'
7          d ← d/M
8      end for
9      return T                                ▷ T is element here
10 end function
```

Listing 2.1: Implementation of PVEC-LOOKUP.

However, a more efficient version exists if we constrain M. Recall the identities for the bit operations $\ll$, $\gg$ and &:

Corollary 2.3. $a \ll c = a \times 2^c$

Corollary 2.4. $a \gg c = \left\lfloor \frac{a}{2^c} \right\rfloor$

Corollary 2.5. $a \,\&\, (M - 1) = a \mod M$

where $M = 2^b$ for some integer $b \geqslant 0$. We can therefore reformulate the formula for i' as follows:

$$
\begin{aligned}
i' &= \left\lfloor \frac{i}{M^{h(T)}} \right\rfloor \mod M \\
&= \left\lfloor \frac{i}{2^{b \times h(T)}} \right\rfloor \mod 2^b \\
&= \left\lfloor \frac{i}{2^{b \times h(T)}} \right\rfloor \,\&\, (2^b - 1) && \text{by Corollary 2.5} \\
&= (i \gg (b \times h(T))) \,\&\, (M - 1) && \text{by Corollary 2.4}
\end{aligned}
\tag{2.1}
$$

While general integer division, modulo and exponentiation performance has improved over the years, bitshift operations make division, modulo and exponentiation with

powers of 2 still significantly more efficient than integers that are not a power of 2 [13]. M is therefore usually a power of two for performance reasons.

As an example of a lookup using the formula, consider a case where $b = 5$ and $M = 2^5$ with the vector P. Assume $h(P) = 2$, which implies $M^{h(P)} = M^2 = 2^{10} < |P|$, and that we want to look up $i = 703_{10}$ in P. It is possible to visualise the lookup as partitions of 5 and 5 bits, as shown in Equation (2.2).

$$i = 703_{10} = \overbrace{\underbrace{00000}_{i_2}\underbrace{10101}_{i_1}\underbrace{11111}_{i_0}{}_2}^{\text{binary representation}} \tag{2.2}$$

Now, for each trie T, we walk down table entry $i_{h(T)}$. Calculating the table entry $i_{h(T)}$ is done through (2.1). Assuming we can calculate the shift calculation $s(T) = b \times h(T)$, i's shift for trie T, efficiently, this new formula would require few expensive operations compared to the original radix access algorithm. In Equation (2.2), we would first walk table entry 0, then 21 and finally the element at index 31.

By realising that $s(T_j) = s(T) - b$ for all $0 < j \leqslant \|T\|$ and that $h(T') = 0 \implies s(T') = 0$, we can avoid storing the height of the root trie and store its shift instead. Applying these new ideas, we can improve the radix access implementation in Listing 2.1 to the bitwise access implementation presented in Listing 2.2.

```
1  function PVEC-LOOKUP(P, i)
2      T ← P_root
3      for s' ← s(T) downto 0 by b do
4          i' ← (i ≫ s') & (M − 1)
5          T ← T_i'
6      end for
7      return T                                  ▷ T is element here
8  end function
```

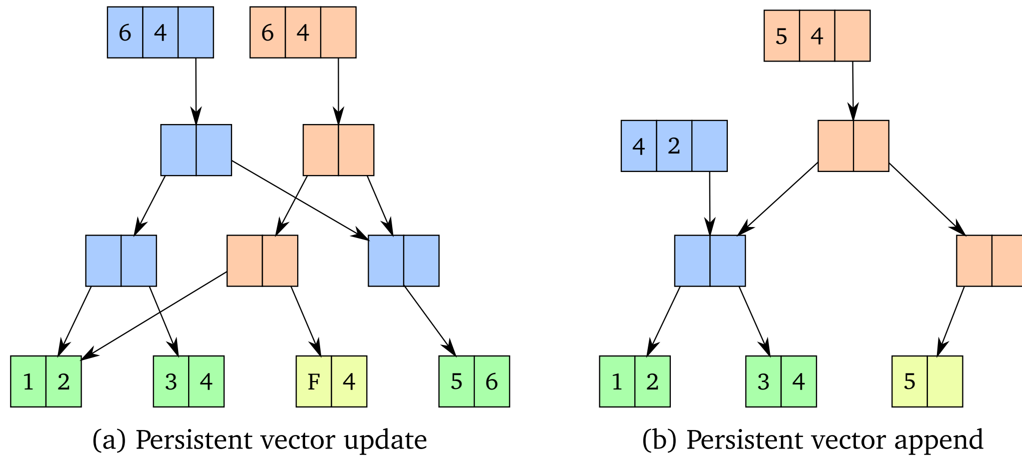
Listing 2.2: Improved implementation of PVEC-LOOKUP, given $M = 2^b$.

It is also possible to remove calculation overhead through loop unrolling. As all operations performed at height h is also performed at height $h - 1$, we can represent the branching loop as a `switch` statement. In a `switch` statement, h_{max} would be the first case, $h_{max} - 1$ the second case and so on. The last case, $h = 0$, might need special treatment as its return value would be of type τ, not a subtrie.

As every lookup require $h(T)$ search steps, the time complexity to lookup any value in a vector is $\Theta(\log_M n)$ where $n = |T|$.

2.4 Update

Replacing the element at index i in the vector P is done through *path copying*: copying all nodes affected by the update. The trie is traversed as by Listing 2.1, but at each



(a) Persistent vector update (b) Persistent vector append

Figure 2.2: Persistent vector operations

step, we perform a shallow copy of the node we need to walk to avoid modifying the original vector. Sarnak and Tarjan coined the term path copying and discuss it in [14].

Using Listing 2.1 as a basis for the update function, we can implement update as presented in Listing 2.3. The bit optimisations in previous section can of course be applied to improve performance.

```
1  function UPDATE(P, i, val)
2      P' ← CLONE(P)
3      T ← CLONE(P_root)
4      P'_root ← T
5      d ← M^h(T)
6      for h ← h(P') downto 1 do
7          i' ← ⌊i / M^h⌋ mod M
8          T_i' ← CLONE(T_i')
9          T ← T_i'
10     end for
11     i' ← i mod M
12     T_i' ← val
13     return P'
14 end function
```

Listing 2.3: Implementation of UPDATE.

Figure 2.2a shows two vectors with a branching factor of $M = 2$. The left vector contains the elements `[1, 2, 3, 4, 5, 6]`, whereas the right vector is an updated version, replacing 3 with F. We can see that, in contrast to a full copy of a structure, only $h(P)$ nodes are copied, and most of the data, including internal nodes, is shared.

Except for the path copying, an update is equivalent to a lookup. Assume that a single

copy requires $\mathcal{O}(M)$ time, and that a copy has to be done at each search step. An update therefore requires $\mathcal{O}(M \log_M(n))$ time to run.

2.5 Append

Inserting a new element in a vector P is easiest described as an update where $i = |P|$, and where we create empty nodes when the node to walk is NIL. In addition, we have to increase the height if the vector is fully dense, by creating a new root node.

```
1  function APPEND(P, val)
2      P′ ← CLONE(P)
3      i ← |P|
4      |P′| ← |P| + 1
5      if d_full(P_root) then
6          n ← CREATE-INTERNAL-NODE()
7          n_0 ← P_root
8          P′_root ← n
9          h(P′) ← h(P) + 1
10     else
11         P′_root ← CLONE(P_root)
12     end if
13     T ← P′_root
14     for h ← h(P′) downto 1 do
15         i′ ← ⌊i / M^h⌋ mod M
16         T_i′ ← CLONE-OR-CREATE(T_i′)        ▷ Creates empty node if T_i′ = NIL
17         T ← T_i′
18     end for
19     i′ ← i mod M
20     T_i′ ← val
21     return P′
22 end function
```

Listing 2.4: Implementation of APPEND.

In the worst case, we have $|P| = cap(P) - 1$. In that case, we have to copy $h(P)$ nodes of size M. If $d_{full}(P)$, we have to perform $h(P) + 2$ node creations. This gives us a time complexity of $\mathcal{O}(M \log_M(n))$, the same runtime as updates.

While Listing 2.4 presents a rather straightforward implementation, it does not explain how we detect $d_{full}(P)$ efficiently. Assuming that the vector head contains its height, $h(P)$, we can check whether its current size is equal to $M^{h(P)+1}$, from Theorem 2.1. The same idea applies when $M = 2^b$, but we can use the more efficient $1 \ll (s(P) + b) = M \ll s(P)$ by using Corollary 2.3.

2.6 Pop

Whereas the append algorithm creates new nodes whenever the radix search attempts to walk a node which is NIL, the pop algorithm removes completely empty nodes from the new vector. Additionally, to ensure that the new vector is leftwise dense, the root node is replaced with its child if it only has a single child. By doing so, we ensure that the height is minimal.

A naïve first attempt might start by walking down the index $i = |P'| = |P| - 1$ down until it has reached the leaf node and removed the last element. It then recursively returns the newly created node to its parent, or return NIL if the node is completely empty. Finally, it checks that the root node's second child is not equal to NIL, and if it is, the root is replaced with its child.

An improved solution realises that the new size can be used to avoid walking parts of the trie. Namely, the following tricks can be used:

1. If the new size is $M^{h(P)}$, the height of the new trie is reduced by one.
2. If all the remaining search steps to walk are though table entry 0, short- circuit the walk and return NIL.

1. is evident: It is easy to show that the only vector with size $M^{h(P)}$ is a fully dense trie of height $h(P) - 1$. From this, we easily realise that the new vector root will be the the first child of the original vector.

1. is a special case of 2., in which one has to shorten the trie height in addition to short-circuit. 2. avoids the additional jumps down the trie even when the height is not changing, and as such avoids unneeded work.

To see why 2. holds, assume the last n search steps indices are zero, and that the $n + 1^{th}$ to last search step index is > 0. The element to remove in the leaf node has index 0. As P is leftwise dense, there are no elements right of this element. Consequently, the leaf node will be empty and we will therefore return NIL. The same logic applies for the level above: NIL will be placed at index 0, meaning that this node is empty, and we return NIL. This is repeated until we reach the level where we are not replacing the new path at index 0, which is the $n + 1^{th}$ to last search step.

The remaining question is then how one efficiently calculates that the remaining n steps all are through table entry 0. This is shown in the following theorem:

Theorem 2.3. *For any non-negative integers* n, h *and positive integer* M,

$$\forall_{i=0}^{h} \left\lfloor \frac{n}{M^i} \right\rfloor \mod M = 0 \Leftrightarrow n \mod M^{h+1} = 0$$

Proof. Let

$$n = \sum_{i=0}^{\infty} m_i M^i$$

where all m_i are non-negative integers less than M. Then

$$n \mod M^{h+1} = \sum_{i=0}^{h} m_i M^i \tag{POP-MOD}$$

and

$$\left\lfloor \frac{n}{M^j} \right\rfloor = \sum_{i=0}^{\infty} m_{(i+j)} M^i \tag{POP-QUOT}$$

By (POP-MOD) and (POP-QUOT), we then get

$$\left\lfloor \frac{n}{M^j} \right\rfloor \mod M = m_j \tag{POP-PART}$$

therefore

$$\overset{h}{\underset{i=0}{\forall}} \left\lfloor \frac{n}{M^i} \right\rfloor \mod M = 0 \Leftrightarrow \overset{h}{\underset{i=0}{\forall}} m_i = 0$$

Additionally, we get that

$$n \mod M^{h+1} = \sum_{i=0}^{h} m_i M^i = 0 \Leftrightarrow \overset{h}{\underset{i=0}{\forall}} m_i = 0$$

as all m_i must be zero for its modulus to be zero.

Through substitution, we then get

$$\overset{h}{\underset{i=0}{\forall}} \left\lfloor \frac{n}{M^i} \right\rfloor \mod M = 0 \Leftrightarrow n \mod M^{h+1} = 0$$ □

By Theorem 2.3, we see that we only have to check

$$(|P| - 1) \mod M^{h(T)+1} = 0$$

to confirm whether the subtrie T and all its children will all walk table entry 0, and thus return NIL. We can therefore short-circuit if this happens, and we avoid walking the trie. By using the binary ideas from Section 2.3, we can convert this to the expression

$$(|P| - 1) \,\&\, ((2M \ll s(T)) - 1)$$

Figure 2.3a presents a normal vector pop, where only the leaf node is modified and the path down to it is copied. It is tempting to visualise a tree where the height changes, but such a figure would be equivalent to Figure 2.2b and is therefore not included.

```
1  function POP(P)
2      P' ← CLONE(P)
3      i ← |P| − 1
4      |P'| ← |P| − 1
5      if |P'| = M^h(P) and |P'| ≠ 1 then
6          h(P') ← h(P) − 1
7          P'_root ← P_root[0]
8      else
9          P'_root ← CLONE(P_root)
10         T ← P'_root
11         for h ← h(P') downto 0 do
12             i' ← ⌊i / M^h⌋ mod M
13             if |P'| mod M^h = 0 then                    ▷ Child will return NIL
14                 T_i' ← NIL
15                 return P'
16             end if
17             T_i' ← CLONE(T_i')
18             T ← T_i'
19         end for
20     end if
21     return P'
22 end function
```

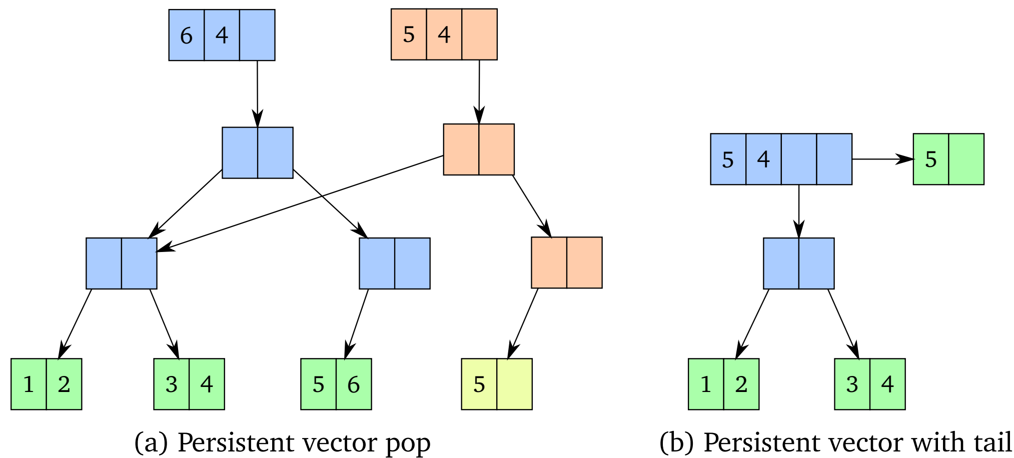
Listing 2.5: Implementation of POP.

2.7 Tail

While the persistent vector focuses on efficient random access operations, many operations work around the end of a vector. A considerable amount of tasks also perform operations in bulks. Examples include sorting, shuffling, reversing and concatenation. Additionally, reading and removing the last elements of a vector happens more frequently in some use cases. For example, stacks pop, peek and push the last element in a vector implementation.

For this reason, the vector head contains a pointer to a leaf node, called a *tail*. Figure 2.3b shows the vector represented in Figure 2.1 with a tail. Whenever an element is inserted at the end of the vector, it is inserted into the tail. If the tail is full (its size equal to M), it is inserted into the actual trie, and the element is inserted into a new leaf node, which replaces the old tail. The same happens when we remove elements: If the tail is empty after a removal, it is replaced with the rightmost leaf node in the trie.

Are there any specific benefits by always keeping the tail nonempty? There is a notable one: The function *last* will always take constant time, as it will always be able to lookup in the tail. Another one, albeit insignificant, is that the average lookup time for a random access operation decreases: There is a higher chance of picking an



(a) Persistent vector pop (b) Persistent vector with tail

Figure 2.3: Persistent vector pop and tail illustration

element in the tail when the tail is guaranteed nonempty. However, if *last* is rarely used, one could consider inserting the tail when it became full: Calculations revolving tail size and tail offset are somewhat easier to reason about, with less special cases. For performance reasons, we will in this thesis assume that a tail P_{tail} will always satisfy $0 < |P_{tail}| \leqslant M$, except when $|P| = 0$, in which $|P_{tail}| = 0$.

At first sight, we may assume that the tail's length has to be stored somewhere to identify whether operations has to work on the tail or on the trie. However, there is no need to keep track of the size of the tail in a leftwise dense persistent vector. As the vector is leftwise dense, only the rightmost leaf node, the tail, is allowed to not be completely filled. This means that all other leaf nodes will contain exactly M elements. The simple expression

$$|P| \mod M$$

will therefore return the size of the tail, or 0 if the tail is completely filled.

To avoid casting 0 back to 32, we can subtract by one, then add one after the modulo:

$$|P_{tail}| = (|P| - 1 \mod M) + 1$$

This formula works fine, with the exception when $|P| = 0$. In that case, the tail's length is $|P_{tail}| = 0$.

Another important value is the tail's *offset index*. This is needed if we want to know whether a certain index refers to a value in the trie or in the tail. It is defined as the size of the vector minus its tail length:

$$\begin{aligned} |P| - |P_{tail}| &= |P| - (|P| - 1 \mod M) - 1 \\ &= (|P| - 1) - ((|P| - 1) \mod M) \end{aligned}$$

For $M = 2^k$, we can use Theorem 2.4 to further simplify the expression:

$$\begin{aligned}(|P|-1)-((|P|-1) \mod M) &= (|P|-1)-((|P|-1)\&M-1) \\ &= (|P|-1)\&\neg(M-1)\end{aligned}$$

where $\neg$ is the binary NOT operation.

Theorem 2.4.

$$a-(a\&b) = a\&\neg b$$

Proof. We know that

$$a-a = a \oplus a = 0$$

where $\oplus$ is the binary XOR operation. Additionally, we can show that

$$a-(a\&b) = a \oplus (a\&b)$$

To see why, realise that a binary subtraction will only cause a carry if and only if the first binary digit is 0, and the second binary digit is 1. This will never happen in $a-a$, as whenever the first binary digit is 0, the second binary digit must also be 0. Performing the operation $a-(a\&b)$ does not change this fact. Consequently, only the binary calculations $1-1$, $1-0$ or $0-0$ can occur in the subtraction $a-(a\&b)$. As $a \oplus \neg a = 1$ and $a \oplus a = 0$, we can therefore replace subtraction with the XOR operation for this formula.

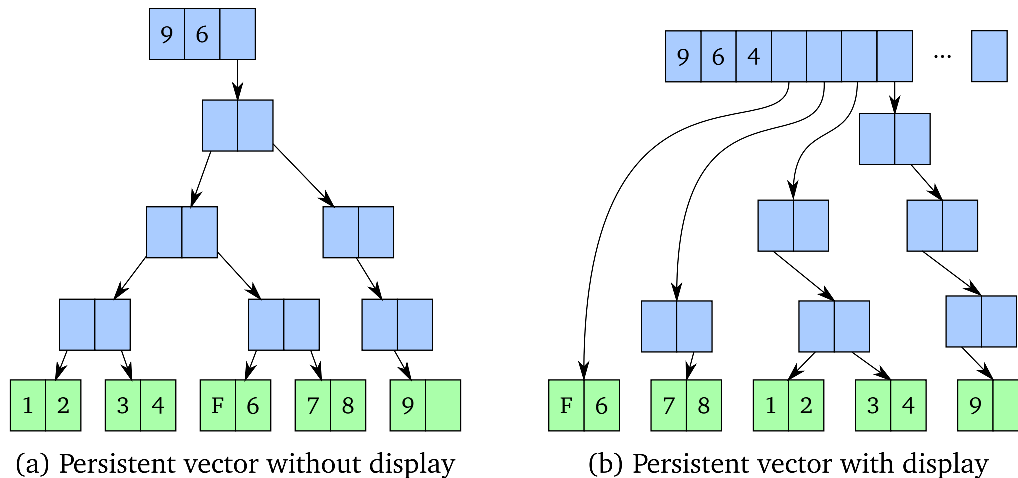
The remaining part can be proven through enumeration, shown in the truth table below.

a	b	$\neg b$	$a\&b$	$a \oplus (a\&b)$	$a\&\neg b$
0	0	1	0	0	0
0	1	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	0

□

One of the good things with the tail is that the previously implemented algorithms does not need to be heavily modified: Updating and indexing only has to check whether the tail is involved, which is done by the tail offset calculation. If it is, we only have to read or update the tail. Reading from the tail is done in constant time, $\Theta(1)$, and writing is linear to the length of the tail, $\mathcal{O}(M)$.

For appending, we check whether the tail is full. If it is, the tail is pushed down in the same fashion we append a single element. Otherwise, the algorithm only has to copy the tail and insert the element. Amortised then, the runtime is



(a) Persistent vector without display (b) Persistent vector with display

Figure 2.4: No display vs. display

$$\Theta\left(\frac{M}{M}\log_M(|P|)+\frac{M(M-1)}{M}\right)=\Theta(\log_M(|P|)$$

as, for every M operations, $M-1$ operations will take $\mathcal{O}(M)$ time, and 1 will take $\Theta(M\log_M|P|)$ time.

Popping works in a similar way: When $1<|P_{tail}|$, we can simply copy and remove the last element from the tail. Otherwise, we have to *promote* the rightmost leaf node in the trie as a new tail, in order to ensure $|P_{tail}|$ is positive. Whereas creating the new path could take $\Omega(1)$ time by the calculations from Section 2.6, we must always walk down to the leaf node to get the new tail. Consequently, the total amortised runtime for popping is the same as appending.

While the tail does not change the worst case asymptotic runtime, the constant factor for these operations on are average considerably decreased for values of M that are reasonably large.

2.8 Display

Scala's immutable vector implementation has taken the concept of a tail and generalised the notion to what is called a *focus*[3]. Instead of only considering the end of a vector, the focus tracks the leaf node which was last updated. The rationale for this generalisation is due to the principle of spatial locality: If an element is either read or written, then it is likely that future operations on the vector will be on elements very close to the element currently read/written[15].

The focus is only modified when the vector itself is modified. As the vector has to be thread-safe, we can either keep the focus with a lock, or only change it during modifications on the vector. Handling a lock per vector is not only time-consuming, but also increases the complexity of a vector implementation.

To further attempt to improve performance, the focus is kept in a *display* – a stack with constant size h_{max}. The display tracks the path taken down to the focus, allowing for, on average, efficient lookup in leaf nodes near the focus: Given two leaf nodes, we can efficiently calculate where the paths to walk first split up, and go directly to their lowest common ancestor within the display. To avoid semantic garbage, the pointers from the topmost nodes down through the path is replaced with NIL.

Whenever the path down to the focus is changing, the old path from the lowest common ancestor down to the leaf node has to be *committed* back to the ancestor. The previously written NIL at level n is replaced with the old display node at level $n - 1$, all up to the lowest common ancestor. In the worst case, $h(P)$ nodes may be copied at this stage.

Figure 2.4 presents the vector `[1, 2, 3, 4, F, 6, 7, 8, 9]`, where F at index 4 was the last value to be changed. As $h_{max} = 16$ when $M = 2$, the display slots after the fourth level are presented as "..." for space reasons. Note that all nodes in the display lacks a single pointer: The missing pointer is the lower node contained in the display, as previously mentioned.

```
1  function UPDATE(P, i, val)
2      P′ ← CLONE(P)                          ▷ Clones the display D as well
3      j ← 0
4      while ⌊i_old / M^(j+1)⌋ ≠ ⌊i / M^(j+1)⌋ do
5          p ← ⌊i_old / M^(j+1)⌋ mod M
6          D_(j+1)[p] ← CLONE(D_j)
7          j ← j + 1
8      end while
9      for h ← j downto 1 do
10         i′ ← ⌊i / M^h⌋ mod M
11         D_h ← CLONE(D_h)
12         D′ ← D_h[i′]
13         D_h[i′] ← NIL
14         D_(h−1) ← D′
15     end for
16     i′ ← i mod M
17     D_0 ← CLONE(D_0)
18     D_0[i′] ← val
19     return P′
20 end function
```

Listing 2.6: Implementation of UPDATE, using a display.

Listing 2.6 presents the update algorithm when the persistent vector uses a display. The variable D_i refers to the ith level of the display in the newly copied persistent vector head P', where $i = 0$ is the leaf level.

Finding the common ancestor is done by retaining the old index i_{old} which was updated. As the subindex from each step in the radix search algorithm is independent from the other steps, finding out whether the current parent T is a common ancestor is easy: We just have to check that the first n steps are similar. This can be done by checking

$$\left\lfloor \frac{i_{old}}{M^{h(T)+1}} \right\rfloor = \left\lfloor \frac{i}{M^{h(T)+1}} \right\rfloor$$

if these are not equivalent, we move up a level and continue the process.

How well does a display fare in contrast to a persistent vector with just a tail? For truly uniformly random updates, a non-display version fares better, as a display would in the worst case have to commit all but one node, then perform the equivalent of a path copy down to the node to update. However, a display is better when many updates happen in localised clusters. There is, of course, no problem to have both options available and a function to move from one representation to another. However, neither Clojure nor Scala have an option to switch between representations.

Another convenient use of a "display" is through iteration over the whole or parts of the vector: We can consider the display as a stack of nodes, where the top of the stack contains the current leaf node we iterate over, the second element is the leaf node's parent, and so on. Not only does the running time of such a solution take $\Theta(|P|)$ proven by Theorem 2.5, but as we do not have to perform any commits – as nothing is modified – there is no need for memory allocations or copying after the "display" has been created.

Theorem 2.5. *Iterating over a persistent vector* P *takes* $\Theta(|P|)$ *time.*

Proof. Walking down to the first element takes $\Theta(\log_M |P|)$ time. Amortised over $|P|$ elements, this takes $\Theta((\log_M |P|)/|P|) \in \mathcal{O}(1)$ time per element.

By using a display, an iteration step takes constant time if the next iteration element is within same leaf node. However, for every M iteration steps, we have to change leaf node, for every M^2 iteration steps, we have to change second level node, and so on. On average, then, each iteration step require us to change

$$\frac{1}{M} + \frac{1}{M^2} + \ldots + \frac{1}{M^{h(P)+1}} = \sum_{i=1}^{h(P)+1} \frac{1}{M^i} = \sum_{i=0}^{h(P)} \frac{1}{M^{i+1}}$$

blocks in the display. This is a finite geometric series, which expand to

$$\begin{aligned}
\sum_{i=0}^{h(P)} \frac{1}{M^{i+1}} &= \frac{M^{h(P)-1}\left(M^{-h(P)+1} - 1\right)}{M - 1} \\
&= \frac{M^{-\lceil \log_M(|P|) \rceil}\left(M^{\lceil \log_M(|P|) \rceil} - 1\right)}{M - 1} \\
&= \frac{1 - M^{-\lceil \log_M(|P|) \rceil}}{M - 1}
\end{aligned}$$

Assuming $M > 1$,

$$\lim_{|P| \to \infty} \frac{1 - M^{-\lceil \log_M(|P|) \rceil}}{M - 1} = \frac{1}{M - 1}$$

which tells us that the worst case amortised amount of node replacements in the display per iteration step is $1/(M-1)$, a constant. Consequently, all iteration steps take on average constant time, and a total iteration therefore takes $\Theta(|P|)$ time. □

Corollary 2.6. *A single iteration step on a persistent vector takes* $\Theta(1)$ *amortised time.*

2.9 Performance

In the previous sections, we have found time complexities for all operations with respect to both M and |P|. However, M is a constant which does not change during program execution. As a result, one might assume that the asymptotic runtimes derived could be simplified to not contain M. While this is valid from a theoretical standpoint, modern computer architectures change the performance characteristics significantly for different values of M. As a result, the runtimes without respect to M reflect real world performance relatively poorly. We will look at this aspect of the persistent vector in this section.

Perhaps the most intuitive difference is copying. Section 2.4 claims that copying a single trie node T takes $\mathcal{O}(M)$ time: $\Theta(n)$ when $n = \|T\|$. For large values of M, this is true. However, for small values of M, in which a trie node T with size $\|T\| = M$ fits within a single cache line, copying would take roughly the same time regardless of its size. Consequently, we can claim that copying takes $\Theta(1)$ when $Mp + o <$ *cache line size*, where p is the size of a pointer, and o is the additional overhead in a trie node. This also assumes leaf nodes contain elements with a size less than or equal to a pointer.

The result of this claim is that tail-less versions of update and append now take $\Theta(\log_M |P|)$ instead of $\Theta(M \log_M |P|)$, similarly for with pop but with $\mathcal{O}$ instead of Θ. Note that, whereas this time complexity is equal to indexing, memory access factors make indexing considerably faster. Versions of append and pop with a tail has an amortised time

$$\Theta\left(\frac{1}{M} \log_M(|P|) + \frac{M-1}{M}\right) = \Theta\left(\frac{1}{M} \log_M |P|\right)$$

compared to the original amortised time $\Theta(\log_M |P|)$.

While assuming copying is fast for relatively small M, a larger M gives better performance for an operation proportional to $\log_M$. Figure 2.5 presents the height of a vector with different M, showing that M is not negligible in practise: On average, $M = 2$ gives a tree roughly 5 times as high compared to $M = 32$. Preferably, M should be as large possible while still give the benefits of keeping nodes within a small number of cache lines.

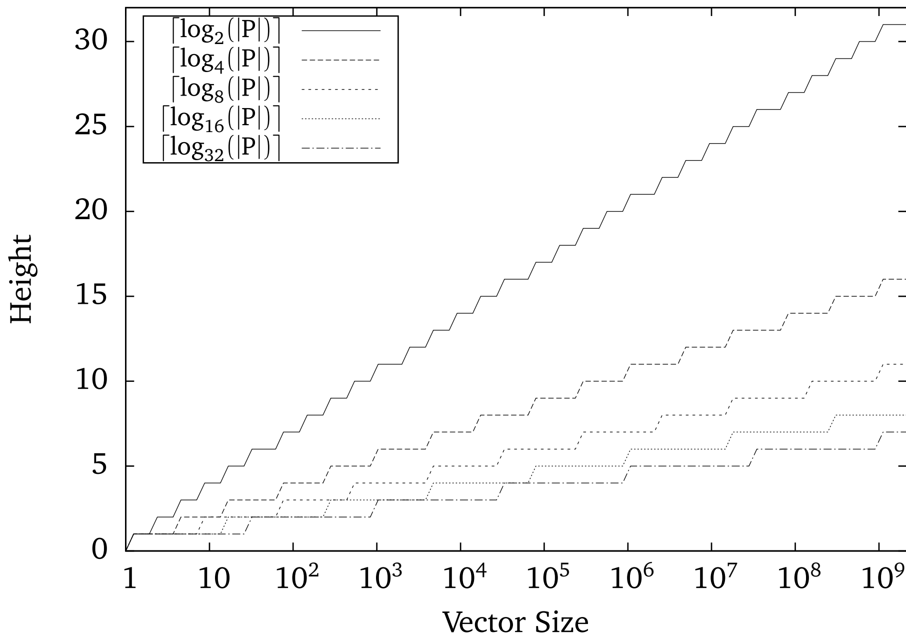


Figure 2.5: Plot of $h(P)$ for different M.

Another consideration when thinking about performance is the overhead and overhead ratio. It is obvious that a smaller overhead, and consequently a smaller overhead ratio, is preferable. It is also somewhat intuitive that a higher branching factor leads to less overhead: In the extreme case, a persistent vector would degenerate down to an immutable array, which has minimal overhead. However, the exact ratio and how much there is to gain by more branching is not evident. Therefore, we will now explore the memory overhead of a persistent vector and find its overhead ratio.

Definition 2.6. *The* ***overhead*** *of a vector or trie* T *is the memory used on the whole trie* T *that is not used to store an element:*

$$mem_{oh}(T) = mem_{tot}(T) - |T| \times |\tau|$$

Definition 2.7. *The* ***overhead ratio*** *for a persistent vector* P *is*

$$\frac{mem_{oh}(P)}{mem_{tot}(P)}$$

Theorem 2.6. *In a fully dense trie* T*, the total amount of internal nodes is*

$$\frac{M^{h(T)} - 1}{M - 1}$$

Proof. In a fully dense trie, there are $M^{h(T)+1}$ elements which are stored within $M^{h(T)}$ leaf nodes. All the leaf nodes are stored within $M^{h(T)-1}$ nodes at the second level,

which are stored within $M^{h(T)-2}$ nodes at the third level and so on. As the trie has a total of $h(T)$ levels if we exclude the lowest level, we get

$$\sum_{i=1}^{h(T)} M^{h(T)-i} = \sum_{i=0}^{h(T)-1} M^{i}$$

This is the well known finite geometric series, which expands to

$$\frac{M^{h(T)} - 1}{M - 1}$$

□

Theorem 2.7. *The **overhead** of a fully dense trie* T *is*

$$mem_{oh}(T) = \frac{p(|T| - M) + o(|T| - 1)}{M - 1}$$

Proof. We assume the constant overhead per node, denoted o, is equivalent for both internal and leaf nodes.

The size of a fully dense trie T is $|T|$. All of these values will be inside leaf nodes, which all contain M elements. A single leaf node has o overhead.

There are $(M^{h(T)} - 1)/(M - 1)$ internal nodes. All of the internal nodes contain M pointers of size p, and an additional overhead of o. We can therefore derive the formula above:

$$\begin{aligned}
mem_{oh}(T) &= \frac{o \times |T|}{M} + \frac{M^{h(T)} - 1}{M - 1} \times (Mp + o) \\
&= \frac{o \times |T|}{M} + \frac{M^{h(T)+1} - M}{M(M - 1)} \times (Mp + o) \\
&= \frac{o \times |T|}{M} + \frac{|T| - M}{M(M - 1)} \times (Mp + o) && \text{by Corollary 2.1} \\
&= \frac{o \times |T|\,(M - 1)}{M(M - 1)} + \frac{|T| - M}{M(M - 1)} \times (Mp + o) \\
&= \frac{o \times |T|\,(M - 1) + (|T| - M)(Mp + o)}{M(M - 1)} \\
&= \frac{o\,|T|\,M - o\,|T| + |T|\,Mp + o\,|T| - M^2p - oM}{M(M - 1)} \\
&= \frac{o\,|T| + |T|\,p - Mp - o}{M - 1} \\
&= \frac{p(|T| - M) + o(|T| - 1)}{M - 1}
\end{aligned}$$

□

Theorem 2.8. *The **overhead** of a leftwise dense trie* T *is*

$$mem_{oh}(T) = o + (p + o) \sum_{i=1}^{h(T)} \left\lceil \frac{|T|}{M^{i}} \right\rceil$$

Proof. In a lefwise dense persistent trie, all of the nodes will, with the exception of the topmost node, have a pointer towards it. Consequently, every node minus one contributes to $o + p$ overhead, and a single node contributes to o overhead.

A trie will contain $\lceil |T| / M^{i+1} \rceil$ nodes at level i, where $i = 0$ is the lowest level. As the height of a trie is $h(T)$, we have that

$$\begin{aligned}
mem_{oh}(T) &= (p+o)\left(\sum_{i=1}^{h(T)+1}\left\lceil\frac{|T|}{M^i}\right\rceil\right) - p \\
&= (p+o)\left(\left\lceil\frac{|T|}{M^{h(T)+1}}\right\rceil + \sum_{i=1}^{h(T)}\left\lceil\frac{|T|}{M^i}\right\rceil\right) - p \\
&= (p+o)\left(\left\lceil\frac{|T|}{M^{\lceil \log_M(|T|)\rceil}}\right\rceil + \sum_{i=1}^{h(T)}\left\lceil\frac{|T|}{M^i}\right\rceil\right) - p \qquad \text{by Theorem 2.2} \\
&= (p+o)\left(1 + \sum_{i=1}^{h(T)}\left\lceil\frac{|T|}{M^i}\right\rceil\right) - p \qquad \text{Assuming } |T| > 0 \\
&= o + (p+o)\sum_{i=1}^{h(T)}\left\lceil\frac{|T|}{M^i}\right\rceil
\end{aligned}$$

For $|T| = 0$, we have exactly one empty leaf node, so $mem_{oh}(T) = o$ if $|T| = 0$. The equation derived above therefore also holds for $|T| = 0$. □

Theorem 2.9. *For sufficiently large tries, the overhead for a leftwise dense trie* T *is approximated by the overhead for a fully dense trie:*

$$o + (p+o)\sum_{i=1}^{\lceil \log_M(|T|)\rceil - 1}\left\lceil\frac{|T|}{M^i}\right\rceil \approx \frac{p(|T| - M) + o(|T| - 1)}{M - 1}$$

Proof. Let

$$\begin{aligned}
a &= o + (p+o)\sum_{i=1}^{\lceil \log_M(|T|)\rceil - 1}\left\lceil\frac{|T|}{M^i}\right\rceil \\
\tilde{a} &= \frac{p(|T| - M) + o(|T| - 1)}{M - 1}
\end{aligned}$$

We then have that the absolute error is

$$\epsilon = \tilde{a} - a$$

and

$$\begin{aligned}
a &= o + (p+o) \sum_{i=1}^{\lceil \log_M(|T|) \rceil - 1} \left\lceil \frac{|T|}{M^i} \right\rceil \\
&\leqslant o + (p+o) \sum_{i=1}^{\lceil \log_M(|T|) \rceil - 1} \left\lfloor \frac{|T|}{M^i} \right\rfloor + 1 \\
&= o + (p+o) \left(\lceil \log_M(|T|) \rceil - 1 + \sum_{i=1}^{\infty} \left\lfloor \frac{|T|}{M^i} \right\rfloor \right) \\
&< o + (p+o) \left(\lceil \log_M(|T|) \rceil - 1 + \sum_{i=1}^{\infty} \frac{|T|}{M^i} \right) \\
&= o + (p+o) \left(\lceil \log_M(|T|) \rceil - 1 + \frac{|T|}{M-1} \right) \\
&= (p+o) \lceil \log_M(|T|) \rceil + \frac{|T|(p+o)}{M-1} - p
\end{aligned}$$

consequently

$$\begin{aligned}
\tilde{a} - a &< \frac{p(|T| - M) + o(|T| - 1)}{M-1} - \left((p+o) \lceil \log_M(|T|) \rceil + \frac{|T|(p+o)}{M-1} - p \right) \\
&= \frac{p(|T| - M) + o(|T| - 1) - |T|(p+o)}{M-1} - (p+o) \lceil \log_M(|T|) \rceil + p \\
&= \frac{-pM - o}{M-1} - (p+o) \lceil \log_M(|T|) \rceil + p \\
&= \frac{-o - p}{M-1} - (p+o) \lceil \log_M(|T|) \rceil
\end{aligned}$$

As $|\epsilon|$ with respect to $|T|$ grows much slower than both a and $\tilde{a}$, $a \approx \tilde{a}$ for sufficiently large vectors. □

Corollary 2.7. *The overhead ratio for a leftwise dense trie* T *with branching factor* M *is approximately*

$$\frac{p(M - |T|) - o(|T| - 1)}{M(p - |\tau| \times |T|) - o(|T| - 1) + |T|(|\tau| - p)}$$

Proof. The overhead can be approximated by Theorem 2.9. Therefore,

$$\frac{mem_{oh}(T)}{mem_{oh}(T) + |T| \times |\tau|} \approx \frac{(p(|T| - M) + o(|T| - 1))/(M-1)}{(p(|T| - M) + o(|T| - 1))/(M-1) + |T| \times |\tau|}$$

The corollary can be derived from this expression through algebraic manipulation. □

Corollary 2.8. *The overhead ratio for any sufficiently large persistent vector* P *with branching factor* M *is approximately*

$$\frac{o + p}{o + p + (M-1)|\tau|}$$

Proof. The overhead ratio of a persistent vector is approximately equal to the overhead ratio of its trie T.

$$\begin{aligned}\lim_{|P|\to\infty}\frac{mem_{oh}(P)}{mem_{tot}(P)} &= \lim_{|P|\to\infty}\frac{p(M-|P|)-o(|P|-1)}{M(p-|\tau|\times|P|)-o(|P|-1)+|P|\,(|\tau|-p)}\\ &= \lim_{|P|\to\infty}\frac{f(|P|)}{g(|P|)}\end{aligned}$$

where

$$\begin{aligned}f(|P|) &= p(M-|P|)-o(|P|-1) = pM+o-|P|\,(o+p)\\ g(|P|) &= M(p-|\tau|\times|P|)-o(|P|-1)+|P|\,(|\tau|-p)\\ &= |P|\,(|\tau|-p-M\,|\tau|-o)+Mp+o\\ &= -\,|P|\,(o+p+(M-1)\,|\tau|)+Mp+o\end{aligned}$$

The derivatives of f and g with respect to $|P|$ are

$$\begin{aligned}f'(|P|) &= \frac{d(pM+o-|P|\,(o+p))}{d\,|P|}\\ &= -(o+p)\end{aligned}$$

and

$$\begin{aligned}g'(|P|) &= \frac{d(-\,|P|\,(o+p+(M-1)\,|\tau|)+Mp+o)}{d\,|P|}\\ &= -(o+p+(M-1)\,|\tau|)\end{aligned}$$

Assuming $o+p<0$ and $o+p+(M-1)\,|\tau|<0$, then

$$\lim_{|P|\to\infty} f(|P|) = \lim_{|P|\to\infty} g(|P|) = -\infty$$

and $g'(|P|) \neq 0$ for all $|P|$. We can therefore use L'Hôpital's rule, and get that

$$\begin{aligned}\lim_{|P|\to\infty}\frac{f(|P|)}{g(|P|)} &\overset{H}{=} \lim_{|P|\to\infty}\frac{f'(|P|)}{g'(|P|)}\\ &= \frac{-(o+p)}{-(o+p+(M-1)\,|\tau|)}\\ &= \frac{o+p}{o+p+(M-1)\,|\tau|}\end{aligned}$$

□

None of these calculations includes the size of unused additional pointers in the rightmost trie nodes, but that is okay. The highest possible amount of unused pointers will be $M-1$ for a single node (except the root node), which there can at most be

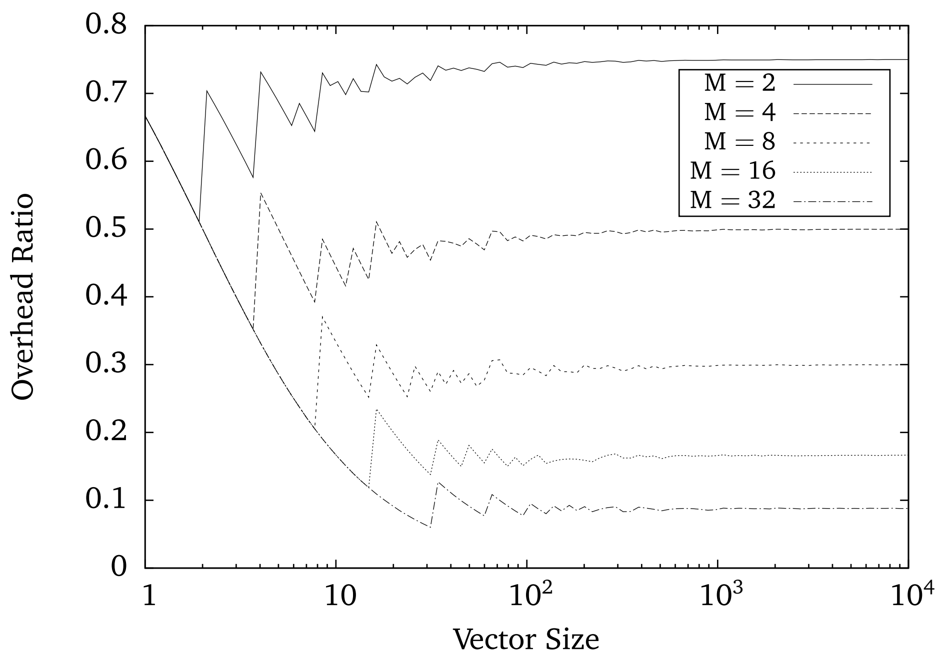


Figure 2.6: Overhead ratio for different M.

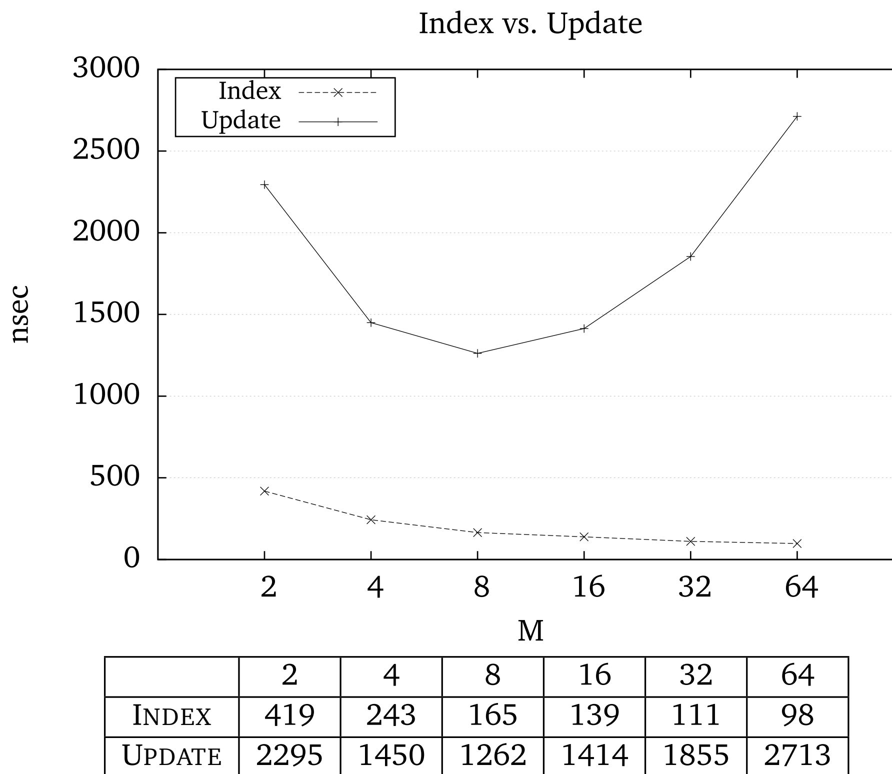
$h(P)+1$ of. As such, the additional overhead may at worst be $< p \times (h(P)+1)(M-1)$. This term grows logarithmically, which is much slower than the total overhead and therefore negligible.

Figure 2.6 presents the *actual* overhead ratio of a persistent vector with varying branching factors, where $o = 16$ and $p = |\tau| = 8$. From Corollary 2.8, we expect the overhead ratio to stabilise around 0.75 for $M = 2$, and around 0.088 for $M = 32$, which it does.

While overhead calculations are important for memory, it is also important for time: Less overhead means that the cache can contain more of the vector simultaneously, and is less likely to swap it out with other data, regardless of whether the data is contiguous or not.

If the data is not contiguous, having trie nodes smaller than the size of a memory cache line would further reduce the possibility to have the entire vector in the cache, and increase the probability of cache misses. However, how big this problem is in reality is hard to estimate: It depends on the garbage collector used, and when the data is allocated on the heap [16].

All in all, using a higher branching factor gives multiple benefits for performance related both to space and time. [3] briefly explores the time performance for different M, and is presented in Figure 2.7. The measurements were done in the Scala programming language, performed on a persistent vector containing 500 000 elements. Assuming updates happen considerably less frequently than indexing, $M = 32$ strikes



	2	4	8	16	32	64
INDEX	419	243	165	139	111	98
UPDATE	2295	1450	1262	1414	1855	2713

Figure 2.7: Persistent vector indexing and update times, from [3].

as a good fit. For this reason, we can consider many of the persistent vector operations to run in effectively constant time:

Definition 2.8. *An algorithm* A *whose runtime depends on* n *is running in* effectively constant time, $\mathcal{O}(\sim 1)$, *if the effective running time on modern computers does not change significantly when* n *changes significantly.*

The definition of effectively constant time is intentionally vague: It should only be considered as a way of conveying that an algorithm is fast and that its runtime can be considered constant with current modern computer architectures. Of course, at some point, the theoretical factors turn noticable. However, for users without very heavy performance requirements, the actual runtime will be indistinguishable from constant time.

Table 2.1 presents all the operations on a persistent vector with a tail, with its theoretical and effective runtimes. As seen, there are very few actual constant time operations, although there are considerably many which runs in effectively constant time. Many of these are commonly done on arrays, the persistent vector is thus an ideal candidate as a functional equivalent to a dynamically growing array.

NAME	OPERATION	ACTUAL RUNTIME	EFF. RUNTIME
Lookup	A_i	$\mathcal{O}(\log_M n)$	$\mathcal{O}(\sim 1)$
Last	*last*(A)	$\Theta(1)$	$\Theta(1)$
Count	$\|A\|$	$\Theta(1)$	$\Theta(1)$
Update	$A_i \leftarrow x$	$\mathcal{O}(M \log_M n)$	$\mathcal{O}(\sim 1)$
Append	*conj*(A, x)	$\mathcal{O}(M \log_M n)$	$\mathcal{O}(\sim 1)$
Pop	*pop*(A, x)	$\mathcal{O}(M \log_M n)$	$\mathcal{O}(\sim 1)$
Iteration	for x in A:	$\Theta(n)$	$\Theta(n)$
Concat	$A \mathbin{+\!\!+} B$	$\mathcal{O}(M \log_M(n)\, \|B\|)$	$\mathcal{O}(\sim \|B\|)$
Right slice	$A_{(0,i)}$	$\mathcal{O}(M \log_M n)$	$\mathcal{O}(\sim 1)$
Left slice	$A_{(i,\|A\|)}$	$\mathcal{O}(M \log_M(n) \times n)$	$\mathcal{O}(\sim n)$
Slice	$A_{(i,j)}$	$\mathcal{O}(M \log_M(n) \times n)$	$\mathcal{O}(\sim n)$

Table 2.1: Time complexity of persistent vector operations, where $n = |A|$.

Chapter 3

RRB-Trees

The persistent vector excels at many operations, but is not good at general-purpose slicing and concatenation, which is sometimes essential for parallel processing. The data structures ropes and finger trees do enable fast concatenation and slicing [17, 18]. However, the index time of a rope is $\mathcal{O}(\log(s + c))$, in which s is the number of splits, and c is the number of concatenations. In addition, splits hold onto the original nodes, creating potential semantic garbage. The finger tree avoids these issues, however its index times can not be considered effectively constant. The RRB-tree attempts to enable efficient concatenation while retaining efficient indexing, by extending the persistent vector [3].

Much of the work in this chapter is similar to that of Bagwell and Rompf, but approached from a different view: We define the RRB-tree differently but semantically equivalent, in what we believe is an easier to comprehend definition. Additionally, we provide formal proofs, and algorithms which should ease the implementation for practitioners.

3.1 Introduction

Recall that a persistent vector is leftways dense: All its internal nodes will contain M slot entries, with the possible exception of the rightmost nodes. The initial idea behind RRB-trees is to relax its requirement of being dense, such that the branching factor for any node can be $M_l \leqslant \|T\| \leqslant M_m$ without breaking the density property:

Definition 3.1. *Let* $d^R_{full}(T)$ *denote that the trie* T *is* fully relaxed dense. *An* M_m*-way branching trie* T *containing type* τ *is* fully relaxed dense *if and only if* $0 < M_l \leqslant \|T\| \leqslant M_m$ *and*

$$\begin{cases} \bigwedge_{i=0}^{\|T\|-1} T_i : \tau & \textit{if } h(T) = 0 \\ \bigwedge_{i=0}^{\|T\|-1} d^R_{full}(T_i) & \textit{if } h(T) \neq 0 \end{cases}$$

Definition 3.2. *Let* $d^R_{left}(T)$ *denote that the trie* T *is* leftwise relaxed dense. *An* M_m-*way branching trie* T *containing type* τ *is* leftwise relaxed dense *if and only if* $0 < \|T\| \leqslant M_m$ *and*

$$\begin{cases} \bigwedge_{i=0}^{\|T\|} T_i : \tau & \text{if } h(T) = 0 \\ \left(\bigwedge_{i=0}^{\|T\|-1} d^R_{full}(T_i) \right) \wedge d^R_{left}(T_{\|T\|-1}) & \text{if } h(T) \neq 0 \end{cases}$$

If an RRB-tree R *is leftwise relaxed dense, its root trie* R_{root} *must be leftwise dense, and either* $h(R) = 0$ *or* $1 < \|R_{root}\|$ *must be satisfied.*

From the theorems related to the persistent vector, we can quickly create a good portion of corollaries:

Corollary 3.1. *If* $d^R_{full}(T)$*, then* $M_l^{h(T)+1} \leqslant |T| \leqslant M_m^{h(T)+1}$.

Proof. In the most dense case, all nodes contain M_m elements. If so, $d_{full}(T)$ will be satisfied, assuming $M = M_m$. This implies $|T| = M_m^{h(T)+1}$ by Theorem 2.1.

In the least dense case, all nodes contain M_l elements. If so, $d_{full}(T)$ will be satisfied, assuming $M = M_l$. This implies $|T| = M_l^{h(T)+1}$ by Theorem 2.1.

It then follows that $M_l^{h(T)+1} \leqslant |T| \leqslant M_m^{h(T)+1}$. □

Corollary 3.2. *If* $d^R_{full}(T)$*, then*

$$\log_{M_m}(|T|) - 1 \leqslant h(T) \leqslant \log_{M_l}(|T|) - 1$$

Proof. This follows from the same logarithmic definition used in Corollary 2.1. □

Corollary 3.3. *Given a* leftwise relaxed dense *RRB-tree* R*,*

$$h_m(R) = \lceil \log_{M_m}(|R|) \rceil - 1$$
$$h_l(R) = \lceil \log_{M_l}(|R|) \rceil - 1$$

and

$$\begin{cases} h(R) = 0 & \text{if } |R| \leqslant M_l \\ h_m(R) \leqslant h(R) \leqslant h_l(R) & \text{if } M_l < |R| \end{cases}$$

Corollary 3.4. $h_l(R) = \lceil \log_{M_l}(|P|) \rceil - 1 \in \Theta(\log_{M_l}(|R|))$

Assuming we can use the algorithms described in Chapter 2, an RRB-tree with minimal branching factor M_l and maximal branching factor M_m will, in the worst case, have the same asymptotic runtimes as an M_l-way branching persistent vector.

However, the RRB-tree is relaxed even further: The *search step* relaxed RRB-tree allows concatenations to be even more efficient by allowing some, but not all nodes to be even smaller. It does so by considering the total amount of children contained over a set of children: If a node has many siblings which are filled, then this node may have very few children, provided the *average* number of children is above some threshold.

In order to comprehend the definition of the search step relaxation, we define T_{opt} through the definition of redistribution:

Definition 3.3. *An* expansion *of a trie* T *is the ordered list of its children:*

$$\langle c_0, c_1, \ldots, c_{\|T\|-1} \rangle$$

An expansion of an ordered list of tries of same height $\langle T_0, T_1, \ldots T_n \rangle$ *is the concatenated, ordered list of their expansions*

$$\langle T_0[0], T_0[1], \ldots T_0[\|T_0\| - 1], T_1[0], \ldots T_n[\|T_n\| - 1] \rangle$$

Definition 3.4. *A* redistribution *of an ordered list of tries of same height*

$$L = \langle L_0, L_1, \ldots L_{\|L\|-1} \rangle$$

is the ordered list

$$L' = \langle L'_0, L'_1, \ldots L'_{\|L'\|-1} \rangle$$

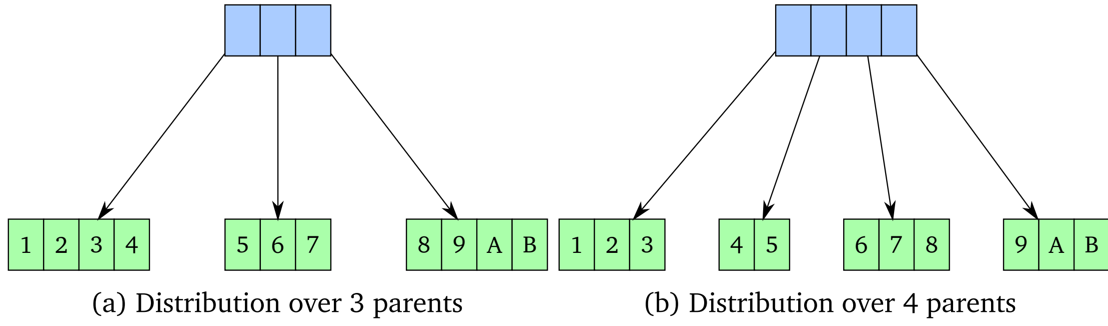
where the expansions of L *and* L' *are equal.*

The redistribution of the trie T *is* T'*, where* L *is the expansion of* T*,* L' *is the expansion of* T'*, and* L' *is a redistribution of* L*.*

A redistribution of a list of nodes can be seen as a way to contain the same children in same order, but where some parents contain different children by having different lengths. Note that a redistribution of a list of tries does not always need to be the same length as the original: It is sometimes possible to distribute the children over less or more parent nodes, while stil satisfying the density constraints.

Figure 3.1 contains an example of such a redistribution: Both Figure 3.1a and 3.1b are redistributions of each other. The children could be both leaf nodes and internal nodes, the only reason for the labels is to show that they are in the same order.

Now we can finally define T_{opt}:



(a) Distribution over 3 parents (b) Distribution over 4 parents

Figure 3.1: Redistribution of nodes.

Definition 3.5. T_{opt} *is some redistribution of* T*,* $h(T) > 0$*, such that for all redistributions* T' *of* T*,* $\|T_{opt}\| \leqslant \|T'\|$.

In other words, T_{opt} is some redistribution of T that gives the smallest length of T_{opt}. Note that there may be multiple T_{opt}, in which case we pick any one of them.

If we go back to Figure 3.1, we see that Figure 3.1a is a T_{opt} assuming $M = 4$: We are unable to store 11 children in less than 3 parents. We easily see that Figure 3.1b is not a T_{opt}, because it stores all the children in 4 parents, instead of the optimal 3.

Theorem 3.1. *For all* T_{opt}

$$\|T_{opt}\| = \left\lceil \frac{\sum_{i=0}^{\|T\|-1} \|T_i\|}{M} \right\rceil$$

Proof. Let S be the sum of all the children of every T_i:

$$S = \sum_{i=0}^{\|T\|-1} \|T_i\|$$

One way to store all S children in the minimum amount of parent nodes is by filling up the first one with M of the remaining children, then repeat for all the following nodes until we have no children left. Using this technique, we will have

$$\left\lfloor \frac{S}{M} \right\rfloor$$

nodes with exactly M children, and a potential last node with $< M$ children. If we need the last node, then S/M has a nonzero remainder, and

$$\left\lceil \frac{S}{M} \right\rceil$$

therefore returns the minimal amount of nodes required. □

Using Theorem 3.1 we can now measure the density of a set of nodes, instead of measuring the density of a single node:

Definition 3.6. *Let* $d_R(T)$ *denote that the trie* T *is* search step relaxed. *An* M*-way branching trie* T *containing type* τ *is* search step relaxed *if and only if either*

1. $h(T) = 0$, $\|T\| > 0$, *and all its table entries are of type* τ*, or*
2. $h(T) > 0$,

$$\|T\| \leqslant \|T_{opt}\| + e_{max}$$

and

$$\bigwedge_{i=0}^{\|T\|-1} d^R(T_i)$$

for some predefined e_{max}

If an RRB-tree R *is search step relaxed, its root trie* R_{root} *must be search step relaxed, and either* $h(P) = 0$ *or* $1 < \|P_{root}\|$ *must be satisfied.*

Curiously, a search step relaxed tree does not require a definition of fully dense or leftwise dense. As such, the height relation is harder to convey, and the results might initially be somewhat surprising.

Theorem 3.2. *The height* $h(R)$ *of an* M*-way branching* search step relaxed *RRB-tree* R *is*

$$h_m(R) \leqslant h(R) \leqslant \infty$$

where

$$h_m(R) = \lceil \log_M(|P|) \rceil - 1$$

Proof. The minimal height of an RRB-tree is when the RRB-tree is leftwise dense. In that case, it has the height of a leftwise dense persistent vector, $h_m(R)$.

Assume $e_{max} > 0$. In the worst case, the top trie T has 0 nodes that are fully dense, and 2 nodes that contain a single subtrie. The 2 nodes can contain a single child without breaking the invariant. These 2 nodes can again contain a single child, and so forth. This implies that the upper height is unbound if we only use this invariant. □

As shown in the proof for Theorem 3.2, from a theoretical standpoint, the height of an RRB-tree can be infinitely high. Clearly, the actual height of an RRB-tree is considerably lower than infinity: In fact, assuming we never break the invariant and use “sensible” operations on the RRB-tree, the tree can be considered to have the height of a leftwise relaxed dense RRB-tree for some M_l. Theorem 3.4 proves this formally after the concatenation algorithm is properly explained.

The RRB-tree uses the same ideas as the persistent vector uses for branching. Consequently all the same bitwise optimisations presented in Section 2.3 can be done for all algorithms presented in this chapter, where values of M are equal to some power of two higher than one.

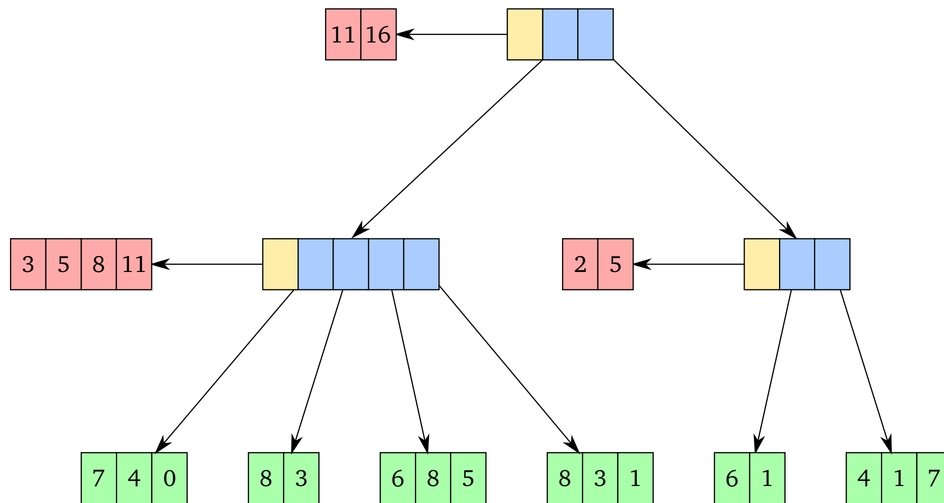


Figure 3.2: Trie containing size tables.

3.2 Relaxed Radix Access

As shown in previous section, a relaxed radix balanced tree does not necessarily have to be fully dense. A result of that is that it is not possible to use radix search in general to find the element at a given index.

In order to implement some sort of index access algorithm, we have to extend tries with a *size table*. The size table of a trie T contains $\|T\|$ integers, where the ith entry is the cumulative sum of the sizes of all children in T up to and including the ith table entry. The size table is denoted $\sigma(T)$, and the ith size table entry is denoted $\sigma_i(T)$. If $\sigma(T) = \text{NIL}$, then $d_{left}(T)$ and the index can be computed through the radix access algorithm presented in Section 2.3. As leaf nodes are by definition leftwise dense, they will never have a size table.

Implementation wise, a size table is not usually stored within the trie node. It is instead a pointer to a fixed array of integers for two reasons: First, if $\sigma(T) = \text{NIL}$, then storing the values in the node itself will waste space. Second, when performing an UPDATE operation, no size table has to be modified. Copying the sizes require more time and space, which is wasteful.

Figure 3.2 shows a subtrie where all nodes, with the exception of the leaf nodes, have size tables. The size table of an internal node is the node immediately left of it. An internal node without a size table will be represented as a node where the arrow extending directly left points to nothing.

We can find the correct table entry to walk by performing a search over the size table: As the size table contains the cumulative sizes of their children, the correct one to walk is the first entry with a cumulative sum strictly higher than the index of the element we want to lookup.

An initial attempt may use binary search to find the correct node to traverse. However, measurements shows that a binary search is is slower than an improved version of a linear search, the *relaxed radix search* [3]:

Assuming we use a normal linear search, we can improve it as we know that we can skip a certain number of slots. Recall that Section 2.3 states that the index to walk in a leftways dense trie T is $i' = \lfloor \frac{i}{M^{h(T)}} \rfloor \bmod M$. Can the index to walk in a relaxed dense trie T be less than i'? Clearly not: The most dense trie T will be fully dense, in which case i' will be the correct index to walk. For less dense tries, the index may be higher, but never lower. As such, we can start the linear search by inspecting $\sigma_{i'}(T)$, then continue onwards.

```
1  function RRB-LOOKUP(R, i)
2      T ← R_root
3      d ← M^h(T)
4      for h ← h(T) downto 0 do
5          i' ← ⌊i/d⌋ mod M
6          if σ(T) ≠ NIL then
7              while σ_i'(T) ⩽ i do
8                  i' ← i' + 1
9              end while
10              if i' ≠ 0 then
11                  i ← i − σ_(i'−1)(T)
12              end if
13          end if
14          T ← T_i'
15          d ← d/M
16      end for
17      return T                                  ▷ T is element here
18  end function
```

Listing 3.1: Relaxed radix search.

Listing 3.1 presents the relaxed radix search algorithm. Notice that we must subtract the additional cumulative size from the subtries in front of us to get the index at next level. Otherwise, the size tables would have to be aware of the sizes of tries at their left side, which would cause concatenation and slices to be linear of the size of the RRB-tree.

With this improved linear search, what is the maximal amount of steps in the linear search per node? Assume the current node T is fully relaxed dense, and we want to look up the element at index $M_l^{h(T)+1} - 1$, possibly the last element. We expect to walk index

$$i' = \left\lfloor \frac{M_l^{h(T)+1} - 1}{M_m^{h(T)}} \right\rfloor = \left\lceil \frac{M_l^{h(T)+1}}{M_m^{h(T)}} \right\rceil - 1$$

We know that in the worst case, we will walk index $M_m - 1$, consequently the maximal extra search steps will at most be

$$i_\delta = M_m - 1 - \left\lfloor \frac{M_l^{h(T)+1} - 1}{M_m^{h(T)}} \right\rfloor = M_m - \left\lceil \frac{M_l^{h(T)+1}}{M_m^{h(T)}} \right\rceil = \left\lceil \frac{M_m^{h(T)+1} - M_l^{h(T)+1}}{M_m^{h(T)}} \right\rceil$$

For $M_m = 32$ and $M_l = 31$, the worst case will requires 5 extra steps at the highest level. For a complete walk of a trie, the worst case amount of extra steps in total will be 15. While this value will remain relatively low for tries where M_m and M_l do not differ by too much, it could theoretically be M_m at each step.

This algorithm will not impact the height in any way, as it does not modify the trie. The worst runtime is $\mathcal{O}(M \times h(R))$: In the worst case, all nodes we traverse have a size table, and we have to check all M entries in it to find the right path. The best access time is $\Omega(h(R))$, in which we either pick the correct path all the time.

3.3 Update

Updating a value in an RRB-tree is in and by itself the same implementation described in Section 2.4: A shallow copy of the RRB-tree and the nodes down the path taken is done. Listing 3.2 shows the algorithm, using the relaxed radix search to find the path down to the leaf node.

```
1  function RRB-UPDATE(R, i, val)
2      R′ ← CLONE(R)
3      T ← CLONE(R_root)
4      R′_root ← T
5      for h ← h(R′) downto 1 do
6          i′ ← CALCULATE-CHILD-INDEX(T, i, h)
7          i ← UPDATE-INDEX(T, i, h)
8          T_i′ ← CLONE(T_i′)
9          T ← T_i′
10     end for
11     i′ ← i mod M
12     T_i′ ← val
13     return P′
14 end function
```

Listing 3.2: RRB-tree update.

To ease readability, the relaxed radix search is abstracted behind the functions CALCULATE-CHILD-INDEX and UPDATE-INDEX. They can be considered functions representing the lines 5-13 in Listing 3.1, but which implicitly calculates d. An actual implementation would include optimisations explained earlier to increase performance.

As easily seen, the algorithm does not update the size tables, nor copy them. As the size of all tries will still be the same, there is simply no need to do so. It also means that the update function will not affect the height of an RRB-tree.

RRB-UPDATE has worst case runtime $\mathcal{O}(2M \times h(R))$, and best case $\Omega(M \times h(R))$. The runtime is equal to the relaxed radix access, with an additional M for the required copy at each level in the trie.

3.4 Concatenation

The concatenation algorithm differs based upon which relaxation method we choose. We will start with the leftwise relaxed dense RRB-tree. From there on, we extend the concatenation algorithm to satisfy the search step relaxation constraints, which increases performance.

3.4.1 $M_l - M_M$ Concatenation

The $M_l - M_m$ *Concatenation* algorithm works as follows: Assume we concatenate two trees, T_l and T_r:

$$T_l +\!\!+\ T_r = T_T$$

The algorithm starts by walking down to the rightmost node at level 1 in T_l, denoted Q_l^1, and the leftmost node at level 1 in T_r, denoted Q_r^1. To be able to concatenate Q_l^1 and Q_r^1, all of their children (leaf nodes) must conform to the leftwise relaxed dense invariant. As such, if the leftmost leaf node in Q_l^1 contains less than M_l elements in it, it either has to redistribute the leaf node over the other leaf nodes, or redistribute slots from the other leaf nodes such that all nodes has at least M_l elements.

It then takes the leaf nodes and insert at most M_m leaf nodes into the replacement node U_l^1, and the remaining nodes (if any) in the replacement node U_r^1. After that, replace the rightmost node at level 1 in T_l with U_l^1 and the rightmost node at level 1 in T_r with U_r^1. The algorithm continues by repeating the process at level 2, until the whole tree has been concatenated.

However, is it always possible to redistribute the leaf nodes such that they satisfy the leftwise dense invariant? There are two cases to consider:

1. Both Q_l^1 and Q_r^1 are leftwise relaxed dense.
2. Q_l^1 is leftwise relaxed dense, and Q_r^1 is fully relaxed dense.

The first case is easy to resolve: If Q_r^1 is leftwise relaxed dense, then the rightmost leaf node can be leftwise relaxed dense. Why? Because Q_r^1 must be, by definition, the rightmost node at level 1 in T_r. If the rightmost leaf node can be leftwise relaxed dense, we can always distribute the slots over some amount of leaf nodes whose size is $M_l \leqslant \|T_{leaf}\| \leqslant M_m$, and put the "leftovers" in the last slot.

In the second case, all leaf nodes must be rebalanced to contain between M_l and M_m node. If the last leaf node in Q_l^1 is less than M_l long, the leaf nodes must be rebalanced. We ignore all the other leaf nodes in Q_l^1, as they satisfy the invariant by definition.

We know that $M_l \leqslant \|Q_r^1\| \leqslant M_m$, and that all the leaf nodes are of length M_l or better. As a consequence, the minimum number of slots the children of Q_r^1 contain is M_l^2, and the most is M_m^2. If we include the rightmost slot in Q_l^1, the total amount of slots S is

$$M_l^2 + 1 \leqslant S \leqslant M_m^2 + (M_l - 1)$$

because the rightmost child in Q_l^1 contains at least 1 and at most $M_l - 1$ slots.

These S slots then has to be redistributed over $M_l \leqslant n \leqslant M_m + 1$ nodes such that all n nodes contain between M_l and M_m slots. If $M_l^2 \leqslant S \leqslant M_m M_l$, then we can distribute the slots such that we have M_l nodes which all contain between M_l and M_m slots. If $M_m M_l < S \leqslant M_m^2$, then we can always have M_m nodes which all contain between M_l and M_m nodes.

If $S > M_m^2$, then there is not enough space in M_m nodes, and we have to redistribute slots into the short node from the full ones. In the worst case, the node contains a single element, and we have to move $M_l - 1$ elements from the other nodes into it. This would require copying up to M_m^2 nodes, and always gives us $M_m + 1$ nodes.

The same logic applies at the next level. The only consideration to take into account is the case where $U_r^1 = \text{NIL}$, in which U_r^1 is discarded.

As the worst case concatenation cost is M_m^2 at a single level, and the height of an RRB-tree is $h(R) \leqslant \lceil \log_{M_l}(|R|) \rceil - 1$, the worst case runtime is $\mathcal{O}(M_m^2 \log_{M_l}(|R|)$.

3.4.2 Search Step Concatenation

The search step concatenation algorithm works in the same fashion, but uses the search step invariant instead: All the S slots must be contained in at most

$$\left\lceil \frac{S}{M} \right\rceil + e_{max}$$

nodes. It is easy to see that it is always possible to redistribute the slots over a set of nodes: We can always fill up all the $\lfloor S/M \rfloor$ first nodes, and then insert the "leftover" slots in the last node. Theorem 3.3 also proves that it is safe split the redistribution into two trie nodes, as both trie nodes will satisfy the invariant.

Theorem 3.3. *Let* T *be the trie representing the expansion of the ordered list* $\langle A, B \rangle$*, where* A, B *are tries.* $\|T\|$ *is allowed to be higher than* M*.*

If $\|T\| \leqslant \|T_{opt}\| + k$ *for some nonnegative integer* k*, then*

$$\|A\| \leqslant \|A_{opt}\| + k$$
$$\|B\| \leqslant \|B_{opt}\| + k$$

Proof. Assume $\|T\| = \|T_{opt}\| + k$, and let A be the result of picking n random children from T. B is the remaining children in T which are not in A.

A can not be of a length higher than $\|A_{opt}\| + k$. As A only contains children from T, if $\|A_{opt}\| + k < \|A\|$, then $\|T_{opt}\| + k < \|T\|$, which contradicts our initial assumption. The same logic applies to B. □

This concatenation algorithm can in fact do even better, as it can safely skip nodes with $M - \frac{e_{max}}{2}$ slots or more, and redistribute nodes with a lower size over the siblings right of them.

To see why, first assume that the concatenation step only contains nodes which have $M - \frac{e_{max}}{2}$ or more slots. In the worst case scenario, we concatenate two fully populated nodes, so we have 2M nodes. In the least dense case, all of these will contain exactly $M - \frac{e_{max}}{2}$ slots. By Theorem 3.1, the optimal slot usage is

$$\left\lceil \frac{(M - \frac{e_{max}}{2}) \times 2M}{M} \right\rceil = 2M - e_{max}$$

and, as such, satisfies Definition 3.6 and require no rebalancing.

Next, consider the case where we have n nodes. The first $m - n \geqslant 0$ nodes all contain at least $M - \frac{e_{max}}{2}$ slots, and the last $m > 0$ nodes contain in total $(m-1)M + 1$ slots. All the last m nodes have less than $M - \frac{e_{max}}{2}$ slots in them. This means that, if we break the invariant, we are unable to rebalance and retain the invariant if we do not redistribute some slots over the first $n - m$ nodes.

The least amount of total slots which we may have to rebalance, is therefore $m - n$ nodes which contain $M - \frac{e_{max}}{2}$ slots. The remaining m nodes contain in total $(m - 1)M + 1$ slots. Thus, the optimal use of slots is, by Theorem 3.1,

$$\left\lceil \frac{(n - m) \times (M - \frac{e_{max}}{2}) + (m - 1)M + 1}{M} \right\rceil = \left\lceil \frac{1 + \frac{e_{max}}{2}(m - n)}{M} \right\rceil + n - 1$$

If we distribute all the $(m - 1)M + 1$ slots over the last m nodes evenly, we get the

following inequality:

$$\begin{aligned}
\frac{(m-1)M+1}{m} &< M - \frac{e_{max}}{2} \\
(m-1)M+1 &< mM - m\frac{e_{max}}{2} \\
mM - M + 1 &< mM - m\frac{e_{max}}{2} \\
mM &< M - 1 + mM - m\frac{e_{max}}{2} \\
m\frac{e_{max}}{2} &< M - 1 \\
m &< \frac{2(M-1)}{e_{max}}
\end{aligned}$$

It follows that we can not have any m higher or equal to $2(M-1)/e_{max}$ as, by the pigeonhole principle, at least one node must have $M - \frac{e_{max}}{2}$ or more slots in it.

Now, let us attempt to minimise the value of the ceiling in the optimal slot usage expression. In order to do so, we minimise m and maximise n:

$$\begin{aligned}
m &= 1 \\
n &= 2M
\end{aligned}$$

By substituting these values in the ceiling expression, we get

$$\begin{aligned}
\left\lceil \frac{1 + \frac{e_{max}}{2}(m-n)}{M} \right\rceil &= \left\lceil \frac{1 + \frac{e_{max}}{2}(1-2M)}{M} \right\rceil \\
&= \left\lceil \frac{1 + \frac{e_{max}}{2} - Me_{max}}{M} \right\rceil \\
&= \left\lceil \frac{1 + \frac{e_{max}}{2}}{M} \right\rceil - e_{max} \\
&= 1 - e_{max}
\end{aligned}$$

and as a result, the lowest possible optimal slot size in these scenarios are

$$(1 - e_{max}) + n - 1 = n - e_{max}$$

This shows that there is no need to rebalance nodes of size $M - \frac{e_{max}}{2}$, which proves that this optimisation is sound.

The CREATE-CONCAT-PLAN presented in Listing 3.3, contains the essence of this algorithm. The function takes a trie node T, which may be wider than M elements, and returns an array c of the rebalanced node sizes, along with the new total length n.

As $\|T_{opt}\|$ is required to know when to stop rebalancing, it is calculated in the lines 3-8: The algorithm itself starts at line 11: Lines 12-14 skips over nodes which we can

```
1  function CREATE-CONCAT-PLAN(T)
2      c ←CREATE-INT-ARRAY(‖T‖)
3      S ← 0
4      for i ← 0 to ‖T‖ − 1 do
5          c_i ← ‖T_i‖
6          S ← S + ‖T_i‖
7      end for
8      ‖T_opt‖ ← ⌈S/M⌉
9      n ← ‖T‖                                   ▷ Length of rebalanced node
10     i ← 0
11     while ‖T_opt‖ + e_max < n do
12         while c_i ⩽ M − e_max/2 do
13             i ← i + 1
14         end while
15         r ← c_i                               ▷ Distribute i over remaining nodes
16         while r > 0 do
17             MIN-SIZE ←MIN(r + c_{i+1}, M)
18             c_i ← MIN-SIZE
19             r ← r + c_{i+1} − c_i
20             i ← i + 1
21         end while
22         for j ← i to n − 1 do
23             c_i ← c_{i+1}
24             j ← j + 1
25         end for
26         i ← i − 1
27         n ← n − 1
28     end while
29     return c, n
30 end function
```

Listing 3.3: Algorithm to create a concatenation plan.

safely skip. Actual redistribution is done at lines 15-21, where the remaining slots to distribute is r, initially all the slots. Then the current slot, initially the old node, is replaced with its right sibling plus some of the slots to distribute. The new node slot count cannot be more than M, and if there are remaining slots to redistribute, we continue the slot distribution.

Since the plan has removed a node from T, all nodes right to it must be shifted one step to the left. The ones that received additional nodes have already been moved, and the remaining ones are moved in the lines 22-25. Finally, realise that the last node we distributed slots over can still contain less than $M - \frac{e_{max}}{2}$ slots. Since we have checked all nodes before this one, we must start next step by checking it again, hence the decrementing at line 26.

The additional functions needed to rebalance an RRB-tree is further explained in Appendix A: The only important thing to know within this section is that the original nodes Q_l^k and Q_r^k are temporarily merged into a single node, which is why CREATE-CONCAT-PLAN must allow trie nodes to be greater than M slots large.

In the worst case, this algorithm receives two fully populated trie nodes and concatenates them together, having 2M nodes. The first node has a single slot, then follows $2M - 2 - e_{max}$ fully dense nodes. Finally, the last $1 + e_{max}$ nodes contains a single slot. In this case, the algorithm will redistribute the first node. Since only the last nodes can contain more slots, the redistribution will happen over all the fully dense nodes, which requires the algorithm to copy $(2M - 2 - e_{max})M < 4M^2$ slots. As a result, this algorithm also has a worst case runtime of $\mathcal{O}(M_m^2 h(|R|))$. However, the worst case scenario happens very infrequently, and on average this algorithm is faster than the $M_m - M_l$ concat algorithm by a factor of 3 [3].

It is not impossible to improve the worst case runtime: For instance, one can attempt to find the cheapest small node to redistribute first. However, this may affect average runtime, and alternative algorithms should be studied with that in mind. As the goal of this thesis is to improve append performance, such research is left as future work.

3.4.3 Height

As earlier mentioned, the size of an RRB-tree is theoretically unbounded if we only use Definition 3.6. Therefore, its lower bound depends on how the operations on it manipulate the tree. Assuming only the concatenation algorithm operates on the RRB-tree, we can prove that it is strictly less than a logarithmic factor not too far off from the leftwise relaxed dense trie:

Theorem 3.4. *For a search step relaxed radix balanced tree* R *with branching factor* M *and error threshold* $e_{max} < M - 1$*,*

$$h_m(R) \leqslant h(R) < h_l(R)$$

if only concatenations are done on the RRB-tree, where

$$\begin{aligned} M_l &= M - e_{max} - 1 \\ h_l(R) &= \log_{M_l}(|P|) - 1 \\ h_m(R) &= \lceil \log_M(|P|) \rceil - 1 \end{aligned}$$

assuming $|R| > 0$.

Proof. Assume that

$$\begin{aligned} M - 1 < |T| \quad &\text{iff } h(T) = 0 \\ (M - e_{max} - 1)^{h(T)} \times M < |T| \quad &\text{iff } h(T) \neq 0 \end{aligned} \qquad \text{(MIN-H)}$$

for a fully populated node T is true.

We know that a fully populated node T where $h(T) = 0$ is a leaf node with M elements, which satisfies (MIN-H).

Next, consider the fully populated node T, $h(T) = 1$. In general, a fully populated node T at height $h(T)$ contains M children and satisfies the expression

$$\|T_{opt}\| + e_{max} \leqslant \|T\|$$

because

$$\|U_{opt}\| + e_{max} \leqslant \|U\|$$

where U is the expansion of the ordered list $\langle T, T' \rangle$, and T' is one of T's immediate, possibly previous, siblings. The expression $\|U_{opt}\| + e_{max} \leqslant \|U\|$ is checked by the concatenation algorithm at some point, and from Theorem 3.3, we thus know T satisfies this expression.

In a least dense, fully populated node T,

$$\|T\| = M = \|T_{opt}\| + e_{max}$$

We know from Theorem 3.1 that some redistribution of T will, in the least dense case, give $\|T_{opt}\| - 1$ fully dense nodes, and 1 node with a single child. Some redistribution of T therefore contains $M - e_{max} - 1$ fully populated nodes of height $h(T) - 1$, and $1 + e_{max}$ nodes which contain only a single child.

As such, some redistribution of a least dense, fully populated node T at height $h(T) = 1$ contains $M - e_{max} - 1$ fully dense leaf nodes. Its size is thus

$$(M - e_{max} - 1) \times M < |T|$$

as the remaining $1 + e_{max}$ nodes contain at least 1 element. This satisfies (MIN-H).

Next, for heights $\leqslant h(T)$, we must prove that the hypothesis holds for $k = h(T) + 1$. Recall that some redistribution of a least dense, fully populated node T' at height $h(T') = h(T) + 1$ has $M - e_{max} - 1$ fully populated nodes at height $h(T)$. As (MIN-H) holds for nodes of height $h(T)$, we know that for all T,

$$(M - e_{max} - 1)^{h(T)} \times M < |T|$$

The size of T' must at least be the sum of the fully dense node's sizes. Therefore,

$$(M - e_{max} - 1)(M - e_{max} - 1)^{h(T)} \times M = (M - e_{max} - 1)^{h(T')} \times M < |T'|$$

and (MIN-H) holds by induction.

Next, notice that concatenating a trie of height k with some other trie of height $\leqslant k$ will in the worst case increase the height of a trie by 1: If there is not enough space

in a single trie node of level k, a new node containing both nodes will be placed at height $k+1$.

There is not enough space in a single trie node if there are at least $M+1$ slots at level k which combined satisfies the invariant. In the least dense case, the total length of all the slots S would be

$$\|S\| = \|S_{opt}\| + e_{max} = M + 1$$
$$\|S_{opt}\| = M + 1 - e_{max}$$

As a result, some redistribution of the $M+1$ nodes will give $M - e_{max}$ fully populated nodes, and $1 + e_{max}$ nodes with a single child. If we assume all $M - e_{max}$ are least dense, we have by (MIN-H) that

$$(M - e_{max})(M - e_{max} - 1)^{h(T)-1} \times M < |T|$$

As $M - e_{max} - 1 < M - e_{max} < M$, we have that

$$(M - e_{max} - 1)^{h(T)+1} < |T|$$

and by the definition of the logarithm:

$$h(T) < \log_{M-e_{max}-1}(|T|) - 1 \qquad \square$$

Although Theorem 3.4 proves that the height of a search step relaxed RRB-tree is as good as a relaxed RRB-tree where $M_m = M$ and $M_l = M - e_{max} - 1$, it is very unlikely that this happens in practise.

Why is this so? Recall that any concatenation will "zip" the rightmost node T of the left trie and leftmost nodes of the right trie U at all levels: The expansion S of those nodes must, according to the concatenation algorithm, satisfy

$$\|S\| \leqslant \|S_{opt}\| + e_{max}$$

Assuming both U and T are least dense, through some redistribution, they will have $e_{max} + 1$ nodes with a single slot, and $\|T_{opt}\| - 1$ or $\|U_{opt}\| - 1$ fully dense nodes. Their expansion will thus contain $2e_{max} + 2$ children with a single child. As such, they must be rebalanced so that U and T *combined* uses e_{max} extra search steps.

It is tempting to draw the conclusion that any immediate siblings must *combined* have at most e_{max} excessive children, but this is not true. Assume L and C have *combined* e_{max} excessive children, and that all the excessive children lie in L: $\|C\|$ is therefore $\|C_{opt}\|$. Next, assume that C is compared with R. R has e_{max} excessive children, so combined, they have e_{max} excessive children. However, the concatenation algorithm redistributes the nodes, such that C', the replacement for C, now contains the small nodes. Now, both L and C combined have more than e_{max} excessive children.

Luckily, L and C will rarely have more than e_{max} exessive children: Benchmarks by Bagwell and Rompf indicates that the height of an RRB-tree will rarely, if at all, be more than $\log_{M-e_{max}}(|R|)$ in height [3].

3.5 Slicing

Although the concatenation algorithm is complex, the slicing algorithm is not: It only cuts the left and right hand side. Cutting the right hand side is done by copying the path down to the last element which resides in the slice, and removing table entries in nodes right of the entry to walk. The size tables, if any, also has to be copied and updated.

Cutting on the left hand side requires a bit more work, but uses the same idea: Remove all table entries in nodes left of the entry to walk. In addition, since the leftmost leaf in the new entry is very likely to not be leftwise dense, we add in size tables to all trie nodes we create. If the size table does not already exist in the original node, we calculate the size table entry by the height of the slot entry multiplied with the entries to its left plus one:

$$\sigma_i(T) = (i + 1) \times M^{h(T_i)+1}$$

The only exception to this rule is the last element: If T is not fully dense, then the rightmost child will contain less elements. The total size of T is either contained in the difference between two size slots stored in its grandparent G, or, if this is the root node, the total size of the vector.

Since we have a size table for the trie node or can calculate its values, we can create a new size table based upon this. Assume there are j elements left of the slot we will walk. If T' is the new trie node, then

$$\sigma_i(T') = \sigma_{j+i}(T) - \mathit{left}$$

where *left* is the index of the first element in the sliced RRB-tree, in the original RRB-tree.

Finally, if the new root node contains just a single child, the root is replaced with its child. This step is repeated until either the new root node contains more than one child, or the node is a leaf node.

The problem with this implementation is that it is not rebalancing the sliced RRB-tree. In the worst case, one can request a slice containing two nodes, which still has the same height of the original tree. Additionally, the invariant may, in some occations, be broken.

Is this a problem in practise? The possible lack of height reduction is certainly problematic, but breaking the invariant will not cause the height of an RRB-tree to increase notably. To see why, notice that the invariant can only be broken by the rightmost and leftmost nodes at each level. As such, the invariant break is constrained to at most $2h(R) + 1$ nodes (the root is counted only once).

How big impact has this invariant break? Recall that Theorem 3.3 says we can split any node into two parts without breaking the invariant. Thus, only the shrinking of

the length of a slot entry may cause the node to break the invariant. As a result, the node may contain one slot more than what is allowed in the worst case scenario:

$$\|T_{opt}\| \leqslant \|T\| \leqslant \|T_{opt}\| + e_{max} + 1$$

Furthermore, since the invariant may be broken only at the leftmost and rightmost nodes, concatenations we perform with this tree will fix some of the breaks: The result of a concatenation will have the breaks fixed at either the left or right side through rebalancing.

The RRB-tree may increase its height for slightly smaller tree sizes, but as the invariant break happens infrequently and does not impact a large part of the tree, it can be considered negigible.

The asymptotic runtime of an RRB-tree slice is proportional to the height of the RRB-tree. For each step, we may in the worst case copy a node and a size table of length M. As we perform the slice both to on the right and the left side, its worst case runtime is $\mathcal{O}(4M \times h(R))$.

3.6 Performance

Assuming all other operations are based upon slicing and concatenation, the invariant will be kept to the extent described in previous section. Appends can be implemented by creating a new RRB-tree with a single element, and concatenate the new tree into the original. Popping can be achieved by slicing off the last element: As this is only a right slice, it takes worst case 2M time instead of the 4M required by a full slice.

Although many of the persistent vector operations can be considered effectively constant time, does the same apply to the RRB-tree? Bagwell and Rompf measured its performance on all of these operations, and found that, on average, index times were twice as slow as in a regular vector, updates a little more than twice as slow. Concatenations were 2 to 6 times as slow as an RRB-tree update on average, and in the worst case 30 times as slow.

It is tempting, then, to consider appends and concatenations to be effectively constant time. However, [19] shows that append times using the concatenation implementation is clearly not constant time, and is in fact very costly. Other operations seem to run in effectively constant time, although they were not as heavily measured.

Table 3.1 presents all the operations with their asymptotic and effective runtimes. While the effective runtimes looks similar to the persistent vector, effectively constant time operations are in the worst case a factor of two slower or more. However, when no concatenation or left slicing is done, all operations on RRB-tree can reuse the persistent vector operations, as the trie is guaranteed leftwise dense. In and by itself, this looks promising, with the notable exception of the worst case append times.

Name	Operation	Actual Runtime	Eff. Runtime
Lookup	A_i	$\mathcal{O}(\log_{M_l} n)$	$\mathcal{O}(\sim 1)$
Last	*last*(A)	$\mathcal{O}(\log_{M_l} n)$	$\mathcal{O}(\sim 1)$
Count	$\lvert A \rvert$	$\Theta(1)$	$\Theta(1)$
Update	$A_i \leftarrow x$	$\mathcal{O}(2M_m \log_{M_l} n)$	$\mathcal{O}(\sim 1)$
Append	*conj*(A, x)	$\mathcal{O}(M_m^2 \log_{M_l} n)$	$\mathcal{O}(\log n)$
Pop	*pop*(A, x)	$\mathcal{O}(2M_m \log_M n)$	$\mathcal{O}(\sim 1)$
Iteration	for x in A:	$\Theta(n)$	$\Theta(n)$
Concat	$A +\!\!+ B$	$\mathcal{O}(M_m^2 \log_{M_l}(\lvert A \rvert + \lvert B \rvert))$	$\mathcal{O}(\log n)$
Right slice	$A_{(0,i)}$	$\mathcal{O}(2M_m \log_M n)$	$\mathcal{O}(\sim 1)$
Left slice	$A_{(i,\lvert A \rvert)}$	$\mathcal{O}(2M_m \log_M(n))$	$\mathcal{O}(\sim 1)$
Slice	$A_{(i,j)}$	$\mathcal{O}(4M_m \log_{M_l}(n))$	$\mathcal{O}(\sim 1)$

Table 3.1: Time complexity of RRB-tree operations, where $n = \lvert A \rvert$.

CHAPTER 4

Transience

Immutable data structures are hard to implement efficiently. Whereas the persistent vector works well in general, its performance might degrade the performance or capacity of a system if used heavily within bottlenecks. By turning persistent structures to *transient* data structures inside such bottlenecks, we can further increase the performance of these structures.

4.1 Definition

Transience can best be explained as removing a data structure's persistence: An update on the transient data structure $P^!$ returns – just as a persistent data structure – a new transient $P^{!\prime}$. However, in addition, the update will *invalidate* $P^!$, meaning that any operation on $P^!$ after or during the update is considered illegal. By not preserving $P^!$, one can perform mutations on $P^!$ and return $P^{!\prime}$ as the mutated variant.

Definition 4.1. *If a transient data structure* $P^!$ *is* invalidated*, it cannot be used as argument to any function.*

One might ask which category transient data structures are in. They are clearly not persistent, as they do not preserve previous versions of the structure. People knowing the internals of a transient might argue that a transient is a mutable data structure, as it mutates the structure. However, it is not mutable based upon the definition used by us: Reading a value from a transient will always return the same value, and a transient has no perceived state or identity. For a user of a transient, the transient is effectively an immutable data structure which does not preserve older versions of itself: The mutations inside a transient are not externally visible when used correctly.

To formally define transience, we need some definitions. Let use denote the type of persistent vectors which contain the type τ by $\mathcal{V}_\tau$, whereas a transient vector of same type is denoted $\mathcal{V}^!_\tau$. An *instance* of $\mathcal{V}_\tau$ will generally be denoted P, whereas its transient counterpart will be denoted $P^!$.

Definition 4.2. *We will denote a function on the data structure type* α *as*

$$f : \alpha \times A_1 \times \ldots \times A_n \to B$$

where the additional n *arguments are of type* A_1 *up to* A_n. B *is the return type from the function.*

As an example of a concrete function, consider the lookup function *lookup* presented in Section 2.3. *lookup* takes in a persistent vector of type $\mathcal{V}_\tau$ and a non-negative integer, returning an instance of type τ:

$$lookup : \mathcal{V}_\tau \times \mathbb{N} \to \tau$$

On a specific persistent data structure, we can categorise functions within two categories: Functions which read values from the data inside the structure, and functions which "update" the structure and return a new version of the data structure.

Definition 4.3. *A* read *function* f_r *which only reads values from the persistent collection type* α_τ *is denoted as*

$$f_r : \alpha_\tau \times A_1 \times \ldots \times A_n \to B$$

where $B \neq \alpha_\tau$ *unless* $\tau = \alpha_\tau$.

Definition 4.3 formalises the read functions, and says that those functions cannot return a new persistent data structure based on the input data structure. The special case is there as the data structure can of course contain other data structures which are of the same type.

Definition 4.4. *A* write *function* f_w *which performs an update on the persistent collection type* α_τ *is denoted as*

$$f_w : \alpha_\tau \times A_1 \times \ldots \times A_n \to \alpha_\tau$$

for all types τ.

Definition 4.4 simply states that the return value of a write function is based upon the first input value.

Definition 4.5. *A transient data structure* $\alpha^!_\tau$ *is an immutable data structure based upon* α_τ. *For any defined* write *function in the core language*

$$f_w : \alpha_\tau \times A_1 \times \ldots \times A_n \to \alpha_\tau$$

where $A_i \neq \alpha_\tau$ *for all* i, *there exists a corresponding function*

$$f^!_w : \alpha^!_\tau \times A_1 \times \ldots \times A_n \to \alpha^!_\tau$$

When any function $f^!_w$ *is called on* $P^!$, *it* invalidates $P^!$.

For any defined read *function in the core language*

$$f_r : \alpha_\tau \times A_1 \times \ldots \times A_n \to B$$

where $B \neq \alpha_\tau \neq \alpha^!_\tau$ *unless* $\tau = \alpha_\tau$ *or* $\tau = \alpha^!_\tau$*, there exists a corresponding function*

$$f^!_r : \alpha^!_\tau \times A_1 \times \ldots \times A_n \to B$$

At any point in time, only a single function $f^!_r$ *or* $f^!_r$ *can operate on a specific transient data structure* $P^!$.

In addition, there exists two functions which can convert an instance of α_τ *to* $\alpha^!_\tau$ *and back:*

$$transient : \alpha_\tau \to \alpha^!_\tau$$
$$persistent^! : \alpha^!_\tau \to \alpha_\tau$$

The function persistent$^!$ *invalidates the transient instance given as input.*

Definition 4.5 states that all read and write functions in the core language on the persistent collection also exists on its transient counterpart, if it is defined. All write operations invalidates the transient, and at any point in time, only a single transient function can be called on a specific data structure. In addition, functions that can convert a persistent data structure to a transient and back must exist.

In addition, we have the following definition for user-defined functions:

Definition 4.6. *A user-defined function*

$$f^!_{uw,m} : A_1 \times \ldots \times A_n \to B$$

is a write *function on argument* $a_m \in A_m = \alpha^!_\tau$*, where* $1 \leqslant m \leqslant n$*, if and only if* $f^!_{uw,m}$ might *call a write function* $f^!_w$ *with* a_m *as its first argument, or if* $f^!_{uw}$ *calls the user-defined write function* $g^!_{uw,k}$ *with* a_m *as the* k*th argument.*

Definition 4.7. *A user-defined function*

$$f^!_{ur,m} : A_1 \times \ldots \times A_n \to B$$

is a read *function on argument* $a_m \in A_m = \alpha^!_\tau$*, where* $1 \leqslant m \leqslant n$*, if and only if it is not a* write *function on* a_m.

These two definitions simply state that a user-defined function $f^!_{uw,m}$ may, either directly or indirectly, call a transient update function which invalidates argument a_m. A user-defined read function $f^!_{ur,m}$, on the other hand, will never invalidate a_m, but may perform read functions either directly or indirectly.

Based on these definitions, we might expect that we can relatively easily know whether it is okay to perform transient operations on a value a passed in. If we *might* end up calling a user-defined write function or a write function on a, it will be incorrect to

call any function with α later on. If we call a function which is a read function, we must wait until the function has finished before we can call any other function with α.

However, it is not as simple as that: A function might start up a new thread, and pass α to the thread. Or a function might save α in a mutable data store, which then is used by another function or the same function later on. We avoid taking these cases into consideration, and assume that all read and write functions never do "bad" things.

4.2 Implementation

While the semantics related to transience is now defined, we have not yet talked about how we can get the claimed performance benefits discussed in the beginning of this section. We will explore the ideas used to enable transience in Clojure's persistent vectors, maps and sets, with a special focus on how they work within vectors. However, the same ideas can be used on any tree-based structure.

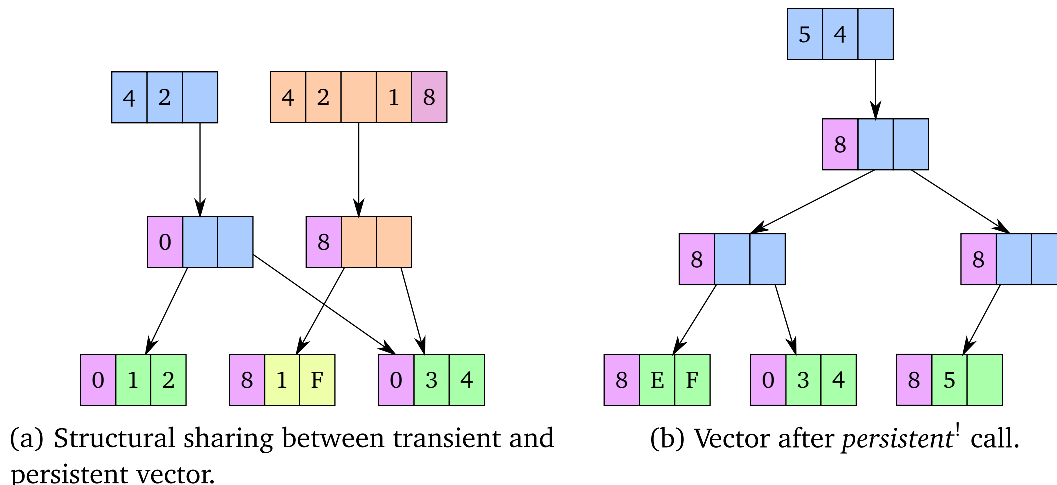
The definition of transience says that the previous transient $P^!$ is invalidated whenever an update operation is performed. This means that we can safely change nodes which are only visible to $P^!$ by mutating them, as no other structure will see these changes. The question, then, is how we can detect and ensure that a mutation on a node is visible for just $P^!$ and not any other vector.

One can ensure the node is only visible for $P^!$ by attaching an ID on every node in the trie. For persistent structures, the ID is irrelevant, and we can therefore use NIL for all persistent vectors. However, transients has to create a unique ID: A cheap way of creating a guaranteed unique ID is by creating a very small object and put it on the heap. Then, whenever one has to match IDs on nodes, compare object pointers.

```
1  function TRANSIENT-CLONE(val, id)
2      if val_id ≠ id then
3          val′ ← CLONE(val)
4          val′_id ← id
5          return val′
6      else
7          return val
8      end if
9  end function
```

Listing 4.1: CLONE function used in transients.

Now we can do path copying in the same fashion as we have done before. However, instead of cloning every single node we traverse, we check its ID first. If the ID is equivalent to the one in $P^!$, we know that the node was created by $P^!$ or an invalidated version of $P^!$. We can therefore safely mutate the node, and thus have no need to



(a) Structural sharing between transient and persistent vector.

(b) Vector after *persistent*$^!$ call.

Figure 4.1: Transient vectors.

create a copy. Otherwise, we clone the node, set the ID of the copied node to the transient ID, and return the copy. Listing 4.1 represents the rather straightforward implementation of this new CLONE function.

To ensure that the transient is not modified after a *persistent*$^!$ call, the implementation NILs the unique ID within the transient vector head. In addition, whenever a transient operation is used, the unique ID is checked, and if it is NIL, an error is thrown. This is not necessary in a language which can do this at compile time, but we will see that this is not easy in the next section.

A specific implementation consideration for the persistent vector is that its tail is usually compressed. For a transient, however, that would require a new copy of the tail for each element appended to the transient, which is costly. Therefore, whenever a transient structure is created, its tail is immediately copied and its capacity is increased to the max size M. Whenever the function *persistent*$^!$ is called, the tail then has to be compressed again.

Let us consider an example of how this would work in practise: For convenience, we shall consider vectors not using a tail. Assume we have a persistent vector `[1, 2, 3, 4]` and we call the function *transient*, followed by updating index 1 to F, 0 to E, appending the value 5 and then finally converting the transient back through *persistent*$^!$.

When the transient is first created, the thread ID[1] is stored and a random ID is generated. Then, when we update index 1, and as none of the nodes have the transient's ID, normal path copying is done. However, we store the transient's unique ID as the ID for the nodes we copy. Figure 4.1a shows the original vector to the left, and the transient on the right: The transient has in this case been created in thread

[1]The thread ID is stored to ensure that the transient is used in a single function at any point in time; see Section 4.4.

number 1, and has the random ID 8. The leftmost slot in each node represents its ID, and nodes with the ID NIL has the ID 0.

Next, the value at index 0 is updated to E: In this case, we do not have to perform any copying, as the ID in the nodes we walk is equal to the transient's ID. We then attempt to insert the value 5 into the trie, but as there is no space, we have to increase the height and create new nodes. The newly created nodes also contain the transient's ID. Finally, we call *persistent!*, and end up with the result visualised in Figure 4.1b. Note that the leaf node containing the values 3 and 4 is still shared with the original vector[2], and the persistent vector does not modify the nodes created by the transient.

We see that, while persistent vectors create new nodes with the ID NIL, they might contain nodes with non-NIL IDs. This naturally brings up the question: Should CLONE explicitly set the ID to NIL? Whether it does or not does not really matter: Recall that the generated IDs are guaranteed unique, so a transient checking the ID of a node will not incorrectly mutate a node it does not own. Additionally, we know that all the IDs in a persistent vector will either be NIL or an ID of a destroyed transient: The only way a transient can transfer its nodes to a persistent vector is by the function call *persistent!*, which invalidates the transient.

The Clojure transients on a persistent vector works somewhat different than the description given here. Instead of storing the thread ID and a unique ID in the transient vector head, it copies the root node and stores the thread ID inside an atomic reference. To check that the transient is used in the thread it was created, it compares the thread ID with the root's atomic reference value, and uses the atomic reference object as unique ID. This difference thus require the implementation to copy the root to store the new ID, even though the transient may not touch the trie at all. Apart from that, these differences are effectively only cosmetic, and does not change the actual semantics.

4.3 Performance

For tree-like structures, the asymptotic runtime will not change. If the node size is a constant, copying the node size will still be a constant time operation. However, in many cases, the most considerable part of an operation is memory allocation and memory copies. A transient will therefore significantly reduce the constant factors for all operations which require memory allocations.

It is important to realise that transient operations on the trie will at first perform a path copy in the same fashion as a persistent vector. As such, transients are not a good fit if you only perform relatively few operations on the vector before turning it back into a persistent vector. Most bottlenecks will not be related to few operations, however, but rather huge bulk operations such as appends or updates.

[2]The original vector is not shown in Figure 4.1b due to lack of space.

4.4 Related Work

Earlier work has lead to the definition of linear types, a way to define mutable types in a similar fashion as transients, with focus on correctness [20]. A linear type can only be used once, although relaxed constraints provide means to temporarily define a linear type as nonlinear, which gives users read capabilities without "using" the instance. In contrast, a transient is an affine type, where the function *persistent!* converts the affine nodes *permanently* to nonlinear, persistent nodes. In addition, use of linear types provide a system where reference counting and garbage collection is unnecessary. As such, it is hard to see how one could use linear types directly to implement transience. However, it seems like a very good starting point to ensure correct usage of transients, which is currently lacking. The new language Rust is a programming language with linear types, and an attempt to implement persistent data structures with transient capabilities in Rust may provide such insights.

```clojure
(loop [t (transient [])
       i 0]
  (if (< i 100)
    (recur (conj! t i)
           (inc i))
    (persistent! t)))
#_=> [1 2 '... 99]

(loop [t (transient {})
       i 0]
  (if (< i 100)
    (recur (assoc! t i (* i i))
           (inc i))
    (persistent! t)))
#_=> {0 0, 32 1024, '...
      63 3969, 95 9025}
```

(1) Correct transient usage

```clojure
(let [t (transient [])]
  (dotimes [i 100]
    (conj! t i))
  (persistent! t))


#_=> [1 2 '... 99]

(let [t (transient {})]
  (dotimes [i 100]
    (assoc! t i (* i i)))
  (persistent! t))


#_=> {0 0, 1 1, 2 4, 3 9,
      4 16, 5 25, 6 36, 7 49}
```

(2) Incorrect transient usage

Listing 4.2: Transient usage in Clojure

Clojure has not a linear type system, and consequently no possiblity to check that one use transients correctly. It is thus fully possible to use transients erroneously and get the "expected" result back, as most transient operations just return themselves. However, as transients *may* return new values, this may cause confusion for users which assumes transients are mutable data structures. As an example, Listing 4.2 shows how one correctly use transients on the left side, and incorrect usage on the right side. While the transient vector (`[]`) returns the same result, the transient map (`{}`) returns a new transient after it contains 8 elements, which makes the incorrect usage returns an erroneous result.

To avoid more than one function using the same data structure at the same time, Clojure forces the user to only use a transient within the thread it was created. This is done by saving the thread ID in the transient, and by checking that the thread ID is equivalent for each transient operation. This is also the reason Figure 4.1a contains the thread ID.

With the exception of Clojure's notion of transience, there has not – to the author's knowledge – been any notable work on increasing performance of persistent data structures explicitly through transients or linear types. The inliner and SSA optimisations in Haskell could potentially detect that there is a only a single reference to a local object, and optimise this by mutating the values instead of creating a new one[21, 22]. However, it seems unlikely that such inlining could provide significant performance benefits to a data structure, and it is clearly not targeting such optimisations explicitly.

Part II

Methodology

Chapter 5

Direct Append

The most notable performance penalty for the RRB-Tree is related to inefficient appending: The implementation by Bagwell and Rompf [3] creates a new RRB-Tree with one element, and merges it into the original RRB-Tree. L'orange [19] also implements appending this way. Although RRB-trees which are leftwise dense can reuse the persistent vector appends, it does not help when the tree is not leftwise dense. However, by modifying the persistent vector append algorithm to work for the RRB-tree, appending can be performed in effectively $\mathcal{O}(\sim 1)$ time as well.

5.1 Introduction

Modifying the original append algorithm for persistent vectors to work for RRB-trees is not as straightforward as one might initially expect. As described in Section 2.5, a persistent vector implementation must generate new empty nodes when there is not enough space in the original trie. This is trivial in the persistent vector through the radix index search: When the size of the vector is $M^{h(P)+1}$, we know that it is fully dense, and therefore we increase its height. When the algorithm attempts to walk a table entry which is NIL, it generates a new empty node.

The problem with the RRB-tree is that the M-ary representation of an index does not tell us which branch to go: As the table entry to walk varies based upon the contents of the size tables, one cannot know there is enough space in the rightmost node without walking down to it. Additionally, all the rightmost nodes may all contain M nodes even if $|R| < M^{h(R)+1}$, meaning that one cannot insert an element without either increase the RRB-tree's height, or through rebalancing. Consequently, an append algorithm for the RRB-tree has to, in the worst case, walk all the rightmost nodes in a vector to ensure there is space enough in the original trie.

The question then, is what the algorithm should do if there is not enough space in the rightmost nodes. Should it attempt to rebalance, or should it just increase the height of the tree? The algorithm explained here just increases the height of the tree: As it is not unlikely that there will be other appends in the future, there may be several rebalancing steps in the future if the height does not increase. If the height

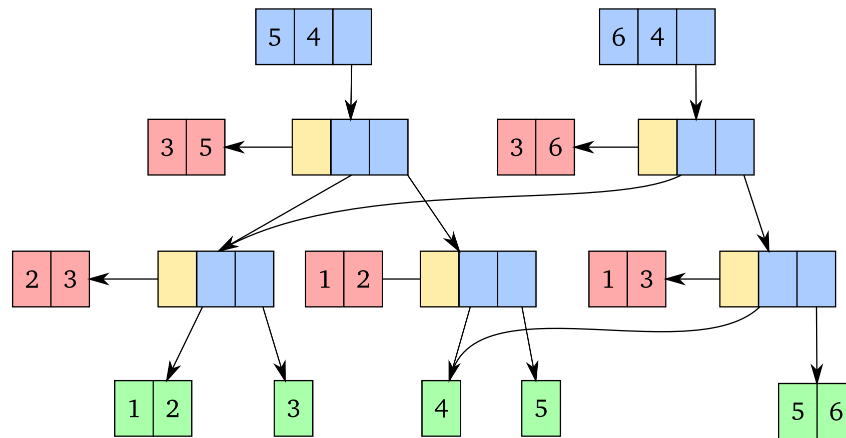


Figure 5.1: Direct appending

does increase as a result of rebalancing, then the rebalancing is just a costly way to perform the height increase.

By not performing a rebalancing, this can clearly break the invariant: Assume that all the n lowest level rightmost nodes in the tree are fully populated, 0 being the lowest level. Adding another child in the rightmost node at level n with a single slot breaks the invariant. This is, however, fortunately the only node affected. A concatenation will fix the invariant break, just as with slicing.

If consecutive appends are applied, the n lowest rightmost nodes will always be leftwise dense, thus they can never break the invariant. When the rightmost node is completely populated, the rightmost node at level n will again satisfy the invariant: Adding a fully dense child to a node which satisfies the invariant will never break the invariant. Another append can do one of two things: Either the rightmost node at level n is not completely full, and we end up with the original case just described. Or, if the rightmost node at level n is fully populated, the same case happens with the rightmost node at level $n + 1$.

A height increase will not break the invariant for the original root, but may break the invariant for the new root. This case is again covered, as the root is a rightmost node in the trie. The direct append algorithm will therefore not break the invariant in a way which will increase the height considerably, and the invariant can be restored by concatenations.

5.2 Implementation

The direct append algorithm is presented in Listing 5.1, and implements the idea described in previous section. COPYABLE-COUNT returns two arguments: n_c, the number of *copyable* nodes, and pos, the position in the last copyable node which is NIL. A node is *copyable* if there is enough space in either this node or in one of its rightmost descendants: The algorithm will copy the node as it still possible to insert an element in the rightmost part of this subtrie.

```
1  function RRB-PUSH(R, elt)
2      R′ ← CLONE(R)
3      |R′| ← |R| + 1
4      n_c, pos ← COPYABLE-COUNT(R)
5      if n_c = 0 then                                  ▷ Not enough space in orig. rrb
6          R′_root ← CREATE-INTERNAL-NODE()
7          R′_root[0] ← R_root
8          σ(R′) ←CREATE-SIZE-TABLE*(R_root)             ▷ Create size table if needed
9          T ← APPEND-EMPTY(R′_root, 1, h(R) + 1)
10         T_0 ← elt
11     else
12         T ← COPY-FIRST-K(R, R′, n_c)
13         T ← APPEND-EMPTY(T, pos, h(R) + 1 − n_c)
14         if h(R) + 1 − n_c = 0 then                    ▷ Enough space in orig. leaf node
15             T_pos ← elt
16         else
17             T_0 ← elt
18         end if
19     end if
20     return R′
21 end function
```

Listing 5.1: RRB-PUSH using direct appending

The function COPY-FIRST-K, as should be evident from its name, copies the first k search steps in the RRB-tree, inserts them into the new RRB-tree, and returns the last copied node. In addition, it updates the size table to the copied nodes, to reflect the change in size. APPEND-EMPTY simply appends k empty nodes in the trie T in its table entry pos. The implementations of these helper functions are further described in Section A.2.

As an example, consider the leftmost vector in Figure 5.1. To append 6 to it, the path down to the rightmost leaf node is walked. We can still insert elements into said node, therefore there are $n_c = 3$ copyable nodes, and $pos = 1$ as that is where the next element is going to be placed. As $h(R) + 1 - n_c$ is zero, we create zero empty nodes, and insert the element at position pos. Notice that the size tables were copied and updated. If the nodes does not contain a size table, then nothing has to be done related to size tables: The trie will still be leftwise dense after the insertion.

5.3 Direct Pop

The opposite operation, POP, could also use the same ideas to get additional performance improvement. In contrast to the direct append implementation, we always know the path down to the element to remove: For a non-leaf subtrie T, walk table entry $\|T\| - 1$. If we are in the leaf node, remove the rightmost entry, and if the rightmost entry is at index 0, return NIL, and repeat at the level above. This is equivalent

to the naïve implementation first explored in Section 2.6. Is it possible to improve this implementation? The improvements discovered for the persistent vector only applies when a subtrie is fully dense, which is not the case for an RRB-tree. Preferably, the implementation avoids both unnecessary memory allocations or recursive functions, in order to avoid unnecessary overhead.

We can do this by keeping all elements we traverse in a stack, then "rewind" by going backwards. In that way, we both avoid the overhead of a recursive function call, and we do not allocate new nodes which may potentially be empty. Listing A.6 in Appendix A contains the algorithm for this implementation of RRB-POP.

Direct pop may also break the search step relaxed invariant. Again, as it can only happen with the the rightmost nodes, the invariant will either be satisfied by next concatenation, or it will eventually be resolved through further pops or appends.

5.4 Performance

Next, let us have a look at the performance of the direct append algorithm: In the case of a height increase, we create $h(T) + 2$ new nodes: $h(T) + 1$ from APPEND-EMPTY, and one which is used as the new root. In this case, the runtime is $\mathcal{O}(M \log_{M_l}(|R|)$.

Otherwise, in the worst case, there is enough space in the rightmost leaf node. We must then copy each node and their size table, if they have one. As a copy takes $\mathcal{O}(M)$ runtime, this runtime is $\mathcal{O}(2M \log_{M_l}(|R|)$.

Finally, we may end up having only m copyable nodes out of $h(R)+1$. In that case, the remaining $h(R)+1-m$ nodes will not contain a size table, as they can safely be leftwise dense. Therefore, this total runtime lie, in the worst case, between $\mathcal{O}(M \log_{M_l}(|R|)$ and $\mathcal{O}(2M \log_{M_l}(|R|)$. As such, the worst case runtime is $\mathcal{O}(2M \log_{M_l}(|R|)$.

There is no extra overhead for the relaxed radix search, in contrast to a slice: We always walk the rightmost node, regardless of the size table. As the size of a node is contained in it, the search step takes constant time for each node.

The more appends we use on the RRB-tree with this algorithm, the algorithm gets potentially faster: More and more of the RRB-tree turns leftwise dense, meaning that fewer and fewer size tables get copied. If the RRB-tree had a size table to begin with, we end up only having to copy the size table of the root node after enough appends.

If the RRB-tree R already satisfies $d_{left}(R)$, then this algorithm returns an RRB-tree R' which also satisfies $d_{left}(R')$. This is to be expected, as the algorithm is a generalisation of the persistent vector append.

5.5 Further Optimisations

The function COPYABLE-COUNT needs in the worst case scenario to walk the full tree down to a leaf node. However, it is trivial to deduce how many nodes we must copy

in a leftwise dense subtrie T, given the index i we're inserting at. Recall that, for persistent vector appends, $i = |T|$, consequently the index of the last element in T is $i^- = i - 1$.

Now, if $i = \mathit{cap}(T)$, T cannot contain any more elements, and we can copy zero nodes from this subtrie. However, if $i < \mathit{cap}(T)$, we partition i and i^- into $h(T)$ partitions containing b bits each, $M = 2^b$. The copyable node count in T will be the amount of equal partitions in a row between i and i^-, starting with the partitions containing the most significant bits.

This computation require no tree walking, consequently no cache misses can occur. Assuming that none of the cache lines looked up are replaced while we perform the append, we avoid $h(T) - n_c(T)$ potential cache misses: As we have to rewalk n_c nodes, only the potential cache misses of non-copyable nodes we visit will be unnecessary.

CHAPTER 6

RRB-tree Tail

Section 2.7 discusses the tail of a persistent vector, designed to increase performance on operations operating around its end. It seems reasonable to assume that attaching a tail to RRB-trees will increase their performance, perhaps most notably for appends and pops. This chapter will therefore explain how a tail can be implemented for the RRB-tree, along with elaboration on specific considerations which are not necessary for the persistent vector.

6.1 Implementation

The first implementation consideration for the persistent vector's tail was the benefits of a nonempty tail. Recall that a nonempty tail gives constant time *last*, clearly better than $\mathcal{O}(\log_M |P|)$. However, the nonempty tail made the tail length calculations somewhat more complex. For the RRB-tree, however, we cannot calculate the length of the tail, nor its offset, only based upon the total length of the RRB-tree. As the tree is relaxed, a given size may be represented in different ways, some of which may affect the size of the tail. We therefore use the length field in the tail in an RRB-tree to calculate both its length and the tail offset. To avoid the tail indirection and a potential cache miss, the RRB-tree head also contains the tail length, R_{tl}. The offset calculation is thus simply

$$|R| - |R_{tail}| = |R| - R_{tl}$$

With the exception of length calculations, checking whether an element is in the tail or inserting the tail into the trie then works in the exact same fashion. When the tail is going to be inserted into the trie, we could either use CONCAT or the direct append method explained in Chapter 5. To retrieve a tail from the trie, we could use the direct pop algorithm, described in Section 5.3. It is also possible to traverse the rightmost nodes, pick the leaf node, then slice off the leaf node's length. Although slicing in general may break the invariant at multiple levels, cutting off a single node will in the worst case only break the invariant at the parent of the lowest non-NIL returned, as described in Section 5.3.

Concatenation could easily be done by realising that $A +\!\!+ B$ does not have to modify the tail of B. As such, the seemingly only modification needed to enable concatenation is by inserting the tail of A into the root of A before concatenating it with B. If we use concatenation to insert the tail, a concatenation then becomes

$$A +\!\!+ B = (A_{root} +\!\!+ A_{tail}) +\!\!+ B$$

where the result has the same tail as B.

Slicing could be done by inserting the tail of the RRB-tree into the trie, perform a normal slice, then promote a new tail again. Although easy, this algorithm is clearly inefficient. A better solution checks whether any of the indices are above or equal to the offset of the tail. There are only three cases to consider in this case, assuming the lower index is less than or equal to the upper index:

1. The lower index is equal or greater than the tail offset.
2. The upper index is greater than the tail offset.
3. The upper index lower than or equal to the tail offset.

For case 1, the remaining parts of the RRB-tree is a slice of the original RRB-tree tail, and can easily be copied over to another tail. In this case, the new RRB-tree root will be NIL.

For case 2, a left slice is done on the trie as explained in Section 3.5. In addition, the tail will be partially cut if the upper index is less than the original RRB-tree length.

For case 3, the usual slice operation is done on the trie, and the original tail is discarded. A new tail is promoted as previously explained. This is safe, as both the left and right slice operations operate on indices inside the trie.

The operations mentioned here work perfectly well in isolation from any other optimisation. However, the concatenation and slicing operations may cause big issues in conjunction with the direct append algorithm, if not special care is taken.

6.2 Interference with Direct Append

The direct append algorithm in isolation works on single elements. It is easy to modify the algorithm to consider leaf nodes instead of single elements, but there is a subtle case when working on RRB-trees. Consider a newly sliced RRB-tree R where $M < |R| < 2M$: All the elements are stored either in the root node R_{root}, which is a leaf node, or the tail, R_{tail}. In this case, could $|R_{root}| < M$?

Regardless of whether $|R_{root}| < M$, R will still satisfy the invariant. However, that is not the issue: Recall that the direct append function was designed to store a single element at a time. As such, whenever there is not enough space in the trie, the height of the trie is increased by one. For root nodes without size tables, the direct append function then implicitly assume that the trie is fully dense – which is clearly not the case if $|R_{root}| < M$!

This problem can be generalised: If the RRB-tree is a leaf root, or the rightmost leaf root has a leftwise dense parent, could its rightmost leaf node contain less than M elements? And if so, how should we insert the tail?

One option would be to insert the tail in the next empty slot, then modify its parents to contain size tables. This would, however, create an otherwise leftwise dense trie to a relaxed trie earlier than necessary. Preferably, then, the algorithm copy elements from the tail over to the rightmost leaf node, so that the original rightmost leaf node contains exactly M elements. The remains of the tail node is then inserted. However, this would require the algorithm to perform the same steps next time a tail is inserted, which is clearly inefficient. Instead, we could imagine that the remains is passed back up and reused as tail.

But at this stage, all the different checks makes the algorithm complex and therefore hard to implement correctly. The concatenation algorithm cannot reuse the algorithm used to push down the tail, as it has to push down the tail even if it is not completely full. In addition, the append function would be faster if it could avoid all those checks: If this special case could be eliminated, appending would both be simpler and more efficient.

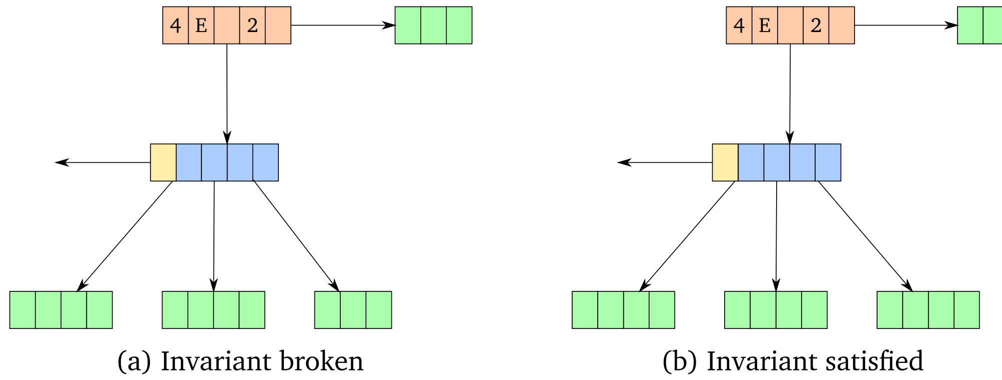
For this reason, we would like to avoid dealing with these problems in the append algorithm, and instead do extra work in the concatenation and slicing algorithm. We do this by creating an invariant which ensures that the append algorithm can work without thinking about these special cases. The initial attempt at an invariant looks as follows

> *In any RRB-tree* R *where* $h(R) = 0$ *or where the parent* T *of the rightmost leaf node satisfies* $d_{left}(T)$*, the rightmost leaf node in* R_{root} *must contain exactly* M *elements if it exists.*

however, we shall see that the nature of the RRB-tree concatenation and slice algorithms enable us to simplify the case to the following invariant:

Invariant 6.1. *In any leftwise dense RRB-tree* R*, the rightmost leaf node must contain exactly* M *elements if it exists.*

We will explain why this is sufficient later in this section, but assume for now that this is the obvious necessary requirement for an RRB-tree.



(a) Invariant broken (b) Invariant satisfied

Figure 6.1: RRB-trees breaking and satisfying tail invariant

Figure 6.1 shows an example of two RRB-trees with a tail. Figure 6.1a breaks the invariant because its leaf node contains 3, not $M = 4$ elements. Figure 6.1b satisfies the invariant as the rightmost leaf node contains exactly M elements.

How can we ensure that the invariant is satisfied by the concatenation algorithm? If we assume both RRB-trees satisfy the invariant, then this is only a concern if the RRB-tree R returned satisfies $d_{left}(R_{root})$. This can only happen in one case:

> Let A^+ be A after tail insertion. A^+ and B_{root} are leftwise dense, and B_{root} can be placed within A^+ without any rebalancing.

Why could it not happen in any other case? Because an RRB-tree that is not leftwise dense must have a root containing a size table. Concatenation will not be able convert this part of the concatenated trie to be leftwise dense, otherwise it would have already happened. Therefore, some part of the concatenated RRB-tree will not be leftwise dense and it follows that the whole tree by definition cannot be leftwise dense. As such, any other case will satisfy the invariant.

For the case mentioned, we have two potential outcomes: We could end up with a leftwise dense trie, or we could end up with a relaxed radix balanced trie. If the latter occurs, the invariant is kept by definition. If the result is a leftwise dense trie, then the rightmost leaf node will either be the rightmost leaf node in B_{root} or A_{tail}. If it is the rightmost leaf node in B_{root}, we satisfy the invariant assuming B satisfies the invariant. This leaves us with the case where A_{tail} is the rightmost leaf node.

There is only one case where A_{tail} satisfy the invariant: If and only if $|A_{tail}| = M$. To handle the other cases, we simply merge the tails, and push down a leaf node if the tails contain too many elements to be put into a single tail. Listing 6.1 shows how this is handled within the RRB-tree concatenation algorithm. The function PUSH-DOWN-TAIL takes the old RRB-tree, the new RRB-tree, and the new tail. It then mutates the new RRB-tree so that the tail it currently points to is pushed down, sets the new tail as new tail, and returns the new RRB.

```
1  function RRB-CONCAT(L, R)
2      ...
3      if R_root = NIL then
4          P ← CLONE(L)
5          |P| ← |L| + |R|
6          if |L_tail| = M then
7              |P_tail| ← |R_tail|
8              return PUSH-DOWN-TAIL(L, P, R_tail)
9          else if |L_tail| + |R_tail| ⩽ M then
10             P_tail ← LEAF-NODE-MERGE(L_tail, R_tail)
11             |P_tail| ← |L_tail| + |R_tail|
12             return P
13         else                                          ▷ |L_tail| + |R_tail| > M
14             P_tail ← LEAF-NODE-CREATE(M)
15             COPY-ELEMENTS(P_tail, L_tail, |L_tail|)
16             COPY-ELEMENTS(P_tail, R_tail, M − |L_tail|)
17             |P_tail| ← |L_tail| + |R_tail| − M
18             t ← LEAF-NODE-CREATE(|P_tail|)
19             COPY-LAST-ELEMENTS(l, R_tail, |P_tail|)
20             return PUSH-DOWN-TAIL(L, P, t)
21         end if
22     end if
23     ...
24 end function
```

Listing 6.1: Code in RRB-CONCAT to satisfy RRB tail invariant.

Next, let us look at slicing, but let us assume that we need to satisfy the initial attempt at the invariant. If we look back at the different slice cases, we see that case 1 can not break the "invariant": Case 1 will always return an RRB-tree where $R_{root} = \text{NIL}$. However, both case 2 and 3 could potentially break the "invariant": We must ensure that the rightmost leaf node contain exactly M elements if its parent is leftways dense.

Assume that the original tree satisfies this condition. In case 2, we will not do a right slice in the trie, only a left slice. In addition, all nodes affected by the left slice will contain a size table, with the exception when the new root is a single leaf node. It is therefore easy to check whether case 2 will satisfy the new invariant or not: If $h(R') = 0$, then $|R_{root}|$ must either be 0 (in which case $R_{root} = \text{NIL}$) or M to satisfy the invariant. Otherwise, we have to distribute the elements over the root and the tail: If $|R| \leqslant M$, we put all the elements in the tail, otherwise we put M in the root and the remaining ones in the tail.

Case 3 has to consider this case, but also another case as well, which happens very infrequently. Assume we have cut through a leaf node, such that the parent of the rightmost leaf node is leftwise dense. Regardless of how many elements there are in that leaf node, we promote it as a new tail. Now, three cases can happen:

1. If there are more leaf nodes in the parent node, they must be fully populated, and we satisfy the “invariant”.
2. If the parent node has no other leaf node, move up to the first ancestor that will still have a child after tail promotion. If the ancestor is leftwise dense, then its rightmost node must be fully dense by definition.
3. If the parent node has no other leaf node, move up to the first ancestor that will have a child after the tail promotion. If the ancestor is not leftwise dense (contains a size table), we are not sure whether the “invariant” is satisfied or not.

The last case here is the problematic one. If the leaf node contains less than M elements, and its parent is leftwise dense, we have broken the “invariant”!

However, the concatenation algorithm has already solved this problem for us: The leftwise dense subtrie must have been created either by the direct append algorithm or a concatenation, as it originally had elements right of itself. If the direct append algorithm was used, then we know that the leftmost leaf node contains exactly M elements, as tail insertion only happens when the tail is full. If the concatenation algorithm was involved, then the rightmost leaf node may contain less than M elements.

But that is fine, because we know that the parent node must have been part of a concatenation. As such, for the parent node to be leftwise dense, it *must* be fully populated: We can therefore not insert more elements into the parent. This logic also applies to any ancestor above it: As the concatenation algorithm completely populates the leftmost node, a node can only retain its leftwise density if either the node is already completely populated, or if the rightmost leaf node is fully dense and the node to concatenate with is leftwise dense. This follows up until we reach the node with the size table.

Why is this not a problem for the direct append algorithm? When direct append walks the rightmost nodes, it discovers that none of the nodes below the one with the size table is copyable: There is not enough space for a new leaf node there. As such, it instead creates a new branch right for the leftwise dense trie in which it stores the leaf node to push down. As the node contains a size table, this is completely fine.

There is only one issue which remains: What happens if the ancestor with the size table is the root node, and the root node now contains a single child? The child will be promoted as root, and is leftwise dense. Now, when the direct append algorithm attempts to push the old tail down, it realises there is not enough space. As such, it creates a new root node *without a size table*, as it assumes the leftwise dense trie is fully dense.

Although height increases in tries happen infrequently, this very specific instance happens even less frequent. As such, we decide to make the slice function responsible for ensuring that the invariant is kept.

While the invariant does make the concatenation and slice algorithms somewhat more complex and slightly more expensive in some specific cases. However, the price paid is insignificant if we perform only slightly more appends than concatenations and slices.

6.2.1 Direct Pop Interference

The direct pop implementation is also affected by the tail. However, the invariant is already covered by the last slice case: The tail promotion in the slice algorithm is equivalent with tail promotion in the direct pop algorithm, meaning that the direct pop algorithm has to consider exactly the same cases: If the height of the trie is reduced as a result of the tail promotion, check if the new trie is leftwise dense. If it is, we have to ensure that the rightmost leaf node contains M elements.

6.3 Display

One could consider generalising the RRB-tree tail to a display in the same fashion as done with the persistent vector in Section 2.8. However, there are disadvantages to displays in RRB-trees, mostly related to the fact that the RRB-tree may not be leftwise dense. In the persistent vector display, finding the lowest common ancestor needed no tree walking: We could find the lowest common ancestor by just comparing the old index and the index to access/update as the search steps were independent. This is not always possible in the RRB-tree due to its relaxation.

To gain the same performance guarantees, the display also needs to contain the upper and lower index of the elements a node contains. This would require an additional overhead on the vector head, and more coordination whenever the display is changed. Whereas it seems likely that this could speed up some specific use cases, the additional overhead could also make it inefficient as a general-purpose RRB-tree. Proper benchmarking is needed to confirm or reject any of these hypotheses. However, it is clear that a display would make the implementation much more complex, leading to higher chance of bugs. Adding a display to the RRB-tree is therefore left as future work, although interesting.

6.4 Performance

Based upon results from the persistent vector tail improvement [9], one could expect the RRB-tree to be at least 2 to 5 times faster. It is not unfeasible that the performance benefits may be even greater, as the access algorithm require both more memory accesses and has more overhead. However, this assumes that the environment is not a factor. The persistent vector results are from a different environment – the JVM – which likely affects the results to some degree.

Chapter 7

RRB-tree Transience

7.1 Introduction

On its own, adding in a transient version of the RRB-tree seems not only like a good way to increase performance, but also very easy to implement. Recall that transience on the persistent vector increased its append performance significantly. As the RRB-tree algorithms may have to copy size tables as well, they would potentially have to copy an additional $h(R)$ structures compared to the persistent vector implementation. Thus, if allocating memory and copying is one of the bigger bottlenecks, we should see considerable performance increases.

However, implementing transience on the RRB-tree could require us to somewhat extend the definition of a transient. Recall that Definition 4.5 implicitly states that the first argument to a function defines whether it is a reading function or a writing function on the transient $\alpha^!_\tau$. But what about confluent data structures, like the RRB-tree? Their concatenation function takes two arguments in, both of which are used to return an "updated" value back. This is problematic: If we have the function

$$+\!\!+ : \alpha_\tau \times \alpha_\tau \to \alpha_\tau$$

then what should the corresponding transient function be? It could be either one of these, or even both:

$$+\!\!+^!_w : \alpha^!_\tau \times \alpha^!_\tau \to \alpha^!_\tau$$
$$+\!\!+^!_w : \alpha^!_\tau \times \alpha_\tau \to \alpha^!_\tau$$

It also depends on how one philosophically decide to define $+\!\!+$. Does it read the second element, or does it "update" both of the values? If it only "updates" the first element, then the type signature $\alpha^!_\tau \times \alpha^!_\tau \to \alpha^!_\tau$ is confusing: The second $\alpha^!_\tau$ is then borrowed, not invalidated. Denoting a function writable or readable through f_w and f_r is clearly not sufficient for confluently persistent data structures, as different arguments may either be borrowed or invalidated. As such, one should preferably be

able to specify whether the type is borrowed or invalidated on the type of the input argument.

Another problem with Definition 4.5 is the specification on how many times a single transient can be used:

> *At any point in time, only a single function $f^!_w$ or $f^!_r$ can operate on a specific transient data structure $P^!$.*

However, assume we have the following application of the transient function $+\!\!+^!$, where both arguments are invalidated:

$$a +\!\!+^! a \mapsto a'$$

Is this application legal? According to Definition 4.5, yes. However, from an implementation point of view, this could potentially return erroneous results: Mutating any part of a will necessarily change both input values, unless parts of them are copied beforehand. Clearly, the definition needs to also ensure that the the transient is only used once in a function call.

There is another option, which is both easier and more sensible for this thesis: To not provide operations for transients with more than one argument of type α_τ when $\tau \neq \alpha_\tau$. Indeed, the eagle eyed reader may have noted that Definition 4.5 explicitly states that a write function is only defined as a write function if $A_i \neq \alpha_\tau$ for all i, meaning that confluently persistent functions are not, by the definitions, write nor read functions.

As the main goal of this thesis is to provide efficient appending, improving concatenation speed is of a less concern and could rather be left as future work. Additionally, the implementation of transience we will use will check whether the function call is called in the thread owning the transient. The parallel grep implementation further explained in Chapter 8 will only make a single RRB-tree per thread, and as such, any transient concatenation cannot be performed on two transients from two different threads. Finally, as there will be very few concatenations compared to appends in a real system, the actual benefits of a transient concatenation may not be as important for overall performance.

7.2 Implementation

One of the results of the transient constraint is that the RRB-CONCAT function cannot be used by RRB-PUSH. As a result, any transient implementation *has to* use direct appending in order to be able to support pushing and popping. While not directly a problem, it means that it not possible to measure the performance of transience in isolation.

Another consideration is *what* one should mutate. In the persistent vector, all nodes could be transient. An RRB-tree may contains size tables as well. It seems reasonable to assume that mutating size tables instead of creating new ones for every trie insertion would speed up their performance. Consequently, the transient implementation also adds an ID to size tables and mutates them when it is safe to do so.

As the implementation is done in C, there is no way to ensure that the transients are not used in multiple function calls and not misused at compile time. To ensure some sort of safety, the same constraints used in Clojure is incorporated: Only the thread owning the transient can perform update operations on it, and the *persistent*!-call will set the transient's ID to NIL.

If the implementation uses a tail, then the tail will – as the Clojure's persistent vector – be expanded to avoid unnecessary copying. In addition, the same technique will be used on all size tables, internal nodes and leaf nodes: While not problematic for a persistent vector implementation, an RRB-tree might decide to keep nodes as small as possible. After the *persistent*!-call, the tail will be compressed, although nodes and size tables in the trie will not. As a result, the RRB-tree conversion operations have the same performance guarantees as the persistent vector implementation: $\mathcal{O}(\mathrm{M})$.

There are not any other major differences, and the optimisation does not interfere with other optimisations discussed in this thesis. The algorithms are modified as explained in Chapter 4, and use a similar version to the TRANSIENT-CLONE algorithm presented in Listing 4.1. The only notable "difference" is that many values which from the current version of RRB-tree in the algorithms, is saved on the stack before the destructive modification happens.

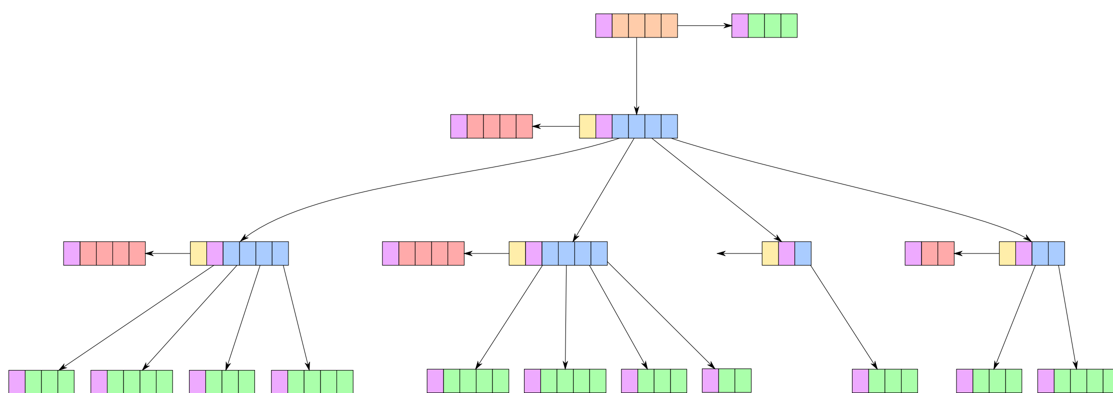


Figure 7.1: RRB-tree with tail and transient

CHAPTER 8

librrb and pgrep

In order to measure the optimisations explained in the last three chapters, they have to be implemented and benchmarked. This require two things: An implementation of the data structure itself, and some way to benchmark the implementation. The result is a C library, `librrb`, which provides a garbage collected variant of the RRB-tree. In addition, the parallel program `pgrep` – a naïve parallel version of `grep` – is implemented to measure the performance of the RRB-tree.

8.1 librrb

librrb is an implementation of the RRB-tree in C with the option to configure the branching factor size to some power of two, and turn on and off different the optimisations which has just been described. For configuration, the automake tools are used. Appendix D contains a list of functions defined in the library, along with its effective runtimes and what they do.

The different options are enabled and disabled by the `#ifdef` and `#ifndef` preprocessor macros in C. As there are 3 different optimisations which all interact with each other in some way, the code can be somewhat difficult to read as there is a considerable amount of permutations. Code for non-transient functions is contained in `rrb.c`, and the transient counterparts lies within `rrb_transients.c`. All options are listed in Appendix D, with a short description on what they do.

8.1.1 Loop Unrolling

As an attempt to increase performance, some functions within `librrb` is manually loop unrolled: `rrb_nth` and `rrb_update`. This is done by converting the for loop into a switch statement, in which the first statement is the highest possible shift, the second the second highest shift, and so on.

Normally, this could be done by hand. However, as the branching factor changes, so does the shift and the maximal height. As such, another approach has to be used to

dynamically change the unrolling. The idea is to utilise the C preprocessor capabilities in order to emit the correct amount of iterations, based upon the max treie height . However, the C preprocessor does not allow `#define`s to recursively call themselves. Therefore, we abuse `#if` statements and `#include` preprocessor flags: Using them correctly, we can compare an iteration value with the desired amount of iterations. If the iteration value is lower than the desired amount, we emit the contents of "loop body" macro, increment the iteration value and recursively include this file again, simulating an actual loop. Otherwise, we undefine the definitions used and stop. This trick is used in the `src/unroll.h` file.

```
    switch (RRB_SHIFT(rrb)) {
#define DECREMENT RRB_MAX_HEIGHT
#include "decrement.h"
#define WANTED_ITERATIONS DECREMENT
#define REVERSE_I(i) (RRB_MAX_HEIGHT - i - 1)
#define LOOP_BODY(i) case (RRB_BITS * REVERSE_I(i)):                          \
      if (current->size_table == NULL) {                                    \
        const uint32_t subidx = (index >> (RRB_BITS * REVERSE_I(i)))      \
                                & RRB_MASK;                                 \
        current = current->child[subidx];                                   \
      }                                                                     \
      else {                                                                \
        current = sized(current, &index, RRB_BITS * REVERSE_I(i));        \
      }
#include "unroll.h"
#undef DECREMENT
#undef REVERSE_I
    case 0:
      return ((const LeafNode *)current)->child[index & RRB_MASK];
    default:
      return NULL;
    }
```

Listing 8.1: Example of manual loop unrolling, from `src/rrb.c`.

Listing 8.1 shows the usage of the manual unroll implementation. `sized` represents the relaxed radix search algorithm, the `WANTED_ITERATIONS` definition specifies the total amount of iterations wanted, and the `LOOP_BODY` definition defines the loop body. The decrement definition is used as the last case within the switch statement is a special case and has to be handled differently.

8.1.2 Threading

To ensure that the transients are called in a single thread, `librrb` must be able to get the current thread's ID and be able to compare it with another thread ID. In this way, the transient RRB-trees can store the thread they were created in and compare it with the thread transient functions is called in.

The pthread library is used to receive and compare thread IDs, as it is available in almost all Unix-like operating systems that are POSIX-conformant. However, as the only requirement for `librrb` compliance is returning and comparing thread IDs, it not a problem to replace pthread with another library of choice[1]. The only required change to replace pthread with some other library is to edit macros defined within `src/rrb_thread.h`.

8.1.3 Garbage Collection

As the RRB-Tree is a persistent data structure, subsequent "modifications" use shared structure for better performance and memory usage. As such, some sort of garbage collection must be performed to avoid memory leakage.

A reference counting algorithm could be a viable option for programs where the memory overhead of a garbage collector would be too large. However, Jones et al. [23, pp. 58-60] argues that a naïve reference counting algorithm *"is impractical for use as a general purpose, high volume, high performance memory manager"*. As parts of the goal of this thesis is to implement a general purpose, high performance data structure implementation, implementing the reference counting by hand seems to not be a fruitful approach. Of course, more efficient reference counting garbage collectors provided as libraries could be used.

Another option would be to use the `shared_ptr` template provided by the Boost library for C++. However, the Boost library only enforces atomicity when manipulating reference counts, meaning that concurrent threads can modify `shared_ptr` instances simultaneously. This is sufficient for nonconcurrent programs, but poses additional implementation details and additional overhead for parallel programs, where concurrency is inevitable. `shared_ptr` has other complications as well, such as its inability to properly support `const` pointers[24].

Using a garbage collector library with support for threads is therefore likely a sensible choice. The C and C++ Boehm-Demers-Weiser conservative Garbage Collector[2] (Boehm-GC) was chosen as it is both stable, robust and well tested. It is easy to set up, as the only work to be done is to replace allocation calls, remove deallocation (`free`) calls, and add in specific define flags to the garbage collector in executable programs.

Many of these features are desirable, but the most important one is that it is compatible with the pthreads library "out of the box", in contrast to naïve reference counting algorithms and use of `shared_ptr`. For this reason, the implementation itself does not have to handle locking of pointers, nor freeing memory – this is left to robust and well tested libraries instead.

[1] Provided the thread library works with the garbage collector used.
[2] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

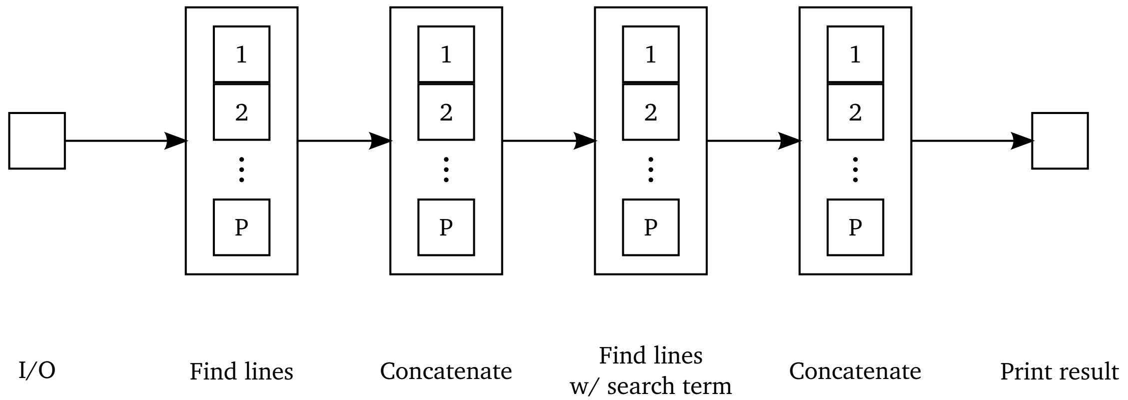


Figure 8.1: Different computation stages in the parallel grep implementation.

However, depending on a garbage collector has disadvantages. The Boehm Garbage Collector is a heavy dependency, and for optimal use, the whole program along with libraries should use it. Using libraries which uses `malloc` and `free` from the standard library can be tedious: As Boehm-GC is unable to see pointers inside memory allocated with `malloc`, data may be garbage collected if not consideration is taken when inserting pointers into structures. Additionally, depending on a specific garbage collector depends on its performance, which may not be optimal.

For these reasons, the Boehm-GC is not tightly coupled with `librrb`: All code includes the header file `src/rrb_alloc.h` containing macros related to memory allocation. This makes it is easy to replace the Boehm-GC with another garbage collector option if necessary.

8.2 pgrep

Grep is a program which takes a pattern to match against, one or more files, and prints all lines matching that pattern. The implementation of grep using RRB-trees, `pgrep`, program is a naive implementation of grep, where the computation is performed in parallel. The pgrep implementation only matches fixed strings, and the matching algorithm itself is not intended to be efficient, but rather mimic real world parallel programs which uses well-known optimisation patterns [25].

8.2.1 Computation Stages

The parallel grep implementation with its different computation stages is presented in Figure 8.1 . We first load in the file to read from disk, which is done by a single thread. After that, we create P threads which find line starts and ends in a contiguous part which is $\frac{1}{P}$ of the file. As all threads have to keep their own local list, the next step is to concatenate these lists together to one list. This is done in parallel through a *reduction*. In the search phase, the lines are evenly distributed: All P threads process a

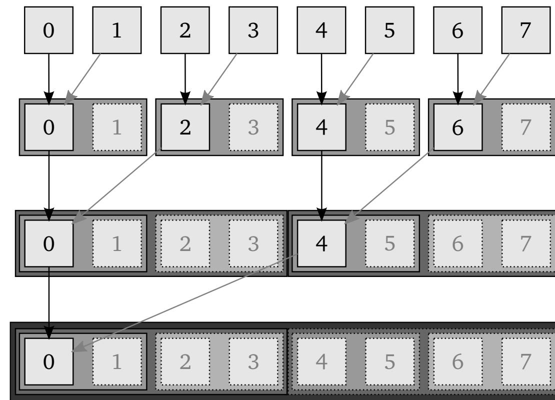


Figure 8.2: A parallel concatenation with $P = 8$ threads.

contiguous part of the lines, equal to $\frac{1}{P}$ of the total line count. After this step is done, another reduction is performed to get all the matched lines in a single list. Finally, this list is printed by a single thread.

As mentioned, the program utilises optimisation patterns explained by Stratton et al. [25]. The line find and line search step utilises privatisation, and the first concatenation step is performed so that regularisation can be done. Additionally, leaving the line intervals in ascending order rather than out-of-order helps cache utilisation, and consequently performance.

The reduction step is visualised in Figure 8.2: The idea is that one concatenate the lists together in parallel by pairing the remaining threads in order. The pairs concatenate their lists, and when they are finished, the right thread in the pair leaves, while the left thread keeps the result and is paired with another thread. This process is performed until only the first thread remains.

All parallel stages could be done in a single step, removing the need to stop and start up new threads. However, such an implementation would restrict timing possibilities: The different stages may overlap and cause calculation or concatenation time to be smaller/larger than the actual count. Whereas overall performance degrades, this design choice should not make any data structure faster or slower compared to other structures.

8.2.2 Work Distribution

The work distribution schedule known as *static chunking*[26], giving each processor a $\frac{N}{P}$-sized chunk, is optimal when the amount of work per element takes the same amount of time. However, in many situations, the workload per element is not known a priori, and all other processors may end up waiting on a single processor finishing its chunk. To ensure better load balancing, adaptive stealing can be performed through frameworks such as the Scala Parallel Collection Framework[27] or Cilk[28]. As

adaptive work stealing algorithms restrict timing possibilities, static chunking was chosen as work distribution schedule.

```
// barrier before merging result, to avoid race conditions
uint32_t sync_mask = (uint32_t) 1;
while ((own_tid | sync_mask) != own_tid) {
  uint32_t sync_tid = own_tid | sync_mask;
  if (thread_count <= sync_tid) {
    // jump out here, finished.
    goto concatenate_cleanup;
  }
  pthread_barrier_wait(&barriers[sync_tid]);
  // concatenate data
  concat_and_replace_own(intervals[own_tid], intervals[sync_tid]);
  deallocate_if_needed(intervals[sync_tid]);
  sync_mask = sync_mask << 1;
}
pthread_barrier_wait(&barriers[own_tid]);
```

Listing 8.2: Concatenation algorithm.

Listing 8.2 shows the concatenation algorithm used for the pgrep implementation. Whereas this is a non-optimal concatenation strategy for the array list, it represents a more realistic benchmark for scheduling schemes more sophisticated than static chunking. In fact, adaptive load balancing schemes are likely to perform more concatenations than the algorithm in Listing 8.2, as they usually split up work in more chunks.

Part III

Results, Discussion and Conclusion

Chapter 9

Results and Discussion

9.1 Practical

The performance of the RRB-tree implementation and the different optimisations were evaluated through repeated runs of the parallel grep implementation, and is compared to the same implementation using mutable array lists. As all computing parts have overhead unrelated to list operations, a third program only counting the number of lines is used to compute an approximation for the overhead. To avoid performance differences due to disk buffering or similar cache policies, 3 warmup runs were performed, followed by 50 benchmarked runs. A single run consists of running all 3 programs sequentially, hence program execution is interleaved.

As the different optimisations may yield different performance, five different optimisation permutations were measured:

1. The naïve, original implementation, which uses no optimisations.
2. Direct appends, denoted (D).
3. Direct appends plus tail, denoted (DT).
4. Transients and direct appends, denoted (D!).
5. Transients, direct appends and tail, denoted (DT!).

The transient implementation must, as described in Chapter 7, use direct append, which is why there is no transient test without direct appends. The tail implementation had to use the direct pop implementation in order to support pops. Although technically possible, it was hard to decouple the direct pop and direct append implementation. Therefore, the permutation only using the tail automatically included the direct append implementation, and gave equivalent results with the (DT) permutation.

All of the mentioned permutations were run with the 50 benchmarked runs interleaved with the array list implementation and the overhead implementation. Additionally, the branching factor effect was measured from $b = 2$ to $b = 6$, $M = 2^b$, with all optimisations (DT!) turned on.

Name	Time (ns)	Time - OH (ns)	Relative	Relative - OH
Overhead	667089430.58	0.00	N/A	N/A
Array	814386692.12	147297261.54	1.000	1.000
RRB - DT!	825221786.44	158132355.86	1.013	1.074
RRB - !	840680019.90	173590589.32	1.032	1.179
RRB - DT	977754573.14	310665142.56	1.201	2.109
RRB - D	1303835118.64	636745688.06	1.601	4.323
RRB - naïve	2494461829.48	1827372398.90	3.063	12.41

Table 9.1: Line search times

The results presented is the median of all 50 runs. All measurements had a median roughly equal to the mean and a very small interquartile range. The only exception is the concatenation phase, which had a very large interquartile range further discussed in Section 9.3.

All runs were run with same search terms on all the public activity on GitHub from January 1. 2013 to January 10. 2013, fetched from `http://www.githubarchive.org/`. This data is 2.4 GB large uncompressed, containing 1.5 million lines. Search terms used is presented in Figure B.1.

For memory benchmarks, programs measuring total memory usage during the search phase and concatenation phase were designed. As neither the array list nor the RRB-tree utilise randomness, a single run per search term was sufficient to determine memory usage.

The benchmarks were performed on a PC with a 4-threaded 2.0 GHz Intel Core i7-3667U processor and 8 GB of memory, running 64 bit Debian Jessie (Linux 3.13). The CPU has a 32 kB L1 cache, 256 kB shared L2 cache and a 4096 kB shared L3 cache. Programs were compiled with Clang, version `3.4.1` from the `http://llvm.org/apt/wheezy/` Debian repository. The pthread implementation used is NPTL 2.17, along with Boehm GC version 7.2. All code was compiled with the flag `-Ofast`, and was ran at runlevel 1 to minimise thread scheduling and interfering programs. All benchmarks use 4 threads.

9.2 Append Time

As append performance was the intended goal to improve, we expect that the running time would be significantly decreased for the line search phase and the search filtering phase. Figure 9.1 shows the total time used per optimisation, from lowest time to highest. As all runs walks over the same file, there is no difference based upon the query given to pgrep.

Unsurprisingly, all optimisations combined yields the best performance. However, the total runtime compared to the array implementation is unexpectedly good: The array

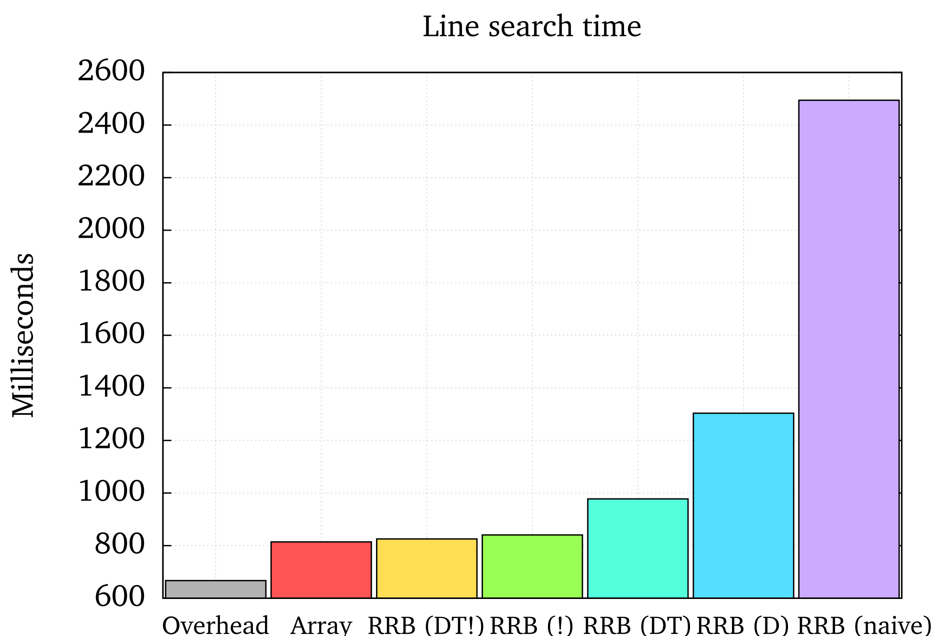


Figure 9.1: Time used in line search phase

implementation uses, on average, about 814.4 milliseconds (msec). A transient RRB-tree with a tail and direct append uses around 825.2 msec. If we remove the overhead which is on 667.1 msec, the RRB-tree version is roughly 7.3% slower. In contrast, the unoptimised append version of the RRB-tree uses almost 2.5 seconds in this stage, over 12 times slower than the array implementation. Table 9.1 shows all results, where RELATIVE represents the relative time compared to the array implementation, and RELATIVE - OH represents the relative time when overhead is removed.

From direct appending to tail usage, we see that the total time required decreases from 1304 msec down to 977.8 msec. If we remove the overhead on 667.1 msec, we get that the tail implementation over doubled the performance of appending in this situation. This seems to fit with earlier measurements on the impact a tail has on the persistent vector append performance, which had time used decreased by a factor between 2 and 5 [9].

Informal benchmarks by Hickey indicates that transients can improve the performance of persistent vector appending with a factor on almost 10 [10]. The relative performance increase from an RRB-tree using direct appends and tails to its transient version is not as good, almost doubling in this phase of the program.

It is not impossible that additional computations and extra checks on an RRB-tree slows down its transient performance. However, it is more likely that the opposite is true: That the `-Ofast` flag improves non-transient performance by a considerable factor. As the measurements of transient vectors were done on the JVM, which has a completely different runtime and memory layout, it is not unreasonable to assume that in this specific situation, the JVM may perform worse. Finally, the overhead

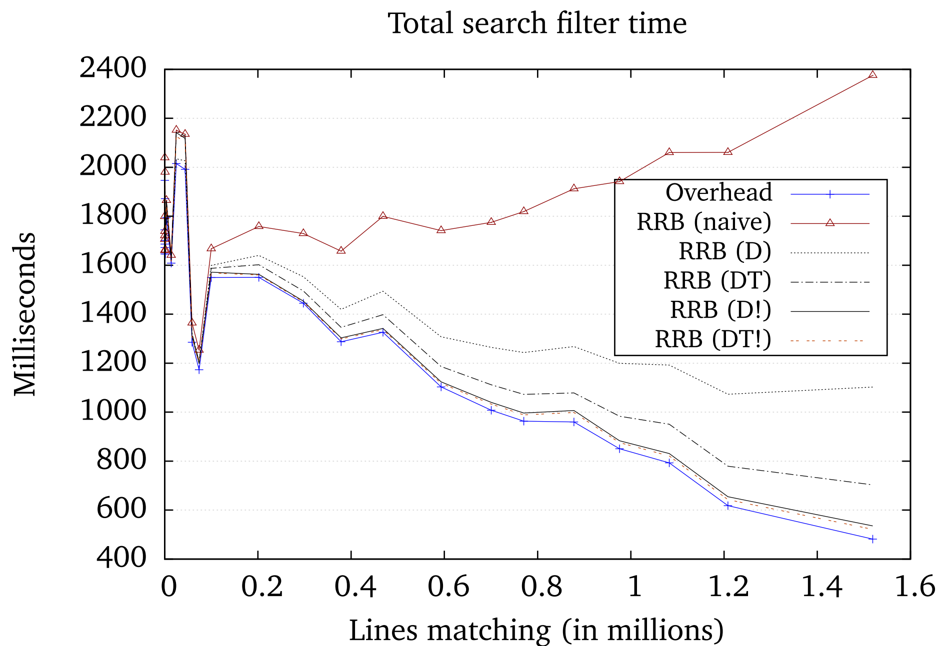


Figure 9.2: Total time used in search filtering phase

implementation runs almost 150.0 msec faster than the array list. It may be that the optimisation step on the overhead implementation performs better by taking shortcuts not viable when the line intervals has to be stored. As a result, the relative performance increases benchmarked may be higher than the actual performance increases.

From the results from the line search phase, one might expect that the search filtering phase has similar results. However, this phase – visually represented in Figure 9.2 – has a higher relative runtime than the line search phase, even more so when the overhead is removed. The array list implementation benchmarks are not shown, as it follows the overhead plot.

There are many factors which might contribute to this additional overhead. One of the more important factors is the fact that the "overhead" calculation is an estimation of the overhead. Recall that we must store the line intervals *somewhere*. As the storage with the least overhead when performing read-only instructions is an array, it is used to make an estimate of the overhead. As such, the array list implementation only times the time to insert an element, and the infrequent reallocations. The relative overhead is therefore not the best fit to measure the work being done not related to the data structure.

While this could explain the rather low time on the array implementation, it does not explain why the RRB-tree require more time. The relative time for the transient RRB-tree with tail is here 8.6%, in stark contrast with the previous 1.3%.

The cause with the highest impact factor is likely the RRB-tree access pattern: While

Name	Time (ns)	Time - OH (ns)	Relative	Relative - OH
Overhead	479452023.68	0.00	N/A	N/A
Array	480092793.46	640769.78	1.000	1.000
RRB - DT!	521358789.04	41906765.36	1.086	65.40
RRB - !	532751545.20	53299521.52	1.110	83.18
RRB - DT	703297313.18	223845289.50	1.465	349.3
RRB - D	1100903284.24	621451260.56	2.293	969.9
RRB - naïve	2381177126.80	1901725103.12	4.960	2968

Table 9.2: Search filter time for 1.5 million matches

the RRB-tree has good access times, it is relatively high compared to an array. Additionally, the trie is walked for every lookup, which is not ideal. Preferably, one should use an iterator which uses the ideas of a stack/"display", elaborated in Section 2.8 and further discussed in Section 6.3.

Another factor involved may be branch prediction: Notice that in Figure 9.2, from around 0.2 to 0.5 million lines, the difference between the overhead and the transient RRB-tree is almost negligible. If the only contributing factor was RRB-tree access, one would expect the gap to be relatively constant. This is not the case, which might imply that the string search function is able to be branch predicted in the array list runs, but not within the RRB-tree runs due to its additional overhead.

Finally, the search filtering happens much later in the program. As a result, the garbage collector is more likely to collect garbage within this period. The array list and overhead implementation does not use any garbage collector, and as a result is not impacted by this.

9.2.1 Branching Factor Effect

Figure 9.3 presents the results from different branching factors in the line search phase, where all optimisations were enabled. Branching factors did not affect the performance as significantly as the persistent vector branching comparison in Figure 2.7. This is to be expected, as transient data structures do not have to copy any internal nodes in this specific situation. In addition $(M-1)/M$ of all append operations run in actual constant time. In contrast, the persistent vector operations has to walk the tree and copy internal nodes.

Nevertheless, Figure 2.7 and Figure 9.3 resemble each other, in that both match the line $f(x) = (ax - b)^2 + c$ for different a, b, c. The branching factor $M = 2^b$ where $b = 5$ seems to give the best performance. This may indicate that, for transients with tails, the branching factor $M = 2^5$ gives a very good ratio between node size and constant time appends. However, the results may vary between different environments, most particularly when there are differences in memory allocation times.

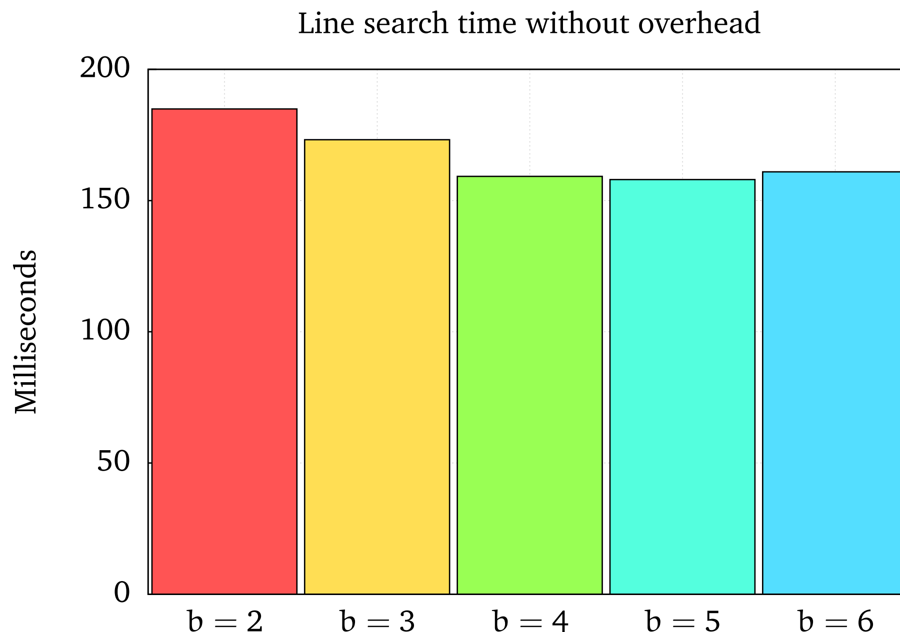


Figure 9.3: Branching factor effects on line search phase

Figure 9.4 shows the different branching factor performance for the search filtering phase. As in the line search phase, $b = 2$ and $b = 3$ is noticeably slower than branching factors $b = 4$, $b = 5$ and $b = 6$. Although the best branching factor varies based upon the number of lines to append, the overall results indicate that higher branching factor is better.

An RBB-tree has better access times with higher branching factors, as it has a lower height. One may thus expect that the overhead related to access lookup will be lower for those runs. An iterator may decrease this overhead, and could make the RRB-trees with lower branching factor perform better.

Although the variation in these tests is small enough to suggests that $b = 5$ is the currently optimal choice, the differences between $b = 4$ up to $b = 6$ is very small. It may be possible that a different hardware setup or different environment could give different results. However, both Clojure and Scala uses $b = 5$, which indicates that this may be a good branching factor for the JVM as well.

9.3 Concatenation Time

While append times are the most important factor within this thesis, they should not severely impact the performance of other functions. From a theoretical perspective, neither slicing, accesses or updates should be impacted. In fact, if anything, they should be faster, as the tree is likely to be more compact.

For concatenation, the answer is not immediately obvious. As all the vectors are built up using direct appending on empty RRB-trees, they will be all be leftwise dense.

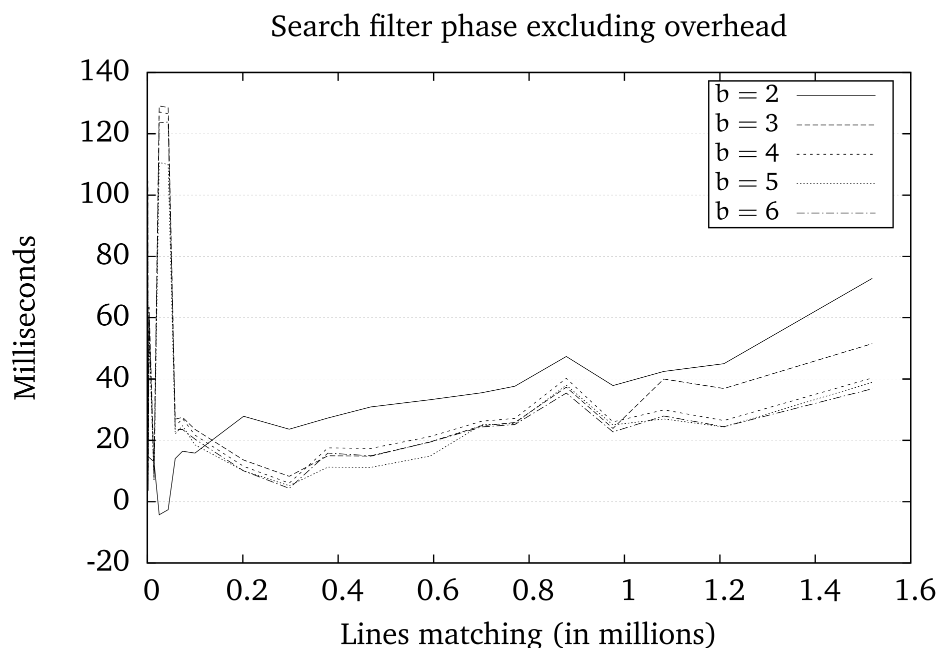


Figure 9.4: Branching factor effects on search filter phase

Therefore, the first concatenations will require no rebalancing: The lowest level will, by default, require no concatenation. The second level contains two nodes, both of which have length $\|T_{opt}\|$. Therefore, none of the nodes have to be rebalanced, as their combined optimal size would at worst be their sizes minus one. This logic follows up to the top node, which may grow if there is not enough space in the top node. As a result, the first concatenations will be fairly cheap. However, the following concatenations will entirely depend on the sizes of the previous vectors, and the results may vary based on that.

Additionally, we may expect the concatenation time for tail implementations to be somewhat higher than the non-tail implementations, simply because the tail implementations have to do more work. The transient RRB-trees should not have considerably different performance measures compared to their persistent counterparts: The only difference would be where the trie nodes are laid out in memory.

Figure 9.5 shows us that the concatenation times does not change considerably. Although not shown directly in the figure, the concatenation time for all points has a very high variation: The difference between first to third quartile is very high, and maximum and minimum runtimes were roughly equal for all runs. See Figure C.1 in Appendix C for more information. Additionally, the same RRB-trees are concatenated over and over, so concatenation times may not be representative for the general RRB-tree concatenation of their total size. Finally, the runtime of a single concatenation is very small: The overhead, which is assumed to be constant, changes, even though the concatenation step is simply a parallel addition reduction. As a result, it is hard to conclude with any trends between the different optimisations.

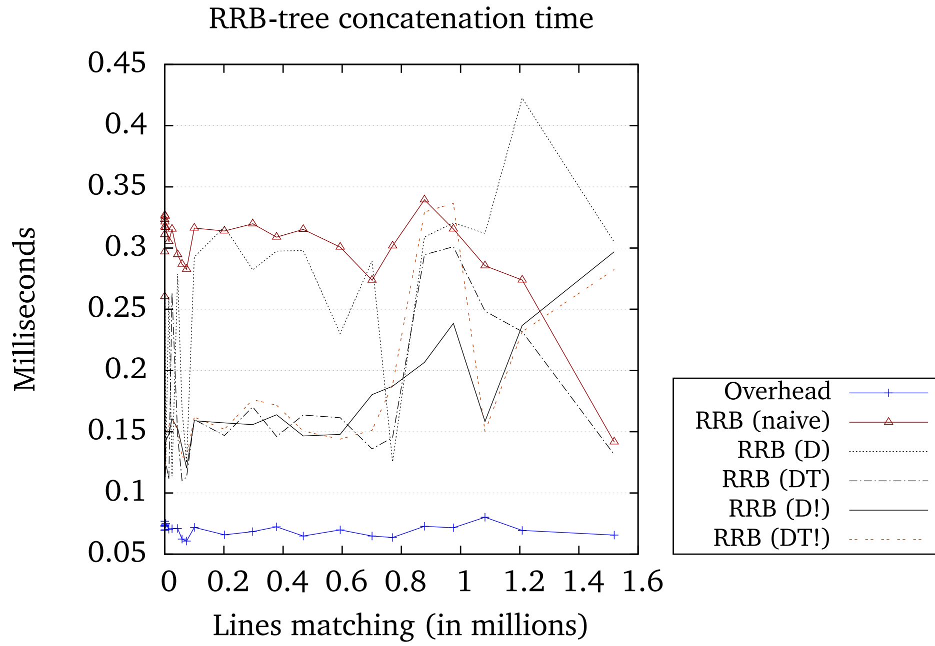


Figure 9.5: RRB-tree concatenation time

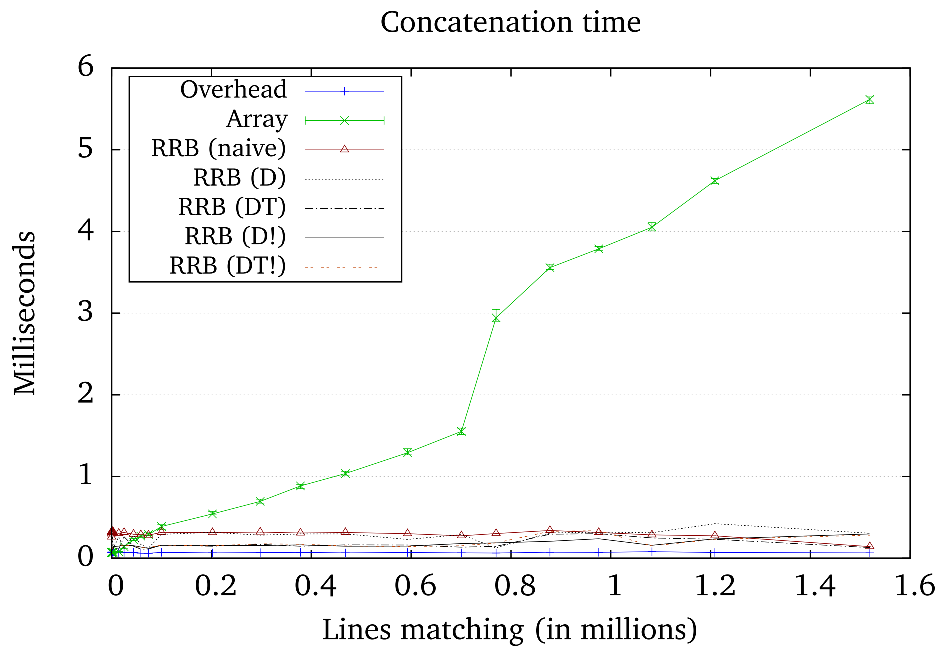


Figure 9.6: Concatenation time with array

However, it is evident that the runtime is relatively the same compared to the array list implementation. Recall that the array list implementation uses linear time for concatenation, backed up by Figure 9.6. At 1.5 million matches, the 5.5 milliseconds copying is 21 times slower than the 0.26 millisecond concatenation for an RRB-tree.

The rather drastic increase in time used by the array list implementation near 800 000 elements is most likely related to the L3 cache not being able to store the whole array.

9.3.1 Branching Factor Effect

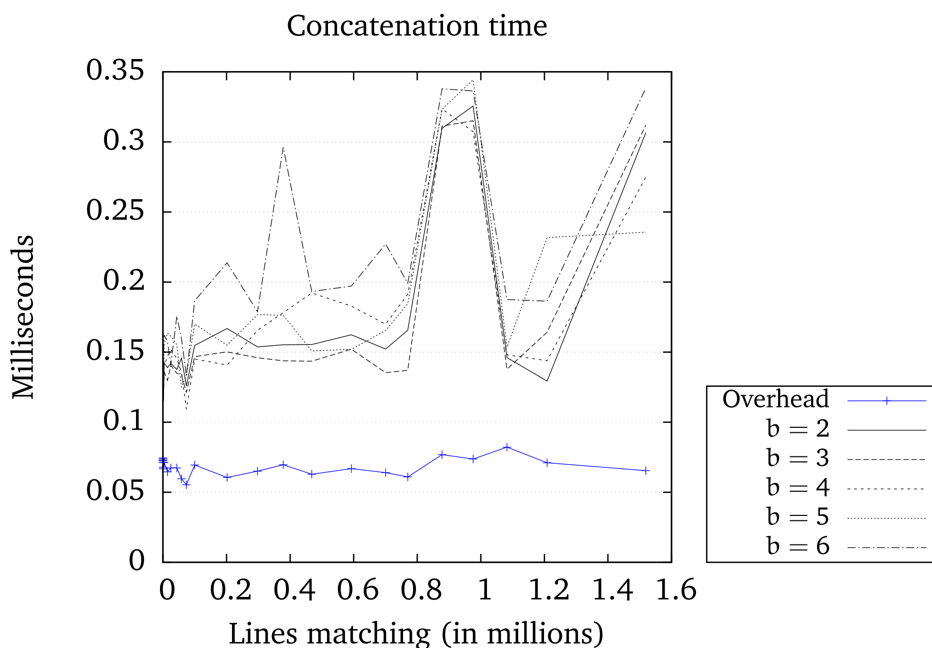


Figure 9.7: Branching factor effects on concatenation

The branching factor over the transient RRB-tree utilising direct append and tails is shown in Figure 9.7. As with the previous concatenation plots, the variance is not low enough to conclude anything. However, the plotted lines have the same trend, so future work could attempt to properly benchmark concatenation with different inputs and different branching factors.

As clearly seen in the figure, there is a huge spike from 0.8 to 1.0 million elements. The spike is also visible in Figure 9.5 as well, and is most likely there because of memory allocations and potentially quick garbage collections: The heap size is not explicitly set, and defaults to 65 536 bytes. After the line search and concatenation phase, the GC has increased the heap size up to 27.4 MB. The next heap increase happens if more than 0.45 million lines matches in the search filtering phase, in which case the heap size increases to 35.8 MB. However, the next heap increase does

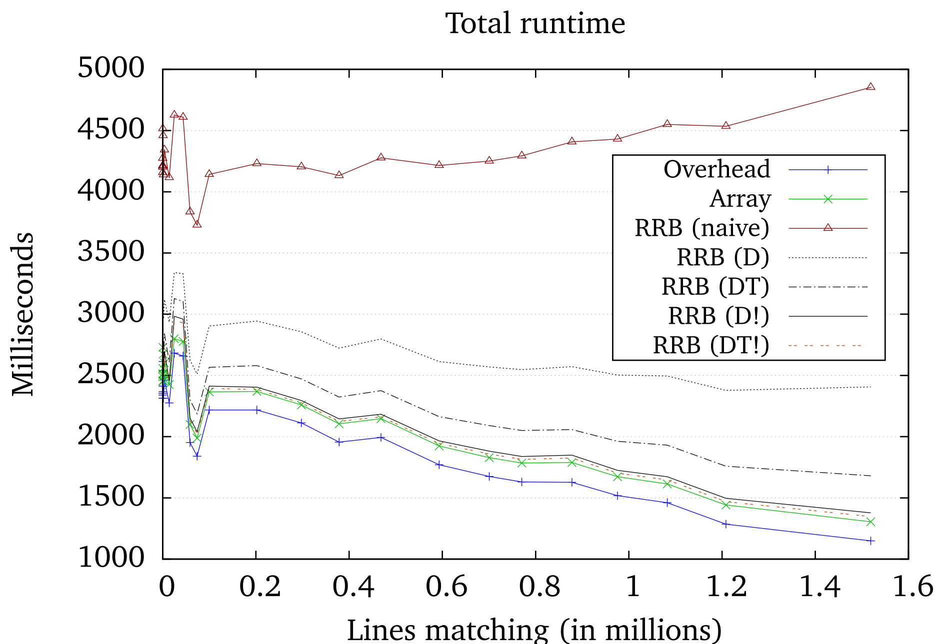


Figure 9.8: Total running time for `pgrep`, excluding I/O

not happen before just over 1 million lines matches (up to 44.2 MB). Therefore, the memory allocator either has to use more time to find a free memory block, or has to reclaim one. This is faster when there is more free space available on the heap, and could explain the additional work being done at these levels. As there is a spike at 1.5 million matching lines, one would expect the same happens there as well. As the heap still remains at 44.2 MB in size at 1.5 million matches, this seems like a reasonable cause.

9.4 Overall Time

The overall runtime, which is the aggregation of all the different phases excluding I/O, is presented in Figure 9.8. As the line and search filtering phases dominates considerably compared to the concatenation stage, it is to be expected that the running times from those program stages defines how much faster or slower an implementation is.

Considering the performance of the line search and search filtering stage has already been discussed, there is no need to discuss the graph itself as we would reiterate the points from those parts. A perhaps more interesting question is whether we can consider the RRB-tree a data structure with *effectively constant* time append. Figure 9.8 shows a graph where the running time for transient RRB-trees does not seem to increase with more matches. Data from Table 9.1 also supports this claim, where the speed of transients is comparable to dynamic arrays, which has an append operation running in amortised constant time with a very low constant factor.

NAME	TIME (NS)	TIME - OH (NS)	RELATIVE	RELATIVE - OH
OVERHEAD	1146721687.44	0.00	N/A	N/A
ARRAY	1303207989.18	156486301.74	1.000	1.000
RRB - DT!	1347461353.48	200739666.04	1.034	1.283
RRB - !	1375463781.68	228742094.24	1.055	1.462
RRB - DT	1681408978.70	534687291.26	1.290	3.417
RRB - D	2404692476.12	1257970788.68	1.845	8.039
RRB - NAÏVE	4860573069.44	3713851382.00	3.730	23.73

Table 9.3: Total runtime for 1.5 million matches

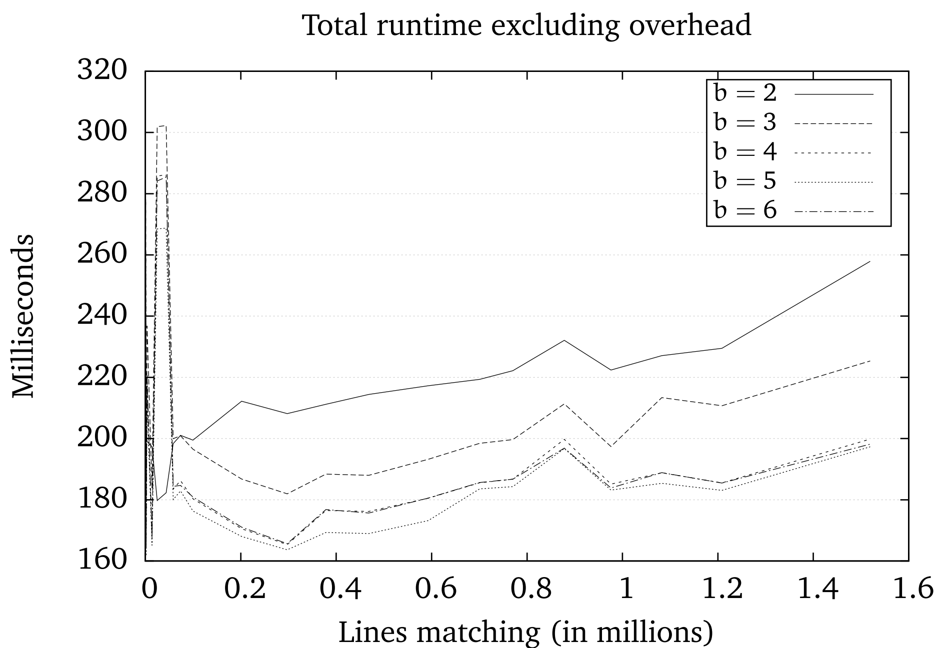


Figure 9.9: Branching factor effects on total runtime

However, calling the append times on a *persistent* RRB-tree for effectively constant *with this environment* seems unreasonable: The eagle-eyed reader will notice an increasing gap in Figure 9.8 between the persistent RRB-tree utilising direct appends and tail, and the transient version. Most likely, this is caused by slow memory allocation and, to some extent, copying. Other environments, such as the JVM, may have more efficient memory allocation semantics, and as a result may have performance characteristics which seems more like constant time.

9.4.1 Branching Factor Effect

Figure 9.9 presents the branching factor impact on the total runtime, excluding overhead. As seen, $b = 5$ gives best overall performance. However, as previously noted, the difference between $b = 4$, $b = 5$ and $b = 6$ is small.

As discussed in Section 9.2.1, the branching factor impacts the height of the RRB-tree containing the line intervals, and thus the iteration over it. An iterator using a display will probably increase the gap between $b = 5$ and $b = 6$ further. However, as new hardware will come out, it is also likely that $b = 6$ will be the most performant in the not too distant future.

9.5 Memory Usage

Memory was measured in two different situations: The total memory usage by the P data structures immediately after the search filtering phase, and memory required for the last concatenation in the concatenation phase. For the RRB-tree, the extra memory required for the concatenations of larger tries is insignificant, and is in practise the sum of the P data structures. Therefore, the memory usage for RRB-trees is only shown as the memory required for the last concatenation. Figure 9.10 presents the total used memory.

As array lists doubles their capacity every time they are full and attempts to append an element, they may use considerably more memory than required: A jump in the graph indicates a large array list doubling its capacity. The concatenation step for an array list could require twice as much memory as a data structure without any overhead, depending on the capacity of the left list. If the left list has capacity enough to copy the right list into itself, the total memory required would be the size of the left list plus the size of the right. However, if there is not enough space in the left list, the contents of the left has to be reallocated into a contiguous block where there is enough space to store both. This may require copying the left list into another block, which requires the old lists and the new one to reside in memory at once.

For 4 threads and sufficiently large lists, the most expensive concatenation would be the last one. The capacity of the lists to be concatenated is likely the length of them, as they are results of previous concatenations. Consequently, the most expensive concatenation would require the sum of the lists sizes multiplied by two.

For an RRB-Tree concatenation, all three trees has to reside within memory at some point. Therefore, total memory needed for an RRB-Tree concatenation is the memory needed for both trees to be concatenated plus the result. However, as RRB-Trees utilise structural sharing, the overhead is small compared to the array list.

Although hard to see, the transient versions use more memory than the non-transient versions. This happens as the transients require the additional ID field in all nodes. Direct append seems to reduce total memory usage, most likely as it guarantees that the non-concatenated trees are leftwise dense. This results in no use of size tables and ensures minimal height.

Figure 9.11 presents the overhead ratio. As expected, the array implementation uses half of the total memory used on overhead during concatenation. On average, one would expect a single array list to have an overhead ratio of 0.25: At worst, the

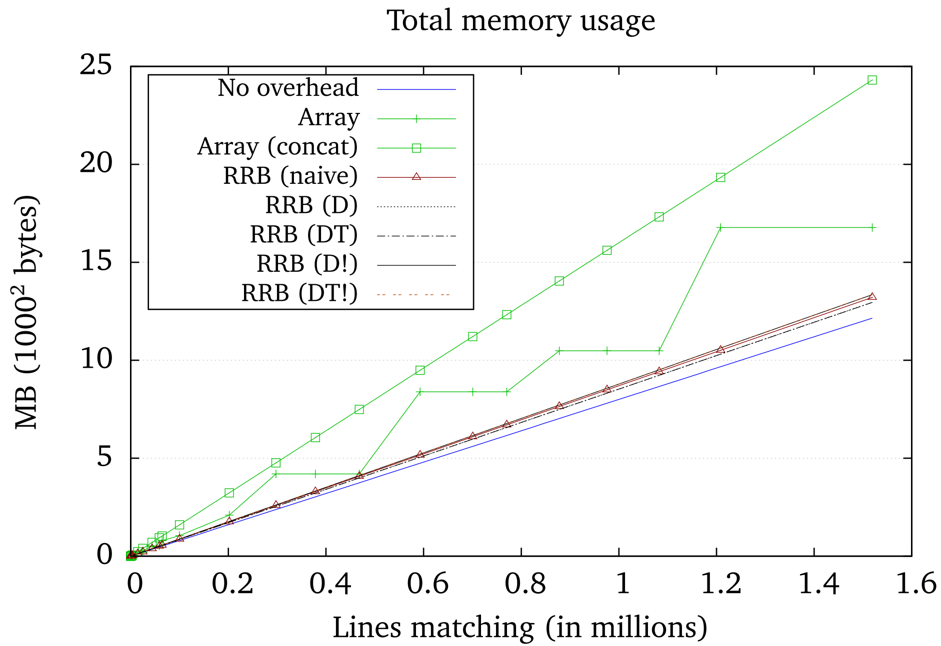


Figure 9.10: Total memory usage for different optimisation permutations.

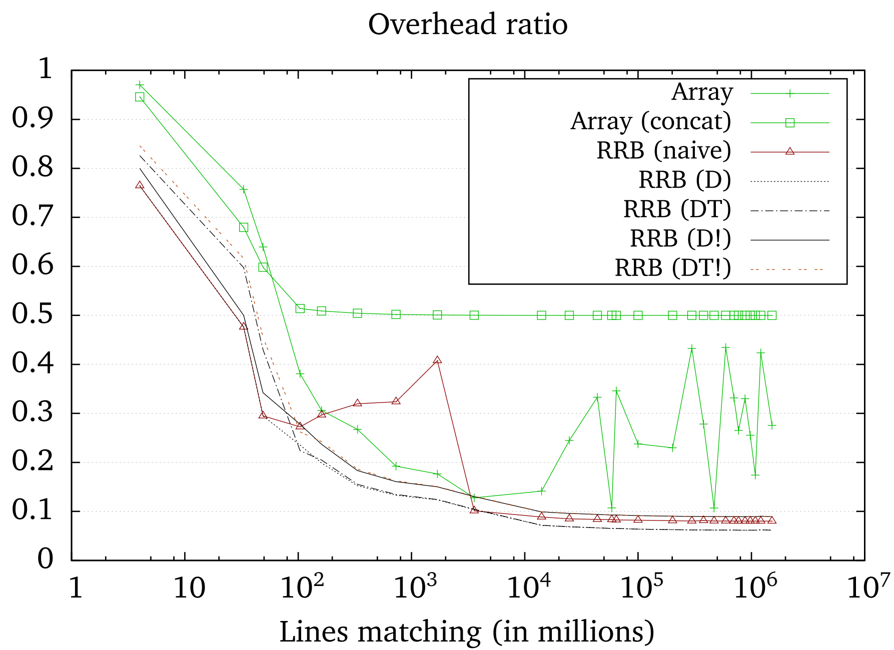


Figure 9.11: Memory overhead ratio for different optimisation permutations.

overhead would take 0.5 of all memory, and at best, it would require near 0.0. In these runs, if we exclude searches with less than 10 000 matches, the mean is near 0.3. Although this is higher than predicted, it is likely to be coincidental.

For the RRB-tree, we see that implementations using a tail and transient has notably higher overhead for smaller trees. This is to be expected, as both tail and transience adds additional fields to the RRB-tree head. The tail field should be considered negligible for larger trees, which can be seen in Figure 9.11.

It is not unreasonable to assume that Corollary 2.8 holds for RRB-trees generated by direct appending. Additionally, if few concatenations have been performed, the potential size tables added in should be considered negligible compared to the total memory used.

For the implementations not using transience, the overhead for each node is the node type (4 bytes[1]), the length field (4 bytes) and the size table pointer field (8 bytes):

$$o = 4 + 4 + 8 = 16$$

$\|\tau\| = 8$, $p = 8$ and $M = 32$. From Corollary 2.8, we then get that the the overhead ratio should be approximately $3/34 = 0.088$. However, for all results over 10 000, the overhead ratio for non-transient direct append RRB-trees is on average 0.062. This is likely because the leaf nodes do not contain a size table pointer. Since we know that there are at least $M^{h(R)}$ leaf nodes, and at most $(M^{h(R)} - 1)/(M - 1)$ internal nodes (by Theorem 2.6), we know that there will be approximately $M - 1$ leaf nodes per internal node. As such, for a better approximation, the size table pointer should be divided by M. The approximated overhead ratio when doing so is 0.0615, much closer to the actual overhead.

For transients, we have an additional 8 bytes for the ID field for each node, including leaf nodes. Corollary 2.8 gives us an approximation of about 0.089, very close to the actual mean overhead of 0.08964.

The naïve RRB-tree variant has a mean overhead ratio of 0.081, roughly 0.02 more than what Corollary 2.8 approximates. Assuming all internal nodes has a size table, we must use 4 bytes to store each pointer's size slot. Incrementing p by 4 gives us 0.0755, considerably less than what we actually have. It is of course not surprising that Corollary 2.8 does not give a good approximation on heavily concatenated RRB-trees, but it is good to verify nonetheless. Corollary 2.8 should therefore only be used to give a lower bound on the approximate overhead ratio of an RRB-tree.

If memory is a concern, it is possible to reduce the memory footprint for array lists in the search filtering phase by reducing the growth rate. In fact, the `ArrayList` implementation in OpenJDK uses a growth factor of 1.5[29], instead of the doubling used in this array list implementation. This should decrease the overhead ratio from

[1] Strictly speaking, we could require only 1 byte of memory for the node type. However, fields should be aligned to increase memory performance, and this adds in 3 bytes.

b	TOTAL MEMORY	OH	OH RATIO	APP. OH RATIO
$b = 2$	25321048	13167992	0.5200	0.5200
$b = 3$	17580624	5427568	0.3087	0.3086
$b = 4$	14636872	2483816	0.1697	0.1696
$b = 5$	13344856	1191800	0.0893	0.0891
$b = 6$	12739416	586360	0.0460	0.0457

Table 9.4: Branching factors and memory usage, for $n = 1519132$.

the search step phase by, on average, about half. Furthermore, it is usually possible to calculate the maximal possible size an array list will be. For instance, performing a filtering operation on a list of length N will at most result in a list of length N. However, reducing the growth will not help with the memory usage in the concatenation phase, which is considerably higher.

9.5.1 Branching Factor Effect

One would expect that the approximation previously derived should work regardless of branching factor. Additionally, we would expect to see a curve not too different from Figure 2.6, but with different ratios: The difference between $b = k - 1$ and $b = k$ should be roughly the double the difference between $b = k$ and $b = k + 1$. As seen in Figure 9.12, this seems to match well.

Table 9.4 shows the memory used along with approximated and actual overhead ratios, where Corollary 2.8 is used as approximation algorithm, with the following values:

$$\begin{aligned} o &= 4 + 4 + 8 + \frac{8}{2^b} \\ p &= 8 \\ M &= 2^b \\ \|\tau\| &= 8 \end{aligned}$$

This seems to fit very well with the actual overhead ratios.

The results in Figure 9.13 shows us that for $b = 2$, the overhead is actually more than half of the total memory used. Additionally, for $b = 3$, the memory used seems to fit well with the average memory usage for an array list. In general, then, it seems safe to say that the persistent vector and the RRB-tree will use considerably less memory overhead compared to an array list, specifically during concatenations.

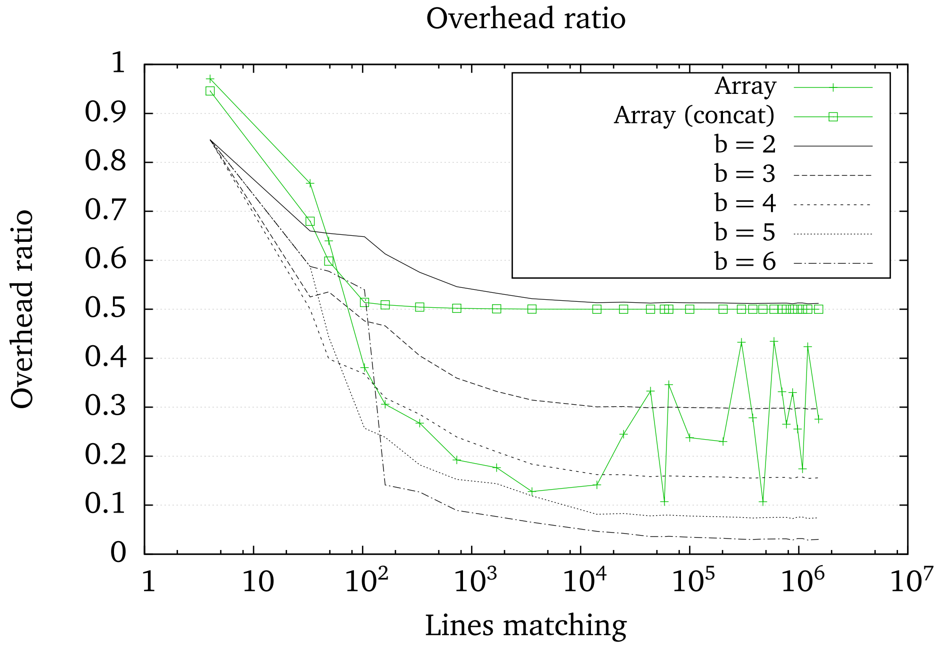


Figure 9.12: Memory overhead ratio for different branching factors.

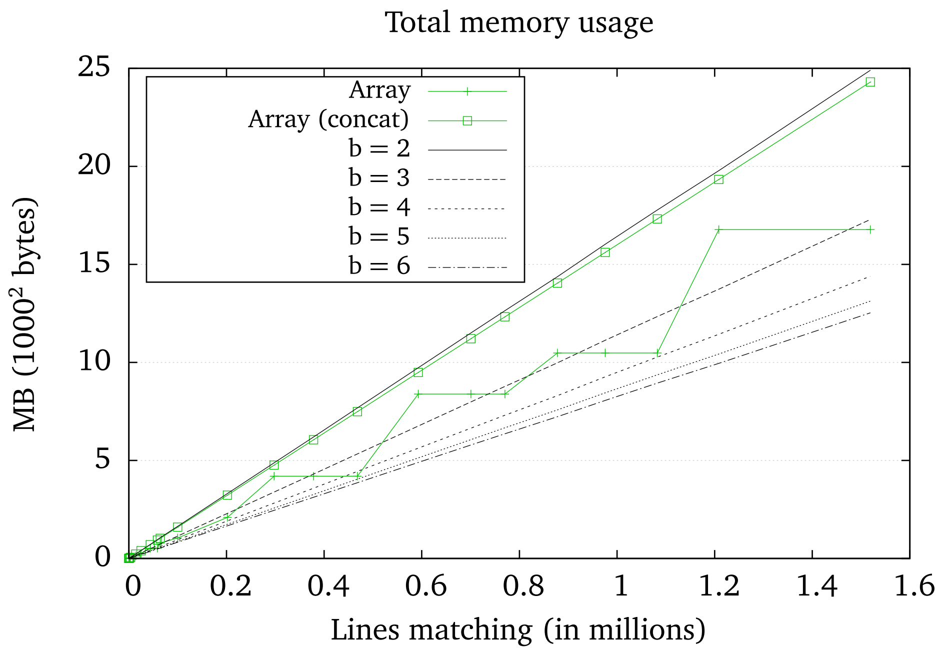


Figure 9.13: Total memory usage for different branching factors.

CHAPTER 10

Conclusion

This thesis has formally analysed the persistent vector and an extension of it, the RRB-tree. Optimisations related to RRB-tree appending has been proposed and formally explained. Measurements reveal that the optimised RRB-tree yields considerably better performance compared to an RRB-tree without these optimisations. The results also suggest that the performance may be comparative to mutable array lists in certain situations.

While the persistent vector claims to have effectively constant time appends, this can not be verified for the persistent RRB-tree in these benchmarks: Although very close to constant time, its append performance decreases somewhat with increasing size. However, this may be due to slow memory allocations by the garbage collector chosen, and the performance may very well be effectively constant time in other environments. Additionally, the benchmarks indicate that *transient* RRB-trees performs appends in effectively constant time.

The RRB-tree library `librrb` has been implemented in C. The library provides an RRB-tree implementation which can be used in programming languages able to use C libraries, or as a reference implementation for implementations in other languages. It is possible to modify both the branching factor and which optimisations to include during compile time.

10.1 Future Work

The RRB-tree is a rather young data structure, and much work has yet to be done on it. As mentioned in Section 6.3, a display and focus for an RRB-tree has not been implemented, nor has efficient iteration. The concatenation algorithm could be further analysed and potentially improved. Efficient and small rebalancing algorithms for the slice operation could be implemented to reduce the problem with potentially nondecreasing tree heights.

Memory allocation is most likely to be one of the bigger bottlenecks in the RRB-tree algorithms. Efficient memory allocation by garbage collectors is therefore essential to get an efficient RRB-tree. As we have only used the Boehm garbage collector, the

performance of other garbage collectors may be measured. Finding good options for garbage collectors tied to a programming language would also be valuable.

In this thesis, we have used static chunking as work distribution schedule. This scheduling algorithm requires considerably fewer concatenations compared to adaptive work stealing, exponential splitting and recursive splitting. For lists where the workload will be nonuniform with static chunking, and the size of the output list is not known in advance, the RRB-tree may perform even better than in the benchmarks performed in this thesis. Future work could look into the performance benefits of using RRB-trees with these scheduling algorithms.

The C library implemented, *librrb*, loop unrolls the update and index algorithms. However, the impact of loop unrolling has not been properly measured. In addition, all other algorithms walking the trie may in theory be loop unrolled. Measuring the performance benefits by loop unrolling different functions could help to further increase the RRB-tree performance.

Finally, a more formal definition of transience would help the development of a statically typed language where transient usage may be analysed and ensure correct usage at compile time. Furthermore, the definition could possibly be used in optimisation passes to increase performance in already existing functional languages, such as Haskell and OCaml.

Part IV

Appendices

Appendix A

Additional Functions

This appendix contains additional functions that are not essential to understand the algorithms themselves, but may ease development of an actual implementation.

A.1 Concatenation Functions

```
1  function CONCAT-SUB-TRIE(L, R, top)
2      if h(L) > h(R) then
3          C ← CONCAT-SUB-TRIE(L_{||L||−1}, R, false)
4          return REBALANCE(L, C, NIL, false)
5      else if h(L) < h(R) then
6          C ← CONCAT-SUB-TRIE(L, R_0, false)
7          return REBALANCE(NIL, C, R, false)
8      else if h(L) = h(R) then
9          if h(L) = 0 then
10             T ← CREATE-INTERNAL-NODE()
11             if top and |L| + |R| ⩽ M then
12                 T_0 ← EXPAND(⟨L, R⟩)          ▷ Merges L and R to a single leaf node
13             else
14                 T_0 ← L
15                 T_1 ← R
16             end if
17             return T
18         else                                  ▷ Two internal nodes with same height
19             C ← CONCAT-SUB-TRIE(L_{||L||−1}, R_0, false)
20             return REBALANCE(L, C, R, false)
21         end if
22     end if
23 end function
```

Listing A.1: CONCAT-SUB-TRIE.

The concatenation algorithm consists of an three additional helper functions: CONCAT-SUB-TRIE, which is the internal main function, REBALANCE, which performs a rebal-

ancing if needed, and EXECUTE-CONCAT-PLAN, which takes in a node and returns a new, rebalanced node based upon a concatenation plan.

We will start by describing CONCAT-SUB-TRIE: It takes in three values, L, R and top. If top is set to true, we concatenate the top nodes of two tries. Otherwise, at least one of the nodes has a parent.

Now, if the nodes have different height, we must even the difference: Only nodes of same height can be concatenated. This happens in line 2 to 7. If L, the left node is the highest, recursively call CONCAT-SUB-TRIE with its rightmost child and R. If R, the right node is the highest, we do the same, but with its leftmost child.

```
1  function REBALANCE(L, C, R, top)
2      T ← CONCAT-NODE-MERGE(L, C, R)
3      c, n ← CREATE-CONCAT-PLAN(T)
4      T ← EXECUTE-CONCAT-PLAN(T, c, n)
5      if n ⩽ M then
6          if top = false then
7              return ⟨T⟩                          ▷ Internal node with T as child
8          else
9              return T
10         end if
11     else
12         L′ ←INTERNAL-NODE-COPY(T, 0, M)
13         R′ ←INTERNAL-NODE-COPY(T, M, n − M)
14         return ⟨L′, R′⟩
15     end if
16 end function
```

Listing A.2: Rebalancing algorithm for RRB-CONCAT.

As CONCAT-SUB-TRIE returns a node C, the “centre” node. We must merge, and potentially rebalance C betweeen L and R: This is done by calling REBALANCE. The rightmost child in L and the leftmost child in R are not included in the merge, and is handled by the function CONCAT-NODE-MERGE. CONCAT-NODE-MERGE may return a node T with more than M slots.

CREATE-CONCAT-PLAN, presented in Section 3.4.2, takes the large node T and checks whether it satisfies the search step relaxed invariant: If

$$\|T\| \leqslant \|T_{opt}\| + e_{max}$$

then nothing is done, otherwise a new node distribution plan is created. T is then passed to the EXECUTE-CONCAT-PLAN, and if the nodes has to be rebalanced, it is done there.

If top is false, it means this result is passed back up to a recursive CONCAT-SUB-TRIE call, and will be rebalanced again with two nodes with higher height. If the total

amount of children are more than M, we have to split the node into two nodes regardless, so we cut them up to two nodes and put them in a new internal node with two children. If the total amount of children is less than or equal to M, then this node is a legal node. Recall that the top node can not contain a single child unless it is a leaf node, so if we are at the top, we do not put the node into an internal node unless it is going to be rebalanced at the next level.

A.2 Direct Append Helper Functions

The direct append algorithm uses three additional functions: COPYABLE-COUNT, COPY-FIRST-K and APPEND-EMPTY. We begin with the most complex one, COPYABLE-COUNT.

```
1  function COPYABLE-COUNT(R)
2      n_c ← 0                                   ▷ Nodes to copy
3      n_v ← 0                                   ▷ Nodes visited so far
4      pos ← 0                ▷ Position to place in node after copy
5      T ← R_root
6      for h ← h(R) downto 0 do
7          i′ ← ‖T‖ − 1
8          T ← T_i′
9          n_v ← n_v + 1
10         if i′ < M − 1 then
11             n_c ← n_v
12             pos ← i′
13         end if
14     end for
15     return n_c, pos
16 end function
```

Listing A.3: Implementation of COPYABLE-COUNT.

COPYABLE-COUNT returns the amount of nodes we can copy until we must create new nodes, along with the position pos where we insert new nodes in the last copied node.

How do we know that we can copy this node safely? There are two criterias we can satisfy: Either there is enough space for more children in this node, or there is space to insert the value in this node's rightmost child. Since we will walk the whole trie, we only have to know that there is at least space in this node to know that its parents are copyable. Consequently, we only check that the node we are at is copyable, and if it is, we know that its parents are.

As we always know we will walk the rightmost node, there is no reason to perform a relaxed radix search, neither in COPYABLE-COUNT nor in COPY-FIRST-K.

COPY-FIRST-K walks the rightmost nodes in the trie and performs a standard path copy. In addition, it copies and increments the size table of all nodes if they have a size table. Finally, it returns the last copied node.

```
1  function COPY-FIRST-K(R, R′, k)
2      T_orig ← R_root
3      R′_root ← CLONE(T_orig)
4      T ← R′_root
5      for h ← h(R) downto 1 do
6          i′ ← ‖T‖ − 1
7          T_i′ ← CLONE(T_orig[i′])
8          INCREMENT-SIZE-TABLE*(T_i′)      ▷ Increment last size table slot if it exists
9          T ← T_i′
10         T_orig ← T_orig[i′]
11     end for
12     return T                                          ▷ Return last copied node
13 end function
```

Listing A.4: Implementation of COPY-FIRST-K.

As COPY-FIRST-K will always copy at least one node (the main direct append function will increase the height if not), the last copied node by COPY-FIRST-K is passed to APPEND-EMPTY. If there are no nodes to copy, APPEND-EMPTY simply short circuits and returns the value passed in. Otherwise, it creates a leaf node and $h_{empty} - 1$ internal nodes bottom up. Finally, the last internal node created is inserted at slot pos in T the node passed in, and the leaf node is returned.

```
1  function APPEND-EMPTY(T, pos, h_empty)
2      if 0 < h_empty then
3          N_leaf ← CREATE-LEAF-NODE()                    ▷ Last node must be leaf
4          N ← N′
5          for i′ ← 1 to h_empty − 1 do
6              N′ ← CREATE-INTERNAL-NODE()
7              N′_0 ← N
8              N ← N′
9          end for
10         T_pos ← N
11         return N_leaf                                  ▷ Return leaf node
12     else
13         return T                 ▷ If no nodes needs to be appended, root is leaf
14     end if
15 end function
```

Listing A.5: Implementation of APPEND-EMPTY.

A.3 Direct Pop

Direct pop is described in detail in Section 5.3: As described, the idea is to use an internal stack to avoid recursive functions to speed things up. The stack is heavily used, and as seen, there are no recursive function calls done.

```
1  function RRB-POP(R)
2      R′ ← CLONE(R)
3      |R′| ← |R| − 1
4      W ←ARRAY-ON-STACK(h_max)                        ▷ All slots initially NIL
5      W_0 ← R_root
6      for i ← 1 to h(R) do
7          W_i ← W_{i−1}[||W_{i−1}|| − 1]
8      end for
9      if ||W_{h(R)}|| = 1 then             ▷ Leaf node contains only single element
10         W_{h(R)} ← NIL
11     else
12         W_{h(R)} ← CLONE(W_{h(R)})
13         W_{h(R)}[||W_{h(R)}|| − 1] ← NIL                  ▷ Remove last element
14     end if
15     for i ← h(R) − 1 downto 0 do
16         if ||W_i|| = 1 and W_{i+1} = NIL then
17             W_i ← NIL
18         else
19             W_i ←CLONE(W_i)
20             W_i[||W_i|| − 1] ← W_{i+1}
21             DECREMENT-SIZE-TABLE*(W_i)
22                           ▷ Copy and decrement last slot in size table if it exists
23         end if
24     end for
25     if h(R) > 0 and ||W_0|| = 1 then
26         R′_root ← W_1                                          ▷ Height decrease
27     else
28         R′_root ← W_0
29     end if
30     return R′
31 end function
```

Listing A.6: RRB-POP using direct popping

Appendix B

Search Terms

Search term	Line count	Percentage
whoopsie	4	0.000 %
250000	33	0.002 %
200000	49	0.003 %
ninety	104	0.007 %
seriously	161	0.010 %
branching	334	0.021 %
rrb	730	0.047 %
wow	1693	0.108 %
coffee	3579	0.229 %
Delete	14071	0.899 %
error	24746	1.581 %
edit	43712	2.792 %
branches	58513	3.738 %
merge	64292	4.107 %
fix	99908	6.383 %
Create	201848	12.895 %
JavaScript	297499	19.005 %
6a	378365	24.171 %
Java	468280	29.916 %
51	592902	37.877 %
21	700723	44.765 %
14	770307	49.210 %
09	878046	56.093 %
10	975886	62.343 %
07	1082471	69.153 %
12	1208500	77.204 %
issues	1519132	97.048 %
(empty string)	1565338	100.000 %

Table B.1: List of search terms

Table B.1 contains the list of search terms used for the benchmarks described in Chapter 9.

Appendix C

Additional Plots

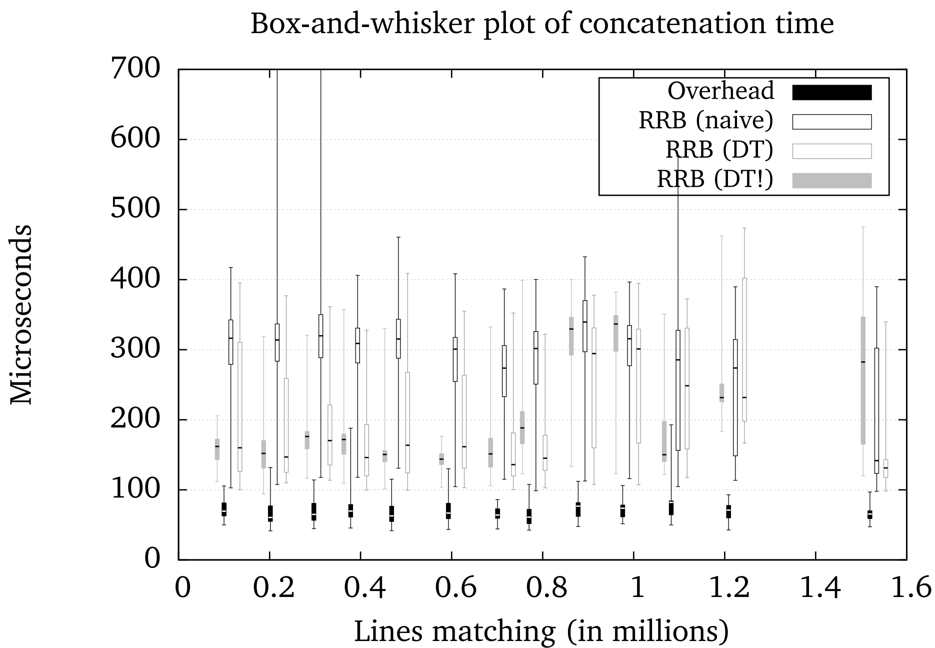


Figure C.1: Box-and-whisker plot for concatenation phase.

The box-and-whisker plot of the concatenation phase is presented in Figure C.1. To make the graph readable, the position of the box-and-whisker plots are offset so that they do not overlap. The direct append and transient direct append were not included, as they had the same pattern and would take up too much space. The y-axis was capped at 700 microseconds; the top whiskerbars from the naïve RRB-tree measurements peaked at 2500 microseconds and would render the figure unreadable if included.

As clearly seen, the variation, even for the overhead plot, is notable. As claimed in Section 9.3, this means any trends based upon the median, shown in that section, can not be considered statistically significant. Note that almost all runtimes were between 100 and 400 microseconds.

APPENDIX D

librrb Installation and Use

D.1 Setup

Listing D.1 includes all necessary actions to configure a Debian or Ubuntu-based system. It has been proven to work on Debian Jessie in May 2014. The setup consist of 3 steps: Installation of necessary programs and libraries, building the library and the benchmark programs, and running the benchmarks and visualise the result.

Line 1-8 installs the bare minimum of dependencies and libraries the RRB-Tree project depends on. `libgc1c2` is the specific Debian version of Boehm GC used for benchmarking, although other versions are likely compatible.

Line 10-18 describes how to install the nightly build of the C compiler Clang. This step is optional to get the library running, but omitting this step would affect benchmark performance.

Line 20-34 builds and optionally installs the program. The `./run-tests.sh` command checks whether the test suite runs fine, with a set of different optimisation permutations. For information on configuration options, see D.2.

Finally, line 36-41 downloads the used input data and runs the benchmarks. Note that the input data is in total 440 MB of zipped download data, and may take some time to download. Additionally, the input data generation will require up to 5 GB space during the build phase. When the file has been created, it will require 2.5 GB of space.

D.2 Options

When running the command `./configure`, different options are available to change which optimisations are set, along with other usable values. When the default value of a `--disable-*` is set to enabled, it means that the the functionality is enabled, not that the command itself is. All options are shown in Table D.1.

```
# To get build autotools up and running
sudo apt-get install build-essential automake autoconf gnu-standards\
     autoconf-doc libtool gettext autoconf-archive
# To install Boehm GC and development dependencies
sudo apt-get install libgc-dev libgc1c2
# Ensure that Boehm GC (libgc1c2) is 7.2 or higher by checking
# its version number
apt-cache show libgc1c2

## Optional steps: For installing clang
sudo echo "# LLVM
deb http://llvm.org/apt/wheezy/ llvm-toolchain-wheezy main
deb-src http://llvm.org/apt/wheezy/ llvm-toolchain-wheezy main" \
    >> /etc/apt/sources.list

wget -O - http://llvm.org/apt/llvm-snapshot.gpg.key | sudo apt-key add -
sudo apt-get install clang-3.4
## Optional steps ending

cd path/to/rrb
autoreconf --install

## If running Clang
CFLAGS="-Ofast -g" CC="clang" ./configure
## If running gcc (NB: CC must be set if clang is available)
CFLAGS="-Ofast -g" CC="gcc" ./configure
## See 'Options' section for configure flags

## For building and checking the consistency against the test suite
./run-tests.sh

## Optional: Install on the system
sudo make install
## Uninstall by doing `sudo make uninstall` in this directory

# Running and creating benchmarks
cd benchmark-suite
./make-data.sh
cd ..

./run-bench.sh
```

Listing D.1: Shell commands for system setup.

D.3 Interface

This section serves as an overview for all library functions provided by the `librrb` library. The functions are prototyped inside `rrb.h` and defined within `rrb.c`.

If a function is said to run in *effectively constant* time, one can assume the function runs in $\mathcal{O}(\log_{32} n)$ time, or the branching factor as specified when the library was

FLAG	EXPLANATION	DEFAULT VALUE
`--disable-tail`	Disables tail	Enabled
`--with-branching=b`	Set b, such that $M = 2^b$	$b = 5$, $M = 2^5$
`--disable-transients`	Disables transients	Enabled
`--disable-transients-check-thread`	Disables transient thread checking	Enabled
`--disable-direct-append`	Disables direct append	Enabled
`--disable-debug`	Disables debugging functions	Enabled

Table D.1: List of configurable options

configured.

D.3.1 RRB-tree Functions

All RRB-tree functions described here do not modify the RRB-tree in any way, shape or form. Passing in any value will never modify the tree.

```
const RRB* rrb_create(void)
```
Returns, in constant time, an immutable, empty RRB-Tree.

```
uint32_t rrb_count(const RRB *rrb)
```
Returns, in constant time, the number of items in this RRB-Tree.

```
void* rrb_nth(const RRB *rrb, uint32_t index)
```
Returns, in effectively constant time, the item at index *index*.

```
const RRB* rrb_pop(const RRB *rrb)
```
Returns, in effectively constant time, a new RRB-Tree without the last item.

```
void* rrb_peek(const RRB *rrb)
```
Returns, in constant time[1], the last item in this RRB-Tree.

```
const RRB* rrb_push(const RRB *rrb, const void *elt)
```
Returns, in effectively constant time[2], a new RRB-Tree with *elt* appended to the end of the original RRB-Tree.

```
const RRB* rrb_update(const RRB *rrb, uint32_t index,
                                       const void *elt)
```
Returns, in effectively constant time, a new RRB-Tree where the item at index *index* is replaced by *elt*.

[1]Or effectively constant time, if the tail optimisation is not applied.
[2]When all optimisation are applied.

```
const RRB* rrb_concat(const RRB *left, const RRB *right)
```
Returns, in $\mathcal{O}(\log n)$ time, the concatenation of *left* and *right* as a new RRB-Tree.

```
const RRB* rrb_slice(const RRB *rrb, uint32_t from, uint32_t to)
```
Returns, in effectively constant time, a new RRB-Tree which only contain the items from index *from* to index *to* in the original RRB-Tree.

D.3.2 Transient Functions

Transient functions are only available when the `--disable-transients` is **not** passed to `./configure`. When enabled, the type `TransientRRB` is available, and acts as defined in Chapter 4. In the worst case, transient functions provide same performance as their counterpart. Repeated use will in most cases increase performance.

The transients will by default check that the thread they are created in is where the functions are called. This feature can be turned off by passing the flag `--disable-transients-check-thread` to `./configure`.

```
TransientRRB* rrb_to_transient(const RRB *rrb)
```
Converts, in constant time, a persistent RRB-tree to its transient counterpart. The persistent RRB-tree can still be used.

```
const RRB* transient_to_rrb(TransientRRB *trrb);
```
Converts, in constant time, a transient RRB-tree to a persistent RRB-tree. The transient RRB-tree is *invalidated*.

```
uint32_t transient_rrb_count(const TransientRRB *trrb)
```
Returns, in constant time, the number of elements in this transient RRB-tree.

```
void* transient_rrb_nth(const TransientRRB *trrb, uint32_t index)
```
Returns, in effectively constant time, the item at index *index*.

```
TransientRRB* transient_rrb_pop(TransientRRB *trrb)
```
Returns, in effectively constant time, a new transient RRB-tree without the last item. The original transient RRB-tree is *invalidated*.

```
void* transient_rrb_peek(const TransientRRB *trrb);
```
Returns, in constant time[3], the last element in this RRB-tree.

```
TransientRRB* transient_rrb_push(TransientRRB *restrict trrb,
                                 const void *restrict elt)
```
Returns, in effectively constant time[4], a new transient RRB-Tree with *elt* appended to the end of the original transient RRB-Tree. The original transient RRB-tree is *invalidated*.

[3] Or effectively constant time, if the tail optimisation is not applied.
[4] If all optimisations are applied.

```
TransientRRB* transient_rrb_update(TransientRRB *restrict trrb,
                      uint32_t index, const void *restrict elt)
```
Returns, in effectively constant time, a new transient RRB-Tree where the item at index *index* is replaced by *elt*. The original transient RRB-tree is *invalidated*.

```
TransientRRB* transient_rrb_slice(TransientRRB *trrb,
                          uint32_t from, uint32_t to)
```
Returns, in effectively constant time, a new transient RRB-tree which only contain the items from index *from* to index *to* in the original RRB-Tree. The original transient RRB-tree is *invalidated*.

D.3.3 Debugging Functions

Debugging functions have no performance guarantees, and may be slow. A dot file may be converted to a graph through the Graphviz tool `dot`. For instance would "`dot -Tpng -o output.png input.dot`" convert the dot file `input.dot` to the png file `output.png`.

Although the debugging functions are intented to be used for persistent RRB-trees, they work completely fine on transient RRB-trees as well. Casting the transient to a persistent RRB-tree by doing `(const RRB*)` should be sufficient to avoid warnings.

```
void rrb_to_dot_file(const RRB *rrb, char *fname)
```
Writes a new file with the filename *fname* with a dot representation of the RRB-Tree provided. If the file already exists, the original content is overridden.

```
DotFile dot_file_create(char *fname)
```
Creates and opens a DotFile. Note that a DotFile is passed by value, not by pointer.

```
DotFile dot_file_close(DotFile dot)
```
Writes and closes a DotFile to disk. Any call performed after the DotFile is closed is considered an error.

```
void rrb_to_dot(DotFile dot, const RRB *rrb)
```
Writes the RRB-Tree to the dot file, if not already written. Shared structure between RRB-Trees is visualised, with the exception of *NULL* pointers.

```
void label_pointer(DotFile dot, const void *node, const char *name)
```
Labels a pointer (Usually an RRB-Tree pointer) in the dot file. It is legal with multiple labels on a single pointer. If the pointer is *NULL*, the label will be attached to the latest *NULL* pointer added to the dot file. Labelling pointers not contained in the dot file is not an error, but will generate strange visualisations.

```
uint32_t rrb_memory_usage(const RRB *const *rrbs,
                          uint32_t rrb_count)
```
Calculates the expected memory used by *rrb_count* RRB-Trees, and takes into account structural sharing between them.

Bibliography

[1] N. Shavit and D. Touitou. "Software Transactional Memory". In: *Distributed Computing* 10.2 (1997), pp. 99–116.

[2] D. Lea. "A Java Fork/Join Framework". In: *Proceedings of the ACM 2000 Conference on Java Grande*. JAVA '00. San Francisco, California, USA: ACM, 2000, pp. 36–43.

[3] P. Bagwell and T. Rompf. *RRB-Trees: Efficient Immutable Vectors*. Tech. rep. EPFL, 2011.

[4] R. Hickey. *The Clojure programming language*. 2006. URL: http://clojure.org (visited on Dec. 16, 2013).

[5] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. "Making Data Structures Persistent". In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. STOC '86. Berkeley, California, USA: ACM, 1986, pp. 109–121. ISBN: 0-89791-193-8. DOI: 10.1145/12130.12142. URL: http://doi.acm.org/10.1145/12130.12142.

[6] C. Okasaki. "Purely Functional Random-access Lists". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: ACM, 1995, pp. 86–95. ISBN: 0-89791-719-7. DOI: 10.1145/224164.224187. URL: http://doi.acm.org/10.1145/224164.224187.

[7] P. Bagwell. *Fast Functional Lists, Hash-Lists, Deques and Variable Length Arrays*. Tech. rep. 2002.

[8] P. Bagwell. *Ideal Hash Trees*. Tech. rep. EPFL, Oct. 2001.

[9] R. Hickey. *Re: 20 Days of Clojure - Day 7: PersistentVector*. [online]. In: *Clojure Mailing List*. Mar. 9, 2008. URL: https://groups.google.com/d/msg/clojure/UtEclYla9N8/8-VzDZUWlPsJ (visited on May 27, 2014).

[10] R. Hickey. *Transient Data Structures*. [online]. In: *Clojure Mailing List*. Aug. 3, 2009. URL: https://groups.google.com/d/msg/clojure/zdYsxH1K0B0/0Gm5YSGKNZcJ (visited on May 27, 2014).

[11] D. Knuth. *The Art of Computer Programming: Sorting and Searching v. 3*. Reading, Mass: Addison-Wesley Pub. Co, 1998. ISBN: 0-201-89685-0.

[12] C. Okasaki and A. Gill. "Fast Mergeable Integer Maps". In: *In Workshop on ML*. 1998, pp. 77–86.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-018. Mar. 2009.

[14] N. Sarnak and R. E. Tarjan. "Planar Point Location Using Persistent Search Trees". In: *Commun. ACM* 29.7 (July 1986), pp. 669–679. ISSN: 0001-0782. DOI: 10.1145/6138.6151. URL: http://doi.acm.org/10.1145/6138.6151.

[15] A. Agarwal, J. Hennessy, and M. Horowitz. "An Analytical Cache Model". In: *ACM Trans. Comput. Syst.* 7.2 (May 1989), pp. 184–215. ISSN: 0734-2071. DOI: 10.1145/63404.63407. URL: http://doi.acm.org/10.1145/63404.63407.

[16] R. Fitzgerald and D. Tarditi. "The Case for Profile-directed Selection of Garbage Collectors". In: *Proceedings of the 2nd International Symposium on Memory Management*. ISMM '00. Minneapolis, Minnesota, USA: ACM, 2000, pp. 111–120. ISBN: 1-58113-263-8. DOI: 10.1145/362422.362472. URL: http://doi.acm.org/10.1145/362422.362472.

[17] H.-J. Boehm, R. Atkinson, and M. Plass. "Ropes: an alternative to strings". In: *Software: Practice and Experience* 25.12 (1995), pp. 1315–1330.

[18] R. Hinze and R. Paterson. "Finger Trees: A Simple General-purpose Data Structure". In: *J. Funct. Program.* 16.2 (Mar. 2006), pp. 197–217. ISSN: 0956-7968. DOI: 10.1017/S0956796805005769. URL: http://dx.doi.org/10.1017/S0956796805005769.

[19] J. N. L'orange. *RRB-Tree Performance in Real Applications*. Tech. rep. NTNU, Nov. 2013.

[20] P. Wadler. "Linear Types Can Change the World!" In: *Programming Concepts And Methods*. North, 1990.

[21] S. Peyton Jones and S. Marlow. "Secrets of the Glasgow Haskell Compiler inliner". In: *Journal of Functional Programming* 12 (4-5 July 2002), pp. 393–434. ISSN: 1469-7653. DOI: 10.1017/S0956796802004331. URL: http://journals.cambridge.org/article_S0956796802004331.

[22] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis transformation". In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[23] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st ed. Chapman and Hall/CRC, Aug. 2011. ISBN: 9781420082791.

[24] D. R. Edelson. "Smart Pointers: They're Smart, but They're Not Pointers". In: *USENIX C++ Conference*. Portland, OR: USENIX, Aug. 1992.

[25] J. A. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-m. W. Hwu, and N. Obeid. “Algorithm and data optimization techniques for scaling to massively threaded systems”. In: *Computer* 45.8 (2012), pp. 26–32.

[26] S. F. Hummel, E. Schonberg, and L. E. Flynn. “Factoring: A Method for Scheduling Parallel Loops”. In: *Commun. ACM* 35.8 (Aug. 1992), pp. 90–101.

[27] A. Prokopec, T. Rompf, P. Bagwell, and M. Odersky. *On A Generic Parallel Collection Framework*. Tech. rep. EPFL, 2011.

[28] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. “Cilk: An Efficient Multithreaded Runtime System”. In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’95. Santa Barbara, California, USA: ACM, 1995, pp. 207–216.

[29] Oracle Corporation. *Source code for OpenJDK 7u40 Build b43*. Mercurial repository, revision 6506 at branch `default`. URL: http://hg.openjdk.java.net/jdk7u/jdk7u40/jdk (visited on Dec. 16, 2013).