BULK UPDATES AND CACHE SENSITIVITY IN SEARCH TREES

TKK Research Reports in Computer Science and Engineering A
Espoo 2009 TKK-CSE-A2/09

BULK UPDATES AND CACHE SENSITIVITY IN SEARCH TREES

Doctoral Dissertation

Riku Saikkonen

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Faculty of Information and Natural Sciences, Helsinki University of Technology for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 4th of September, 2009, at 12 noon.

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering
P.O. Box 5400
FI-02015 TKK
FINLAND
URL: http://www.cse.tkk.fi/
Tel. +358 9 451 3228
Fax. +358 9 451 3293
E-mail: rjs@cs.hut.fi

© 2009 Riku Saikkonen

Layout: Riku Saikkonen (except cover)
Cover image: Riku Saikkonen
Set in the Computer Modern typeface family designed by Donald E. Knuth.

ISBN 978-952-248-037-8
ISBN 978-952-248-038-5 (PDF)
ISSN 1797-6928
ISSN 1797-6936 (PDF)
URL: http://lib.tkk.fi/Diss/2009/isbn9789522480385/

Multiprint Oy
Espoo 2009

ABSTRACT OF DOCTORAL DISSERTATION		HELSINKI UNIVERSITY OF TECHNOLOGY P.O. Box 1000, FI-02015 TKK http://www.tkk.fi/	
Author	Riku Saikkonen		
Name of the dissertation Bulk Updates and Cache Sensitivity in Search Trees			
Manuscript submitted	09.04.2009	Manuscript revised	15.08.2009
Date of the defence	04.09.2009		
☒ Monograph		☐ Article dissertation (summary + original articles)	
Faculty	Faculty of Information and Natural Sciences		
Department	Department of Computer Science and Engineering		
Field of research	Software Systems		
Opponent(s)	Prof. Peter Widmayer		
Supervisor	Prof. Eljas Soisalon-Soininen		
Instructor	Prof. Eljas Soisalon-Soininen		

Abstract

This thesis examines two topics related to binary search trees: cache-sensitive memory layouts and AVL-tree bulk-update operations. Bulk updates are also applied to adaptive sorting.

Cache-sensitive data structures are tailored to the hardware caches in modern computers. The thesis presents a method for adding cache-sensitivity to binary search trees without changing the rebalancing strategy. Cache-sensitivity is maintained using worst-case constant-time operations executed when the tree changes. The thesis presents experiments performed on AVL trees and red-black trees, including a comparison with cache-sensitive B-trees.

Next, the thesis examines bulk insertion and bulk deletion in AVL trees. Bulk insertion inserts several keys in one operation. The number of rotations used by AVL-tree bulk insertion is shown to be worst-case logarithmic in the number of inserted keys, if they go to the same location in the tree. Bulk deletion deletes an interval of keys. When amortized over a sequence of bulk deletions, each deletion requires a number of rotations that is logarithmic in the number of deleted keys. The search cost and total rebalancing complexity of inserting or deleting keys from several locations in the tree are also analyzed. Experiments show that the algorithms work efficiently with randomly generated input data.

Adaptive sorting algorithms are efficient when the input is nearly sorted according to some measure of presortedness. The thesis presents an AVL-tree-based variation of the adaptive sorting algorithm known as local insertion sort. Bulk insertion is applied by extracting consecutive ascending or descending keys from the input to be sorted. A variant that does not require a special bulk-insertion algorithm is also given. Experiments show that applying bulk insertion considerably reduces the number of comparisons and time needed to sort nearly sorted sequences. The algorithms are also compared with various other adaptive and non-adaptive sorting algorithms.

Keywords	cache-sensitivity, cache-consciousness, bulk insertion, bulk deletion, adaptive sorting, AVL tree, binary search tree		
ISBN (printed)	978-952-248-037-8	ISSN (printed)	1797-6928
ISBN (pdf)	978-952-248-038-5	ISSN (pdf)	1797-6936
Language	English	Number of pages	144 p.
Publisher	Department of Computer Science and Engineering		
Print distribution	Department of Computer Science and Engineering		
☒ The dissertation can be read at http://lib.tkk.fi/Diss/2009/isbn9789522480385/			

<table>
<tr><td colspan="2">VÄITÖSKIRJAN TIIVISTELMÄ</td><td colspan="2">TEKNILLINEN KORKEAKOULU
PL 1000, 02015 TKK
http://www.tkk.fi/</td></tr>
<tr><td>Tekijä</td><td colspan="3">Riku Saikkonen</td></tr>
<tr><td colspan="4">Väitöskirjan nimi
Eräpäivitykset ja välimuistitietoisuus hakupuissa</td></tr>
<tr><td>Käsikirjoituksen päivämäärä</td><td>09.04.2009</td><td>Korjatun käsikirjoituksen päivämäärä</td><td>15.08.2009</td></tr>
<tr><td>Väitöstilaisuuden ajankohta</td><td colspan="3">04.09.2009</td></tr>
<tr><td colspan="2">☒ Monografia</td><td colspan="2">☐ Yhdistelmäväitöskirja (yhteenveto + erillisartikkelit)</td></tr>
<tr><td>Tiedekunta
Laitos
Tutkimusala
Vastaväittäjä(t)
Työn valvoja
Työn ohjaaja</td><td colspan="3">Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan laitos
Ohjelmistojärjestelmät
prof. Peter Widmayer
prof. Eljas Soisalon-Soininen
prof. Eljas Soisalon-Soininen</td></tr>
</table>

Tiivistelmä

Väitöskirja käsittelee kahta binäärisiin hakupuihin liittyvää aihetta: välimuistitietoista solmujen sijoittamista muistiin sekä AVL-puiden eräpäivitysoperaatioita. Eräpäivityksiä myös sovelletaan mukautuvaan järjestämiseen.

Välimuistitietoiset tietorakenteet on sovitettu nykyisten tietokoneiden laitteistotason välimuisteihin. Väitöskirja esittelee tavan tehdä binäärisistä hakupuista välimuistitietoisia muuttamatta hakupuun tasapainotusmenetelmää. Välimuistitietoisuus säilytetään tekemällä puun muuttuessa pahimmassakin tapauksessa vakioaikaisia operaatioita. Väitöskirja esittelee AVL- ja punamustilla puilla tehtyjä kokeita, sisältäen vertailun välimuistitietoisiin B-puihin.

Seuraavaksi väitöskirja käsittelee AVL-puiden erälisäys- ja eräpoisto-operaatioita. Erälisäys lisää monta avainta puuhun kerralla. Väitöskirjassa näytetään, että AVL-puun erälisäys tekee pahimmassa tapauksessa logaritmisen määrän kiertoja suhteessa lisättyjen avainten lukumäärään, jos avaimet kuuluvat puussa samaan kohtaan. Eräpoisto poistaa puusta kokonaisen avainvälin. Jonossa eräpoisto-operaatioita kukin poisto tarvitsee tasoitetusti logaritmisen määrän kiertoja suhteessa poistettavien avainten lukumäärään. Lisäksi tutkitaan hakukustannusta ja koko tasapainotuksen kustannusta tilanteessa, jossa lisäyksiä tai poistoja tehdään useaan kohtaan puussa. Koetulosten mukaan algoritmit toimivat tehokkaasti satunnaisesti tuotetuilla syötteillä.

Mukautuvat järjestämisalgoritmit ovat tehokkaita, kun syöte on ennestään melkein järjestyksessä jonkin järjestystä kuvaavan suureen mukaan. Väitöskirja esittelee AVL-puupohjaisen muunnelman paikallinen lisäysjärjestäminen -nimisestä mukautuvasta järjestämisalgoritmista. Erälisäystä sovelletaan keräämällä järjestettävästä syötteestä peräkkäisiä kasvavia tai väheneviä avaimia. Lisäksi esitellään muunnelma, joka ei tarvitse varsinaista erälisäysalgoritmia. Koetulosten mukaan erälisäyksen soveltaminen vähentää tarvittavia vertailuoperaatioita ja suoritusaikaa huomattavasti, kun järjestettävä jono on melkein järjestyksessä. Algoritmeja verrataan myös muihin mukautuviin ja ei-mukautuviin järjestämisalgoritmeihin.

<table>
<tr><td>Asiasanat</td><td colspan="3">välimuistitietoisuus, erälisäys, eräpoisto, mukautuva järjestäminen, AVL-puu, binäärinen hakupuu</td></tr>
<tr><td>ISBN (painettu)</td><td>978-952-248-037-8</td><td>ISSN (painettu)</td><td>1797-6928</td></tr>
<tr><td>ISBN (pdf)</td><td>978-952-248-038-5</td><td>ISSN (pdf)</td><td>1797-6936</td></tr>
<tr><td>Kieli</td><td>englanti</td><td>Sivumäärä</td><td>144 s.</td></tr>
<tr><td>Julkaisija</td><td colspan="3">Tietotekniikan laitos</td></tr>
<tr><td>Painetun väitöskirjan jakelu</td><td colspan="3">Tietotekniikan laitos</td></tr>
<tr><td colspan="4">☒ Luettavissa verkossa osoitteessa http://lib.tkk.fi/Diss/2009/isbn9789522480385/</td></tr>
</table>

Preface

First of all, I would like to thank the supervisor of this work, Professor Eljas Soisalon-Soininen. He has introduced me to the field of algorithm and data structure research and provided an environment where I had the opportunity to concentrate primarily on the research work. He has been very supportive, and an extremely valuable guide and source of new ideas for the research.

I am grateful to Professors Tapio Elomaa (Tampere University of Technology) and Otto Nurmi (University of Helsinki) for their detailed and valuable comments on the manuscript, and to Professor Peter Widmayer (ETH Zürich) for kindly agreeing to be the opponent.

The research presented in this thesis was carried out at the Department of Computer Science and Engineering at the Helsinki University of Technology. I thank the Department, as well as the Helsinki Graduate School in Computer Science and Engineering and the Academy of Finland for providing the financial support that enabled me to work on this thesis.

I would also like to thank the authors (far too many to list here) of the free open-source software that I used in this work. The most useful programs were GNU Emacs, Latex and its packages, Gnuplot, the GNU C compiler, and Perl. Thanks also to the people behind the Debian GNU/Linux distribution, which has provided a stable operating system that included all the above software.

Finally, I wish to thank my family and friends for their support throughout my studies.

Espoo, August 2009

Riku Saikkonen

Contents

1 INTRODUCTION 1

1.1 Bulk updates · *p. 1*
1.2 Cache sensitivity · *p. 2*
1.3 Adaptive sorting · *p. 3*
1.4 Organization · *p. 4*

2 BACKGROUND 5

2.1 Search trees · *p. 5*
2.2 AVL trees · *p. 6*
2.3 Red-black trees · *p. 7*
2.4 Internal and external trees · *p. 8*
2.5 Height-valued trees · *p. 10*
2.6 Parent links · *p. 11*
2.7 Relaxed balancing · *p. 11*
2.8 Bulk updates · *p. 12*
2.9 Cache-conscious search trees · *p. 13*
2.10 Adaptive sorting · *p. 15*
2.11 Finger trees · *p. 17*

3 CACHE-SENSITIVE BINARY SEARCH TREES 19

3.1 Cache model · *p. 19*
3.2 Multi-level cache-sensitive layout · *p. 21*
3.3 Global relocation · *p. 24*
3.4 Local relocation · *p. 29*
3.5 Experiments · *p. 35*

4 BALANCING AVL TREES 43

4.1 The node balancing algorithm · *p. 43*
4.2 Standard insertion and deletion · *p. 44*
4.3 An implementation of relaxed balancing · *p. 45*
4.4 Analysis of node balancing · *p. 46*

5 BULK UPDATE ALGORITHMS FOR AVL TREES 51

5.1 Single-bulk insertion · *p. 51*
5.2 Log-squared rebalancing algorithm · *p. 52*
5.3 Logarithmic rebalancing algorithm · *p. 56*
5.4 Inserting multiple bulks · *p. 73*
5.5 Single-bulk deletion · *p. 77*
5.6 Deleting multiple intervals · *p. 84*
5.7 Comparison to relaxed balancing · *p. 86*
5.8 Experiments · *p. 87*

6 USING BULK INSERTION IN ADAPTIVE SORTING 93

6.1 Simplified finger: Binary saved path · *p. 93*
6.2 Finding ascending or descending sequences · *p. 95*
6.3 Bulk-insertion sort in an AVL tree · *p. 97*
6.4 Lazy bulks · *p. 98*
6.5 The bulk tree · *p. 99*
6.6 Analysis · *p. 101*
6.7 Experiments · *p. 104*

7 CONCLUSION 113

7.1 Cache-sensitive binary search trees · *p. 113*
7.2 Bulk updates · *p. 115*
7.3 Application to adaptive sorting · *p. 115*

REFERENCES 119

List of Algorithms

3.1 Global relocation · *p. 26*
3.2 Address translation for aliasing correction · *p. 28*
3.3 Local relocation · *p. 33*
4.1 Node balancing · *p. 44*
4.2 AVL tree single insertion in terms of Algorithm 4.1 · *p. 45*
4.3 AVL tree single deletion in terms of Algorithm 4.1 · *p. 46*
5.1 Single-bulk insertion · *p. 52*
5.2 Log-squared rebalancing for bulk insertion · *p. 53*
5.3 Logarithmic rebalancing for bulk insertion · *p. 57*
5.4 Simple bulk insertion with multiple bulks · *p. 73*
5.5 Advanced bulk insertion with multiple bulks · *p. 76*
5.6 Actual bulk deletion in an internal tree · *p. 79*
5.7 Rebalancing for bulk deletion · *p. 80*
6.1 Bulk-insertion sort · *p. 98*

List of Figures

2.1 Actual insertion and deletion in binary search trees · *p. 9*
2.2 Binary search tree rotations · *p. 10*
3.1 Reading a 4-byte word in a hypothetical memory hierarchy · *p. 20*
3.2 An optimal single-level memory layout for a binary tree · *p. 22*
3.3 Broken nodes · *p. 32*
3.4 Effect of global and local relocation on search time · *p. 37*
3.5 Effect of aliasing correction on search time · *p. 39*
3.6 Effect of local relocation on insertion and deletion time · *p. 40*
3.7 Degradation of locality after global relocation · *p. 42*
4.1 Single rotation case in proof of Theorem 4.1 · *p. 47*
4.2 Double rotation case in proof of Theorem 4.1 · *p. 48*
4.3 Special case of double rotation in proof of Theorem 4.1 · *p. 49*
5.1 Minimum growth of siblings of a path in an AVL tree · *p. 53*
5.2 Maximum growth of siblings of a path in an AVL tree · *p. 54*
5.3 Notation used in Lemmas 5.2 and 5.3 · *p. 55*
5.4 Example of different rebalancing in [77] · *p. 58*
5.5 Definitions used in Section 5.3.2 · *p. 60*
5.6 A position tree in bulk insertion with multiple bulks · *p. 74*
5.7 Overview of bulk deleting an interval · *p. 78*
5.8 Storing detached subtrees · *p. 81*
5.9 Rotations used by bulk insertion · *p. 88*
5.10 Time spent in bulk insertion · *p. 89*
5.11 Comparison of approaches for multiple bulks · *p. 90*
5.12 Rotations used by bulk deletion · *p. 92*
5.13 Time spent in bulk deletion · *p. 92*
6.1 Binary saved path construction · *p. 94*
6.2 Example of created bulks · *p. 98*
6.3 Comparisons per element used in sorting · *p. 108*
6.4 Total time used in sorting, integer keys · *p. 109*
6.5 Total time used in sorting, string keys · *p. 110*

List of Tables

2.1 Comparison of AVL and red-black trees · *p. 8*
3.1 Search time, path lengths and space usage for various trees · *p. 38*
5.1 Rotations in the logarithmic rebalancing algorithm · *p. 57*
5.2 Cases for Lemmas 5.6 and 5.7 · *p. 61*
5.3 Read-set sizes and subtree detachments in bulk deletion · *p. 91*
6.1 A selection of experimental results on sorting · *p. 111*

CHAPTER 1

Introduction

This chapter gives a brief introduction to the topics of the dissertation and an overview of the main results. More detailed background information and references are given in the next chapter.

1.1 Bulk updates

Work for this dissertation began as a continuation of the 2002 article by Soisalon-Soininen and Widmayer [77] about bulk updates in AVL trees.

Bulk updates [2, 27, 36, 46, 49, 50, 54, 55, 66, 68, 77–79] are operations that insert or delete several keys from a search tree in one operation. Their primary motivation is that bulk updates provide efficient rebalancing when many keys are inserted or deleted from the same location in the tree. For instance, as will be seen in later chapters, bulk insertions and deletions in AVL trees use $O(\log m)$ amortized time for rebalancing after inserting or deleting m keys in the same location, compared to $O(m)$ for repeated insertions or deletions of single keys.

Potential applications of bulk updates include data warehouses [40], differential indexing [46, 67, 79] (where an index is updated when a bulk of recent updates is received from a main index) and full-text indexing of document databases using the inverted index technique [21, 49, 76]. Moreover, this dissertation considers applying bulk insertion to the field of adaptive sorting, where it serves as an algorithmic tool for optimizing a search-tree-based adaptive sorting algorithm.

This dissertation examines the bulk insertion and deletion algorithms for AVL trees in detail (the article [77] only gave an overview of the algorithms), and corrects a slight error in the algorithm of [77]. Detailed proofs of the rebalancing complexity that also examine constant factors to some degree are included. In addition, a more efficient

rebalancing algorithm is given for the case where new keys are inserted in several locations in the tree. Finally, experimental results from implementations of the bulk insertion and bulk deletion algorithms are presented, confirming that the bulk update algorithms are very efficient in practice.

1.2 Cache sensitivity

Experimentation with the bulk update algorithms eventually lead to an exploration of cache sensitivity in binary search trees, which forms the second topic of this dissertation. Cache-sensitive data structures are specially tailored to the hardware caches present in modern computers. These caches transfer data between different levels of the memory hierarchy in blocks of, for example, 64 or 4096 bytes. Cache-sensitive algorithms are assumed to know these block sizes and configure themselves accordingly. In cache-sensitive search trees, the primary goal is to enhance search performance by reducing the number of separate cache blocks visited on a search path.

The previous research on cache-sensitive search trees has concentrated on variants of the B-tree [11, 16, 17, 37, 69–71], but the present dissertation considers binary search trees. The main result is an algorithm that preserves a cache-sensitive memory layout in any dynamic binary search tree that uses rotations for balancing (e.g., AVL trees and red-black trees), without changing the complexity of insertion or deletion or the structure of the nodes. The algorithm is based on preserving a simple memory-layout invariant using a constant-time operation whenever the tree changes. This algorithm is one-level, i.e., sensitive to only one cache block size in the memory hierarchy.

The dynamic one-level algorithm is compared to a multi-level approach that makes a copy of a tree into a more optimized memory layout which is not preserved during updates; in this algorithm, the main contribution of the dissertation is the consideration of a specific problem that arises from using multi-level memory layouts with set-associative caches. The latter approach is also applied to cache-sensitive B-trees, where experiments show that it gives a slight increase in search performance.

The experiments reported in the dissertation apply the techniques to both red-black and AVL trees, and include a comparison with cache-sensitive B-trees. These indicate that the cache-sensitive memory layouts improve the search performance of binary search trees by 30–50%.

In the experiments, binary search trees did not quite reach the performance of the cache-sensitive B-trees. However, in practice there can be other reasons to use binary search trees than the average-case efficiency that the experiments study.

Though the idea of studying cache-sensitivity in binary trees, and especially of using a memory-layout invariant, was derived from the bulk-update experiments, the specialized topic of cache-sensitive bulk updates is not considered in this dissertation.

1.3 Adaptive sorting

The final part of the research conducted for this dissertation was to apply bulk insertion to the field of adaptive sorting. Adaptive sorting [26, 65] studies sorting algorithms that are adaptive to the amount of order present in the input sequence; that is, they are more efficient when the sequence is already nearly sorted according to some intuitive measure of presortedness. Several measures of presortedness exist in the literature, and the various adaptive sorting algorithms have running times that depend on the values of one or more of these measures.

This dissertation applies bulk insertion to a variation of the adaptive sorting algorithm known as local insertion sort [56]. Local insertion sort inserts the values to be sorted one by one into a finger search tree, and finally traverses the tree to read the sorted output. A finger search tree [14,33,39,44,56] is used to make tree traversal to the next insertion position efficient when the input data is nearly sorted. Instead of the finger search tree, which is a complex form of a B-tree, this dissertation uses a standard AVL tree with an auxiliary data structure that works as a comparison-efficient simplified finger.

Bulk insertion is applied to insert a bulk of consecutive ascending or descending elements in the input in one operation. The produced sorting algorithm, called bulk-insertion sort, is shown to be optimal with respect to a measure of presortedness called *Inv* (the number of inversions, or pairs of elements that are in the wrong order, present in the input), and with respect to a new measure called *Bulk*, which captures the adaptivity produced by applying bulk insertions.

In addition to using the bulk-insertion algorithm, the dissertation also presents a solution where bulks can be inserted as single nodes, without using an actual bulk-insertion algorithm. This simpler solution, called the bulk tree, is shown to be optimal with respect to the *Bulk* measure.

The experiments in the dissertation compare bulk-insertion sort and the bulk tree with an AVL-tree based local insertion sorting algorithm that does not apply bulk insertion, as well as the adaptive sorting algorithms Splaysort [60] and Splitsort [53], and standard Quicksort and Merge sort. The experiments study adaptivity with respect to the *Inv* measure, and the bulk-insertion methods are shown to be very efficient when the sequence is nearly sorted. For instance, both of the new algorithms use only slightly more than n comparisons to sort a sequence of n elements that is already nearly sorted.

1.4 Organization

This dissertation is organized as follows. The next chapter gives background information on bulk updates, cache sensitivity and adaptive sorting. Results related to cache-sensitive binary search trees are the topic of Chapter 3. Chapter 4 gives auxiliary algorithms for balancing AVL trees, which are then used in Chapter 5 in presenting and examining the bulk-insertion and bulk-deletion algorithms. Chapter 6 applies bulk insertion to adaptive sorting, and Chapter 7 concludes the dissertation.

CHAPTER 2

Background

This chapter describes various kinds of binary search trees and some concepts that are used in the following chapters. Also included are introductions to cache-conscious search trees (for Chapter 3) and to the problem of adaptive sorting (for Chapter 6).

2.1 Search trees

A *search tree* is a data structure that is used to represent a totally ordered set whose elements are called *keys*. A search tree allows for inserting new keys, deleting existing ones, and searching for a given key x. In addition, search trees support searching for the successor (or next-larger) key: it is easy to find the smallest key k in the tree where $k > x$ for a given key x.

As is described by any textbook on data structures and algorithms (e.g., [43]), a search tree is implemented as a set of *nodes* that form a tree structure starting from the *root* node. Each node stores pointers to other nodes in the tree that are called its *children*. The tree structure dictates that every node except for the root has a unique *parent*, although links to the parents are typically not explicitly stored, as will be discussed in Section 2.6 below. A node that has no children is called a *leaf*; other nodes are called *internal nodes*. A *binary search tree* limits the number of children to at most two: the *left* and *right child*.

In addition to the key and child pointers, nodes often store one or more *data fields* which represent application-specific data associated with the key of the node. A search operation can then return the data fields associated with the key that was searched for. The storage of keys and data fields is discussed further in Section 2.4.

All search trees satisfy the *search-tree property*, which imposes an order for the nodes of the tree. Specifically, for binary search trees:

Definition 2.1 *A binary search tree T satisfies the (binary) search-tree property if, for all non-leaf nodes p in T, the key of the left child of p is smaller than the key of p, and the key of the right child of p is larger than or equal to the key of p.**

Most practical search trees are *balanced*: a restriction called the *balancing criterion* limits the shape of the tree so that in a tree that stores n elements each root-to-leaf path has length $O(\log n)$. Two balancing criteria are given in Sections 2.2 and 2.3. The *height* of a tree is defined to be the length of the longest root-to-leaf path (see Section 2.5).

The binary search trees considered in this thesis are balanced by executing sequences of operations called *rotations*, which are described in detail below. Rotations are small constant-time operations that change the shape of the tree while preserving the search-tree property.

2.2 AVL trees

AVL trees are the oldest form of balanced search trees, first presented by G. M. Adelson-Velsky and E. M. Landis in 1962 [1]. However, AVL trees are still widely used in practice.

An AVL tree is a binary search tree, balanced using the restriction that, for every node p, the heights of the children of p must differ by at most one. This means that the root-to-leaf path length in an n-node AVL tree is $\Theta(\log n)$, more specifically between $\log_2(n+1)$ and about $1.45 \log_2(n+1)$.

Theorem 2.1 *The longest root-to-leaf path in a binary search tree with n nodes contains at least $\log_2(n+1)$ nodes.*

Theorem 2.2 *The longest root-to-leaf path in an AVL tree with n nodes contains at most $\log_\Phi(n+1) \leq 1.45 \log_2(n+1)$ nodes, where $\Phi = (1+\sqrt{5})/2$ is the golden ratio.*

Proof. See [1,29], noting that $\log_\Phi(n+1) = (1/\log_2 \Phi) \log_2(n+1)$. □

* With this definition, a possible duplicate key needs to be stored in the right child. A symmetric definition where duplicates are stored in the left child is also possible. For clarity and as is common in the literature on search trees, this thesis does not explicitly consider duplicate keys – they are easy to incorporate into the presented algorithms if necessary.

Note that Theorem 2.1 holds for any binary tree, regardless of the balancing criterion.

In a typical AVL tree, each node stores, in addition to the key and child pointers, a *balancing direction* that describes the shape of the tree. The balancing direction in a node p stores one of three values (denoted "$\cdot$", "$-$" and "$+$") that describe how the height of the children of p differ: either both children have the same height ($\cdot$) or the left child is higher ($-$) or the right child is higher ($+$). Section 2.5 describes an alternative to the balancing directions.

2.3 Red-black trees

Red-black trees are binary search trees where the balancing criterion depends on a color assigned to each node. Each node is conceptually colored either red or black, such that a parent and child cannot both be red, leaves are black, and every root-to-leaf path has the same number of black nodes. Red-black trees were first described by L. J. Guibas and R. Sedgewick in 1978, although their article [34] was somewhat more general and used different terminology. They show:

Theorem 2.3 *The longest root-to-leaf path in a red-black tree with n nodes contains at most $2\log_2(n+1)$ nodes.*

Table 2.1 compares AVL and red-black trees in terms of complexity. Perhaps the most important differences are that single deletion in an AVL tree performs $O(\log n)$ rotations in the worst case (compared to $O(1)$ for red-black trees), and the amortized complexity for a sequence of mixed insertions and deletions is also better in a red-black tree. However, the worst-case search path is shorter in an AVL tree: a root-to-leaf path in an AVL tree has at most $1.45\log_2(n+1)$ nodes, while red-black trees can have $2\log_2(n+1)$. Since the search path length affects the performance of insertions and deletions as well as searches, AVL trees may be more efficient in applications where deletions are not very frequent.

Main-memory red-black and AVL trees are compared experimentally (using randomly generated input operations) in Chapter 3 on cache conscious search trees.

AVL *trees*	*Rotations*	*Height or balance changes*	*Ref.*
Worst-case insertions	$O(1)$	$O(\log n)$	[1]
Worst-case deletions	$O(\log n)$	$O(\log n)$	[43]
Amortized, a sequence of			
insertions	$O(1)$	$O(1)$	[59, 77]
deletions	$O(1)$	$O(1)$	[77, 83]
mixed insertions and deletions	$O(\log n)$	$O(\log n)$	[77, 83]
Worst-case search path length	$1.45 \log_2(n+1)$		[1, 29, 43]

Red-black trees	*Rotations*	*Color changes*	*Ref.*
Worst-case insertions	$O(1)$	$O(\log n)$	[34]
Worst-case deletions	$O(1)$	$O(\log n)$	[80]
Amortized, a sequence of			
insertions	$O(1)$	$O(1)$	[39]
deletions	$O(1)$	$O(1)$	[39]
mixed insertions and deletions	$O(1)$	$O(1)$	[39]
Worst-case search path length	$2 \log_2(n+1)$		[34]

Table 2.1 Comparison of AVL and red-black trees.

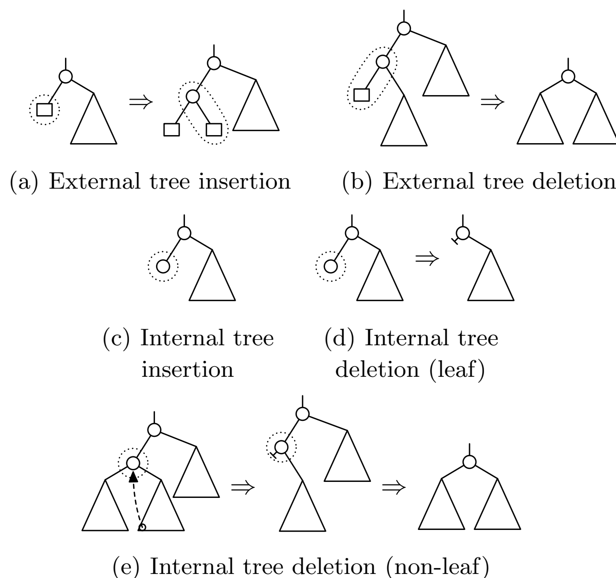
2.4 Internal and external trees

Most binary search trees come in two forms: internal and external trees. In the original *internal trees*, each node contains an actual key (and any associated data fields) of a data element stored in the tree – in other words, the number of elements stored in the tree equals the number of nodes.

In *external** or *leaf-oriented trees*, only leaf nodes store actual elements, and non-leaf nodes have so-called router keys. These router keys are used only to look for the appropriate leaf nodes, and can include keys that are no longer present in the elements stored in the tree.

External AVL and red-black trees are somewhat simpler to implement, but have many more nodes – $2n-1$ vs. n, where n is the number of elements stored in the tree. Because of the smaller space usage, internal trees are particularly suitable for the adaptive sorting application of Chapter 6.

* The term "external tree" is also often used for trees that are stored on disk. However, in this thesis, "external" always means leaf-oriented.



(a) External tree insertion (b) External tree deletion

(c) Internal tree insertion (d) Internal tree deletion (leaf)

(e) Internal tree deletion (non-leaf)

Figure 2.1 Actual insertion and deletion in binary search trees. The dotted lines indicate the nodes that the operation works on.

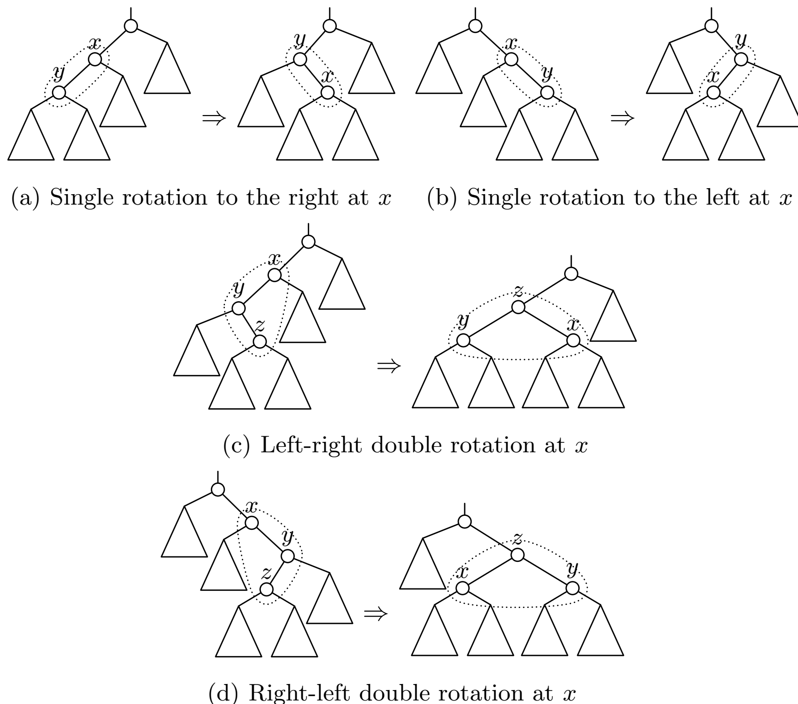
This thesis considers both internal and external trees, though internal trees were used in all of the experiments. (Previous articles on AVL-tree bulk updates [55, 77] considered only external trees.)

Most insertion and deletion algorithms can be divided into two stages. In *actual insertion or deletion*, nodes are inserted or deleted without regard to keeping the tree in balance. Afterwards, *rebalancing* puts the tree in balance, e.g., by performing rotations.

Figure 2.1 shows how actual insertion and deletion are done in external and internal trees.

In external trees, actual insertion is done by replacing a leaf node with a new internal node with two children: the old leaf and a new one (Figure 2.1(a)). Actual deletion deletes a leaf and its parent, and replaces the parent with its remaining child (Figure 2.1(b)).

In internal trees, actual insertion simply adds a new (leaf) node to an empty location in the tree (Figure 2.1(c)). Actual deletion is a bit more complicated, with two cases. When the node to be deleted is a leaf or has only one child, it is simply removed (Figure 2.1(d)). Otherwise, deleting a non-leaf node x is done by locating the node y with the next-larger key (y is the leftmost node in the subtree rooted at the right child of x), copying the key and possible associated data



(a) Single rotation to the right at x (b) Single rotation to the left at x

(c) Left-right double rotation at x

(d) Right-left double rotation at x

Figure 2.2 Binary search tree rotations. The dotted lines indicate the nodes that the operation works on.

fields to x, and then deleting y (Figure 2.1(e)). Due to the search-tree property, y has no left child (any left child would have a key between x and y), and y is deleted by replacing y with its right child (if any). A symmetric implementation that looks for the next-smaller key of x is also possible.

Rotations work the same way in both internal and external trees: 4–6 pointers are changed. The four standard rotations are shown in Figure 2.2.

2.5 Height-valued trees

The bulk update algorithms in Chapter 5 will use the variation of AVL trees called *height-valued trees* [30], where each node stores the height of the subtree rooted at the node instead of the balancing direction used in the standard AVL tree.

Height-valued trees make bulk operations much simpler, and the disadvantages are minor. Storing height values requires a small amount of space in each node – 6 or 7 bits are enough, since a red-black tree of height $2^7 - 2 = 126$ is full for the first 62 levels, and about 2^{62} nodes are difficult to fit in any kind of memory. Another minor disadvantage is that rebalancing operations often need to know the height difference of the children of a particular node – for instance, to see if the node is in balance. The height difference of the children of a node p is found directly from the balancing direction in p, but calculating it from height values requires looking at both children of p.

There exist various conflicting definitions for the height of a leaf node. This thesis defines the height of a leaf node to be 0, following what appears to be the most common definition. The height of a (sub)tree then equals the number of parent-child links traversed on its longest root-to-leaf path. As a special case (sometimes encountered in the algorithms that follow), the height of an empty (sub)tree is -1.

2.6 Parent links

Some implementations of search trees use parent links, where each node has a link to its parent in addition to the two links to its children. Parent links make many of the algorithms slightly easier to implement, but they have a proportionally rather large overhead, especially in small-node AVL trees. They also have the disadvantage that they are more difficult to maintain in a concurrent environment, since if the parent changes, all of its children need to be modified to update the parent links.

All of the algorithms given in this thesis avoid parent links by using auxiliary stacks when necessary. Sections 2.11 and 6.1 will explain this further.

2.7 Relaxed balancing

Relaxed balancing [62] is a method for balancing search trees where the rebalancing operations are decoupled from individual updates. That is, when using relaxed balancing, the individual insertion and deletion operations do not perform any rebalancing. For instance, insertion in binary search trees simply inserts a new node on the leaf level, with no rotations. Rebalancing is done periodically on the whole tree, either

as a separate batch operation or as a background process that runs concurrently with new updates.

Algorithms that use relaxed balancing have been given for various kinds of search trees, including AVL trees [48, 52, 55, 84], red-black trees [35, 47] and B-trees [45, 50]. Some solutions apply to multiple kinds of trees [51, 62]. Though the details vary, each of these has the same structure of rearranging parts of the tree in a batch process, some using the usual rebalancing operations (such as rotations) and others having their own set of rebalancing operations.

Because the concept of relaxed balancing is related to that of bulk updates, we will return to relaxed balancing in Sections 4.3 and 5.7.

2.8 Bulk updates

Bulk updates, also called group updates or batch updates, insert or delete a set of keys from the search tree in a single operation.* The primary advantage of bulk updates is that the amount of rebalancing that needs to be done after a bulk update is considerably smaller than if repeated single updates were used.

Bulk update algorithms have been presented for, among others, B-trees [46, 50, 54, 66, 68, 79], AVL trees [55, 72, 77], red-black trees [36, 49], generalized search trees called stratified trees [78], as well as various multidimensional structures (e.g., [2, 27]).

As will be explained in Section 5.1, bulk insertion algorithms generally work by collecting bulks of keys that need to be inserted in the same location in the tree, and creating balanced subtrees called update trees out of them. The update trees are inserted into the original tree as whole subtrees, and rebalancing is performed to bring the tree in balance. Bulk insertions are thus most efficient when the bulks are large, i.e., when a large number of new keys lie between two consecutive keys in the original tree.

Bulk deletion, or interval deletion, algorithms generally delete a given interval of keys (or a set of several intervals) by looking for subtrees that fall completely in the interval and detaching them from the tree. The tree is then rebalanced at the points where subtrees were

* A related bulk operation, bulk loading, constructs a new search tree from a large amount of data. However, bulk loading is not an update operation and is not considered in this thesis.

detached. Bulk deletion is also most efficient when a large number of keys falls within an interval, but in addition to efficient rebalancing, bulk deletion also has the advantage that it is not necessary to look at each individual node to be deleted. Instead, the detached subtrees can be saved in an auxiliary structure that serves as a source of new nodes for subsequent insertions [54]. More detail will be given in Section 5.5.

An important application of bulk insertions is full-text indexing of document databases using the inverted index technique [21, 49, 76]. Adding a new document into such a database involves inserting entries to the inverted index for each word that occurs in the document, and this is naturally represented as a bulk insertion. Other applications of bulk insertion include data warehouses [40], differential indexing [46,67, 79], where an index is updated when a bulk of recent updates is received from a main index, and even real-time databases [45]. As noted in the previous section, bulk updates can also be used in implementing relaxed balancing.

Chapter 5 presents bulk insertion and bulk deletion algorithms for AVL trees, and Chapter 6 applies bulk insertion to the problem of adaptive sorting.

2.9 Cache-conscious search trees

Most current computers have a hierarchical memory system, where a number of hardware caches are placed between the processor and the main memory. Caches are small but fast memories that provide faster access to recently used memory locations. Caching has become an important factor in the practical performance of main-memory data structures. Its relative importance will likely continue to increase [37, 71]: processor speeds have increased faster than memory speeds, and many applications that previously needed to read data from disk can now fit all of the necessary data in main memory. In data-intensive main-memory applications, reading from the main memory is often a bottleneck similar to disk I/O for external-memory algorithms.

There are two types of cache-conscious algorithms. This thesis focuses on the *cache-sensitive* or *cache-aware* model, where the implementation knows the specific parameters of the caches that are in use and adapts itself to them. In contrast, *cache-oblivious* algorithms attempt to optimize themselves to an unknown memory hierarchy.

The most important cache parameters for the current thesis are

the *cache block sizes*, i.e., the granularities at which data is stored in the caches and transferred between levels of the memory hierarchy. Section 3.1 will discuss the block sizes in detail.

Most of the research on the effect of caching on search trees has concentrated on variants of the B-tree [4]. The simplest cache-sensitive B-tree variant is an ordinary B$^+$-tree where the node size is chosen to match the size of a cache block (which could be, e.g., 64 or 128 bytes) [70]. A more advanced version called the Cache-Sensitive B$^+$-tree or CSB$^+$-tree [71] increases the fanout of the tree by removing most of the child pointers and storing the children of a node consecutively in memory. The CSB$^+$-tree has been further optimized using a variety of techniques, such as prefetching [16], storing only partial keys in nodes [11], and choosing the node size more carefully [37].

The above structures were optimized to only one level of the cache (often the one closest to the CPU). B-trees in two-level cache models (one level of cache plus the Translation Lookaside Buffer or TLB) are examined in [17, 69].

Several B-trees have been proposed in the cache-oblivious model, including the original rather complex cache-oblivious B-tree [7], its two simpler variants [8, 12], and a string B-tree optimized for longer keys [9]. They are based on weight-balanced B-trees [3], and on an implicit tree stored in an array (without explicit pointers) called the packed-memory array [7], whose structure and rebalancing operations are dictated by the cache-oblivious memory layout. Rebalancing in all four depends on rebuilding or copying parts of the structure, and most of the complexity bounds are amortized. Unlike the cache-sensitive B-trees, the cache-oblivious trees are not direct extensions of the usual height-balanced B-trees [4] or (a, b)-trees [39].

The node size of a cache-sensitive binary search tree cannot be chosen as freely as in the B-tree. Instead, the fixed-size nodes are placed in memory so that each cache block contains nodes that are close to each other in the tree. Binary search tree nodes are relatively small; for example, AVL and red-black tree nodes can fit in about 16 or 20 bytes using 4-byte keys and 4-byte pointers, so 3–8 nodes fit in one 64-byte or 128-byte cache block. This assumes that the nodes contain only small keys – larger keys could be stored elsewhere, with the node storing a pointer to the key.

Caching and explicit-pointer binary search trees have been considered in [41], which presents a cache-oblivious splay tree based on periodically rearranging all nodes in memory. This, as well as the cache-oblivious B-trees, is based on a cache-oblivious memory layout

known as the van Emde Boas layout [7], which is analyzed in detail in [5].

For the cache-sensitive model, [63, 64] present a one-level periodic rearrangement algorithm for explicit-pointer binary trees. A similar one-level layout (extended to unbalanced trees) is analyzed in [6]. A more complex rearrangement-based one-level layout that optimizes for space and considers unbalanced trees and non-uniform search patterns is discussed in [6, 31].

Chapter 3 begins by applying the ideas of [6, 64] in presenting an algorithm that rearranges a tree into a multi-level cache-sensitive memory layout, also optimizing an inefficient interaction with set-associative hardware caches. The presented layout can be considered to be a generalization of the one-level cache-sensitive layouts of [6, 64] and the two-level layouts of [17, 69] to an arbitrary hierarchy of block sizes.

However, the main topic of the chapter is to explore how a simple cache-sensitive layout can be preserved in the presence of insertions and deletions to standard binary search trees, without increasing the complexity of the update operations.

The experiments reported in [12,69], as well as those in Section 3.5, support the intuition that multi-level cache-sensitive structures are more efficient than cache-oblivious ones. A cache-oblivious layout has been shown to be never more than 44% worse in the number of block transfers than an optimal cache-sensitive layout, and the two converge when the number of levels of caches increases [5]. However, the cache-sensitive model is still important, because the number of levels of caches with different block sizes is small in practice (at least currently – for instance, only two in the computer used for the experiments in this thesis).

2.10 Adaptive sorting

Adaptive sorting, or the sorting of nearly sorted sequences, is the problem of sorting a sequence of values that is already "almost" in sorted order according to some intuitive notion of presortedness. One such notion is the number of inversions (pairs of elements in the input that are not in ascending order): any permutation of n distinct elements has $Inv = 0$ to $Inv = (n^2 - n)/2$ inversions.

Any sorting algorithm whose running time depends on the number of inversions is said to be *Inv-adaptive*. However, adaptive sorting is mostly concerned with the concept of optimal adaptivity. All measures of presortedness (such as *Inv*) have a lower bound for the number of

comparisons needed to sort the sequence; for *Inv*, the lower bound is $\Omega(n \log(1 + \mathit{Inv}/n))$ [56]. A sorting algorithm that reaches the lower bound up to a constant factor, i.e., sorts a sequence of n keys and *Inv* inversions in time $O(n \log(1 + \mathit{Inv}/n))$, is said to be *Inv-optimal* or *optimal with respect to Inv*.

The various measures of presortedness form a hierarchy where some measures supersede others. For instance, the measure *Rem* [20] is defined as the minimum number of keys that need to be removed so that the remaining keys form a sorted sequence. The measure *Block* [15, 65] is defined as the number of blocks of consecutive elements in the original sequence that are also present in the sorted sequence. It has been shown [15] that any *Block*-optimal sorting algorithm is also *Rem*-optimal.

The hierarchy of measures forms a partial order, as is explained in more detail in [65]. There exists in some sense a "best" measure, called *Reg*, so that any *Reg*-optimal sorting algorithm would be optimal with respect to all the other measures. However, it is an open problem whether a practical *Reg*-optimal sorting algorithm exists. The practical adaptive sorting algorithms are optimal with respect to a selection of measures lower down in the hierarchy, so the full hierarchy remains important. In addition to [65], which discusses the hierarchy of measures, the survey [26] gives an overview of research on adaptive sorting. Experimental studies on adaptive sorting algorithms include [23, 24, 26, 60].

Local insertion sort [56] is an adaptive sorting algorithm that inserts the values to be sorted into a search tree called a finger tree (described in the next section), where they will be stored in sorted order. Traditional local insertion sort inserts the values one by one. However, Chapter 6 applies AVL-tree bulk insertion to insert runs of sorted values more efficiently. For instance, the number of rotations needed to insert a bulk of m values is $O(\log m)$ in the worst case, compared to $O(m)$ for repeated single insertions.

The measures *logDist* [42] and *Loc* [65] are two equivalent measures of presortedness used to characterize local insertion sort using a finger tree. Any *Loc*-optimal algorithm is also *Inv*-optimal and *Block*-optimal [65].

Splaysort [60] is an efficient search-tree-based adaptive sorting algorithm, and the AVL-tree-based algorithms of Chapter 6 will be compared with it in the experiments in Section 6.7. Splaysort simply inserts the items to be sorted into a Splay tree [75] and produces the sorted data by traversing the final tree. Splaysort has been shown to be *Loc*-optimal via a complex proof [18, 19].

2.11 Finger trees

The local insertion sort algorithm must be able to avoid traversing a root-to-leaf path in the search tree during every insertion. Traversing an $O(\log n)$-length root-to-leaf path for each of the n values to be sorted would require $O(n \log n)$ time, so the algorithm would be no better than, e.g., Merge sort, even when the sequence is already fully sorted.

Repeated root-to-leaf path traversal is avoided by applying a *finger* [14, 33, 39, 44]. The finger saves the position of the last inserted value, and the search for the next value begins there. Thus, for example, if the next key is the successor of the previous one, it can be inserted in amortized $O(1)$ time in a typical search tree. Actual finger trees are B-tree-based and use parent links and sideways links, making them quite complex to implement.

Adaptive sorting using fingers has been studied based on, among others, (a, b)-trees [56,58] and AVL trees [22,57,82]. Instead of the most recently inserted key, the finger can also point to the node containing the largest key, as in [22, 82].

Previous AVL-tree-based sorting approaches [22,57,82] implemented the search from the finger to the place of the next key either by modifying the tree structure and rebalancing operations [82], or by using a sequence of AVL trees with increasing heights, which are sometimes split and merged together according to specific rules [22, 57].

Chapter 6 presents an implementation of the finger that does not need modifications to the tree nodes and uses a single standard AVL tree – and can therefore apply the AVL-tree bulk-insertion algorithm of Chapter 5.

The *simplified finger* of Chapter 6 is based on a *saved path* [61], a notion used to enhance concurrency when traversing index structures in databases. The saved path is a stack that stores the previously used root-to-leaf path. This simplified finger has the disadvantage that it is not as efficient in the worst case as the full finger tree, because it is not possible to move sideways in the tree.

In terms of the measures discussed in the previous section, the simplified finger leads to an *Inv*-optimal algorithm, while the full finger tree gives *Loc*-optimality. Removing sideways links from the full finger tree, while keeping parent links, results in a finger that is restricted in the same way, i.e., *Inv*-optimal but not *Loc*-optimal. Such a restricted finger tree is discussed in the context of (a, b)-trees in [39], which also very briefly mentions the idea of using an auxiliary stack instead of

parent links. The simplified finger applies this idea to binary trees, with a focus on optimizing the number of comparisons.

The other AVL-tree-based sorting algorithms [22, 57, 82] are also *Inv*-optimal but not *logDist*- or *Loc*-optimal.

Besides the easier implementation, an advantage of the simplified finger over the full finger tree is that, because parent or sideways links are not needed, the tree uses a smaller amount of space per inserted key. Each node in the internal AVL tree used in Chapter 6 stores the key, two pointers to the children, and the height of the node – a total of between $3n$ and $4n$ pointer-sized words of space if there are n values to be sorted and keys are pointer-sized or smaller.

The "hand" structure of [10] provides a *Loc*-optimal finger using an auxiliary data structure similar to the saved path. However, this structure is designed for degree-balanced trees such as (a, b)-trees, and not for rotation-using binary search trees with leaves at different depths. Moreover, the hand is much more complex to implement than the simplified finger of Chapter 6.

The technique of saving a root-to-leaf path is also used in most other algorithms in this thesis, such as in the rebalancing phase of bulk updates considered in Chapter 5. However, the adaptive sorting application extends the technique to be more like the database saved path: the simplified finger is stored between separate tree operations (searches, insertions, deletions) and used when traversing the tree to find the location of the next operation.

CHAPTER 3

Cache-sensitive binary search trees

This chapter examines the placement of binary tree nodes in memory so that search operations are cache-efficient. The chapter begins by introducing a simple cache model and a multi-level memory layout optimized for it, as well as an algorithm to produce this layout by making a copy of the tree. However, the main focus is on an incremental algorithm that maintains a single-level cache-sensitive layout by executing a constant-time operation whenever the tree is changed. This algorithm can be applied to any binary search tree that uses rotations for balancing. The chapter ends with an experimental evaluation of the various algorithms applied to AVL and red-black trees, including a comparison with cache-sensitive B-trees.

The main results of this chapter appear in the article [73] by the author and supervisor of this thesis, though with less detail. Please see Section 2.9 for a short introduction to cache-conscious search trees.

3.1 Cache model

The memory architecture of a typical modern computer consists of several levels of fast but relatively small cache memories before the slower and larger main memory. Data is transferred between consecutive levels of the memory hierarchy in *cache blocks*, which have a fixed size at every level.

Figure 3.1 gives an example of a hypothetical memory hierarchy with two levels of cache memories using 64-byte and 256-byte cache blocks. When a word of 4 bytes is read from memory and is not present in either of the caches, the 256-byte cache block containing the word is first copied from main memory into the larger cache. Then the 64-byte cache block containing the word is copied from the larger cache to the

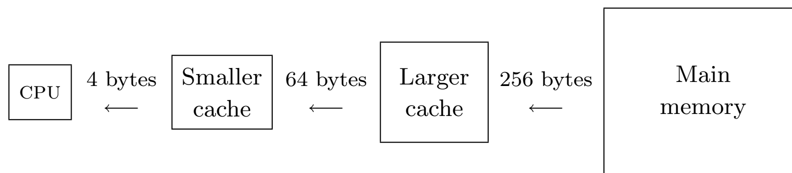


Figure 3.1 Reading a 4-byte word in a hypothetical memory hierarchy.

smaller one, and finally the 4 bytes are read into the CPU from the smaller cache.

The cache model used in the discussion below consists of a k-level cache hierarchy with block sizes $B_1, \ldots, B_k$ (in bytes) at each level. Also, B_0 is defined to be the node size of the tree that is to be laid out in the cache, in bytes, and $B_{k+1} = \infty$. All nodes are assumed to have the same size.

In addition to the data caches described above, the mapping of virtual addresses to physical addresses used by multitasking operating systems employs another kind of cache: the *Translation Lookaside Buffer* or TLB cache. Although the TLB does not work quite like the other caches, the algorithmic effect is similar: it is faster to fetch two nodes from the same TLB block than from two separate blocks. The TLB can thus be modelled as an additional level of cache in the hierarchy. The block size of the TLB cache is usually the page size of the computer (e.g., 4096 bytes), although larger blocks are sometimes also used.

A concrete example of the cache hierarchy is given by the computer used for the experiments in Section 3.5. Its AMD Athlon XP processor has two levels of data caches: a 64 Kb level 1 ("L1") cache and a 512 Kb level 2 ("L2") cache. However, both caches have 64-byte blocks, so they only use one level in the hierarchy of the cache model. The TLB cache uses 4096-byte pages (a small number of 2 Mb and 4 Mb TLB blocks also exist in the TLB of this processor, but these were not used in the experiments). The parameters of the full hierarchy used in the experiments are thus:

$k = 2$
$B_0 = 16$ (the binary trees used have 16-byte nodes)
$B_1 = 64$ (the block size of the L1 and L2 caches)
$B_2 = 4096$ (the page size of the TLB cache)
$B_3 = \infty$.

The algorithms of this chapter are assumed to know the cache pa-

rameters B_1 to B_k. In practice, the parameters can be inferred by measuring the time taken by selected memory accesses, or from the CPU model or from metadata stored in the CPU.

The algorithms assume that for $i > 1$, each block size B_i is an integer multiple of B_{i-1} – but this is trivially satisfied, since the values of B_i with $i > 0$ are always powers of 2 in actual hardware caches. Additionally, if B_1 is not an integer multiple of the node size B_0, a node should not cross a B_1-block boundary, so that it is never necessary to fetch two cache blocks from memory in order to access a single node. To achieve this, the allocation of new nodes must be modified so that the last ($B_1 \bmod B_0$) bytes of each B_1-block are not used.

The main efficiency measure used below to analyze search trees is the *B_i-block search path length*:

Definition 3.1 *The B_i-block search path length, denoted P_i, $0 \le i \le k$, is the length of a root-to-leaf path in a tree, measured in the number of separate cache blocks of size B_i encountered on the path.*

Thus, P_0 is the traditional search path length in nodes (assuming that the search ends at a leaf), P_1 is the number of separate B_1-sized cache blocks encountered on the path, and so on. Note that traditional balanced search trees attempt to minimize P_0 (of a random search, i.e., averaged over all root-to-leaf paths), and that $P_{i-1} \ge P_i$, since each B_{i-1}-block is completely contained in a single B_i-block.

3.2 Multi-level cache-sensitive layout

It is fairly easy to think of a good single-level cache-sensitive layout for a tree. For instance, assume that four binary search tree nodes fit in a single cache block ($B_1 = 4 \cdot B_0$). Then the block that contains the root node should include nodes that are closest to the root, namely its two children and one of the grandchildren. The next blocks should contain nodes in a similar arrangement: one node, its children, and one of its grandchildren in a single block (see Figure 3.2).

The above layout can be generalized to a multi-level layout in a simple manner. The single-level layout conceptually forms another tree, where each single-level block is a “super-node” with larger than binary fanout. Lay out this tree (whose nodes are B_1-sized blocks) using the single-level layout (with block size B_2) to produce a two-level layout, and repeat this for each of the k levels in the memory hierarchy.

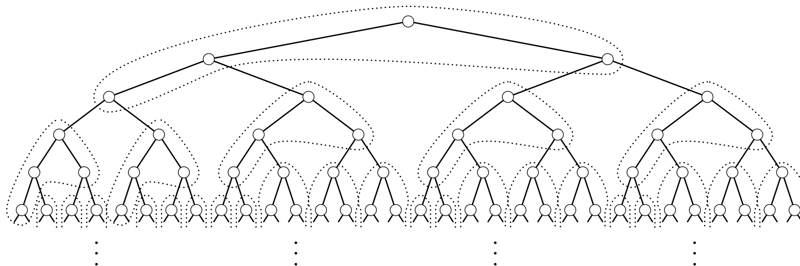


Figure 3.2 An optimal single-level memory layout for a binary tree, with $B_1 = 4 \cdot B_0$.

3.2.1 Analysis

The following analysis of the single- and multi-level layouts is restricted to complete trees, i.e., ones where all root-to-leaf paths have the same length. This makes the analysis independent of the balancing strategy of the underlying search tree, since the top part of any balanced binary search tree (up to the level of the highest leaf) is a complete tree. For instance, in a red-black tree of height h, the top part of the tree that forms a complete tree has height at least $h/2$ (Theorem 2.3). In an AVL tree, the top part has at least height $h/1.45$ (Theorem 2.2).

The single-level layout described above is optimal with respect to random searches in a complete binary tree, in the sense that the expected value of P_1 cannot be made smaller:

Theorem 3.1 *When the single-level layout described in Section 3.2 is applied to a complete binary tree, the expected value $E[P_1]$ of a random search is optimal.*

Proof. (Outline; see [31] and [6] for detailed proofs that apply to a more complex situation.) Consider the cache block C that contains the root node r of the tree. To optimize the B_1-block path length, C should be full and should contain a connected part of the tree [31, Lemma 4.3]. Of all the possibilities for which nodes to place in C, the minimum $\sum P_1$ (where the sum is taken over all root-to-leaf paths) is reached by using the topmost nodes, i.e., those found with a breadth-first search starting from r. The same argument applies recursively to each of the cache blocks containing the children of the nodes in C (i.e., the "grey" or border nodes in the breadth-first search). □

With a non-uniform distribution of searches or an unbalanced tree,

the layout of Section 3.2 is not optimal – [6, 31] give optimal and near-optimal single-level solutions for this more complex case.

On the level of B_1-blocks, the multi-level layout is exactly the same as the single-level layout. Thus:

Theorem 3.2 *When the multi-level layout described in Section 3.2 is applied to a complete binary tree, the expected value $E[P_1]$ of a random search is optimal.*

On cache block levels $i > 2$, the multi-level layout is not optimal. It is not possible to be optimal on all levels simultaneously [5], because an optimal level l layout cannot always be created by combining optimal level $l-1$ blocks. The multi-level layout of this chapter resolves the tradeoff by keeping the optimal lowest-level blocks intact at the cost of possible suboptimality on higher levels. An improved multi-level layout could be achieved by analyzing the relative costs of cache misses at each level. However, this approach would need more information about the cache than just the block sizes.

The following result can be used to calculate worst-case values of P_i in the multi-level layout of this chapter, for any level i of a given multi-level cache hierarchy.

Theorem 3.3 *Assume that a complete binary tree of height h has been laid out in the multi-level layout described in Section 3.2. Then, for $0 \le i \le k$, the worst-case B_i-block path length is*

$$P_i = \lceil h/h_i \rceil, \text{ where } h_0 = 1$$
$$h_i = h_{i-1} \cdot \lfloor \log_{d_{i-1}}(B_i/B_{i-1} + 1) \rfloor$$
$$d_0 = 2$$
$$d_i = \begin{cases} B_i/B_0 + 1 & \text{if } B_1 \text{ is an integer multiple of } B_0 \\ (d_{i-1} - 1) \cdot \lfloor B_i/B_{i-1} \rfloor + 1 & \text{otherwise.} \end{cases}$$

Proof. Consider a cache block level $i \in \{1, \ldots, k\}$. Each level $i-1$ block produced by the layout (except possibly for blocks that contain leaves of the tree) contains a connected part of the tree with $d_{i-1} - 1$ binary tree nodes. Each of these blocks can be thought of as a "super-node" with fanout d_{i-1}. A level i block is formed by allocating B_i/B_{i-1} of these super-nodes in breadth-first order (i.e., highest level $i-1$ block first). The shortest root-to-leaf path of the produced level i block has h_i binary tree nodes. □

For instance, in the cache hierarchy used in the experiments ($k = 2$, $B_0 = 16$, $B_1 = 64$, $B_2 = 4096$), the calculated values are $h_1 = 2$ and $h_2 = 6$, so a complete tree of height $h = 23 = P_0$ would have worst-case

path lengths $P_1 = 12$ and $P_2 = 4$ (compare these with experimental average values from Table 3.1: for an AVL tree, average $P_0 = 22.69$, $P_1 = 10.54$, $P_2 = 3.38$).

3.2.2 Cache-oblivious layout

The multi-level layout is similar in structure to the van Emde Boas layout [7] used as the basis of many cache-oblivious algorithms, although there is a fundamental difference: in the cache-sensitive model, we cannot choose the block sizes according to the structure of the tree, as is done in the van Emde Boas layout.

In fact, the van Emde Boas layout is almost a special case of the multi-level layout, with the fixed block sizes $B_i = (2^{2^i} - 1) \cdot B_0$ (with $i = 1, \ldots, k$ where $k = 4$ or $k = 5$ is enough for trees that fit in main memory). With these block sizes, the multi-level layout of Section 3.2 is very close to the van Emde Boas layout (as described in, e.g., [6]): the only differences are that recursive subdivision is done top-down instead of bottom-up, and some leaf-level blocks may not be full. These differences are unavoidable: since the van Emde Boas layout is defined for a complete tree, it is not clear what a bottom-up van Emde Boas layout should do to a tree with leaves at different depths.

3.3 Global relocation

Algorithm 3.1 will lay out the nodes of a given tree into the multi-level memory layout of the previous section. This *global relocation algorithm* makes a copy of the tree into a newly-allocated memory area. The algorithm could be used with any kind of tree with fixed-size nodes (not just binary search trees), as long as the tree is balanced, i.e., root-to-leaf paths have similar lengths.

The new copy of the tree is still a dynamic tree – children are reached via explicit pointers (not, for example, by indexing a large array) – and the internal structure of the nodes is unchanged. The search, insert and delete algorithms do not need to be changed in any way to support the newly relocated tree. However, if updates are performed without regard to cache-sensitivity, the cache-sensitive memory layout will degrade over time. Section 3.5 examines this degradation experimentally.

The global relocation algorithm does not modify the old copy of the tree. The old copy can thus be read freely concurrently with the relo-

cation algorithm, and possibly also used in a multiversion concurrency control scheme. However, concurrent modifications are not possible.

3.3.1 The algorithm

Algorithm 3.1 works as follows. On the lowest level of the cache hierarchy ($l = 1$), the first block is filled with a breadth-first traversal of the tree starting from the root. When this "root block" is full, each of its children (i.e., the "grey" or border nodes in the breadth-first search) will become the root node of its own level 1 block, and so on. On levels $l > 1$, level $l - 1$ blocks are allocated to level l blocks in the same manner.

The breadth-first search is implemented by using a queue on each level as follows (please ignore lines 16–18 of Algorithm 3.1 while reading this description). Each recursive call RELOC-BLOCK(l, r) fills a new level l block starting from a given node r and returns any nodes that did not fit in this level l block (each returned node is a child of one of the nodes in the block). The node r is first placed in the level l queue (line 9), which is otherwise empty, and the following step (lines 11–13) is repeated until the current level l block is full: call RELOC-BLOCK$(l - 1, n)$ on a node n removed from the level l queue, and place any nodes returned from the recursive call at the end of the queue. When the current block is full, all nodes in the queue are returned to the caller (i.e., to the level $l + 1$ queue).

Level 0 blocks are defined to be single nodes (consistent with B_0 being the node size) – thus, level $l = 1$ in the algorithm works like a standard breadth-first search until the level 1 block is full. The algorithm is initialized with $l = k + 1$ (as B_{k+1} was defined to be infinite, the "level $k + 1$ block" will never be full) and $r =$ the root of the tree.

Lines 16–18 of Algorithm 3.1 are an additional, optional space optimization for the special case where there are not enough nodes to fill a block. This only happens close to the leaves of the tree, when a node has fewer descendants than fit in the block. The optimization ensures that each level l block is at least half full, as follows. If a level l block was not completely filled, this optimization attempts to allocate the next available node (taken from the level $l + 1$ queue) in the remaining space. If the subtree rooted at the next available node does not fit in the remaining space, and this remaining space consists of less than one half of the level l block (line 16), then the attempt is undone and the remaining space is left empty (line 17).

This optional space optimization actually makes the layout not

RELOCATE(r):

```
1  A ← beginning of a new memory area, aligned at a level k block boundary
2  RELOC-BLOCK(k + 1, r)        {B_{k+1} = ∞, so this relocates everything}
```

RELOC-BLOCK(l, r):

```
1  if l = 0 then
2     Copy node r to address A, and update the link in its parent.
3     A ← A + B_0
4     return children of r
5  else
6     S ← A
7     E ← A + F(A, l) − B_{l−1}
8     Q ← empty queue
9     put(Q, r)
10    while Q is not empty and A ≤ E do
11       n ← get(Q)
12       c ← RELOC-BLOCK(l − 1, n)
13       put(Q, all nodes in c)
14    if Q is not empty then
15       A ← start of next level l block (= E + B_{l−1})
16       if F(S, l) < B_l/2 then        {less than half of the block was free}
17          Free the copies made above, i.e., all nodes at addresses S to A−1.
18          return r        {our caller will try to relocate r again later}
19    return remaining nodes in Q
```

Algorithm 3.1 The global relocation algorithm. The argument r is the root of the tree to be relocated. The address A of the next available position for a node is a global variable. $F(A, l) = B_l - A \bmod B_l$ is the number of bytes between A and the end of the level l block containing A. To be able to update the link in a parent when a node is copied, the algorithm actually needs to store (node, parent) pairs in the queue Q, unless the tree structure contains parent links; this is left out of the pseudocode for clarity.

quite optimal (in terms of Theorem 3.2) at the leaf level, since a block may be allocated in $B_i/2$ space instead of using a new B_i-sized block. However, this only happens close to the leaves of the tree, so the effect on large trees is small.

Theorem 3.4 *Algorithm 3.1 rearranges the nodes of a tree into a multi-level cache-sensitive memory layout in time $O(nk)$, where n is the number of nodes in the tree and k is the number of memory-block levels.*

Proof. Each node in the tree is copied to a new location only once, except that the space optimization (line 17) may undo (free) some of these copies. This undo only happens when attempting to fill a level l cache block that was already more than half full, and the layout is then restarted from the next level l cache block, which is empty. Thus, an undo concerning the same nodes cannot happen again on the same level l. However, these nodes may already have taken part in an undo on a level $l' < l$. In the worst case, a node may have taken part in an undo once on all k memory-block levels. Each of the n nodes can thus be copied at most k times.

Consider then the queues Q at various levels of recursion. Each node enters a queue at level $l = 1$ (line 13, using c from line 4), and travels up to a level $l' \leq k + 1$, where it becomes the root of a level $l' - 1$ subtree and descends to level 0 in the recursion. Thus, each node is stored in $O(k)$ queues. □

3.3.2 Aliasing correction

Some experiments done with the multi-level layout (Section 3.5) indicated that it, as well as other multi-level cache-sensitive layouts, can suffer from a problem called *aliasing*, a kind of repeated conflict miss caused by set-associative hardware caches. The problem appears to be specific to multi-level cache-sensitive layouts – for instance, [28] reports that associativity has very little effect on non-cache-optimized search trees.

Many hardware caches are d-way set associative ($d \in \{2, 4, 8\}$ are common), that is, there are only d possible places in the cache for a block with a given address A. Typically certain bits of A specify which set of d possible locations is used for the cache block. See [38] for a more detailed explanation of associativity and an evaluation of its effects on normal non-cache-optimized programs.

A multi-level cache-sensitive layout for a tree can result in inefficient interaction with set-associative caches as follows. If the multi-level

```
TRANSLATE-ADDRESS(A):
1 for i in 1 to k do
2     u ← (A ÷ B_i) · B_i
3     l ← A mod B_{i−1}
4     t ← (A − u − l) ÷ B_{i−1}
5     t′ ← (t + A ÷ B_i) mod (B_i/B_{i−1})
6     A ← u + l + t′ · B_{i−1}
7 return A
```

Algorithm 3.2 Address translation for aliasing correction. If the block sizes B_i are powers of two as is usual, bit operations can be used instead of $\div$ (integer division) and mod. This translation is applied to every address used in lines 2 and 18 of Algorithm 3.1. The other addresses S, A and E do not need to be translated, because they are only used to detect block boundaries.

layout is too regular, set associativity may map several cache blocks on a root-to-leaf path to the same location in the cache. For instance, the ith cache block in each TLB page is often mapped to the same set of d locations. If the ith cache blocks of several TLB pages are accessed, the cache can then only hold d of these blocks, even though the capacity of the cache is much larger than d blocks.

A straightforward multi-level cache-sensitive layout, including the one produced by Algorithm 3.1, fills a TLB page (of size B_l for some l) with a subtree so that the root of the subtree is placed at the beginning of the TLB page (in its first B_{l-1}-sized cache block). Then, for example, when a particular root-to-leaf path is traversed in a search, only d root nodes of these B_l-sized subtrees can be kept in the set-associative B_{l-1}-block cache.

The problem is not specific to TLB pages but to any multi-level cache hierarchy with set-associative caches on lower levels of the hierarchy. Also, the root of the B_l- or TLB-sized subtree is not of course the only problematic node, but the problem is most pronounced at the root.

Fixing the problem is simple, because the B_{l-1}-blocks inside a TLB page (B_l-sized block) can be ordered freely: there is no reason to have the root of a subtree in the first block of the page. The B_l-sized TLB page consists of B_l/B_{l-1} cache blocks, and the subtree located in the TLB page can use these cache blocks in any order. We can simply use a different ordering for separate TLB pages, so the root node of the subtree will not always be located in the first cache block.

Algorithm 3.2 implements the reordering by performing a cache-sensitive translation of the addresses of each node allocated by the

global relocation algorithm. Every address A can be partitioned into components according to the cache block hierarchy: $A = A_k \dots A_2 A_1 A_0$, where each A_i, $i \in \{1, \dots, k-1\}$, has $\log_2 B_i/B_{i-1}$ bits of A, and A_0 and A_k have the rest. For each level $i = \{1, \dots, k\}$, the upper portion $A_k \dots A_{i+1}$ is simply added to A_i, modulo B_i/B_{i-1} (so that only the A_i part is changed).

For example, if B_l is the size of the TLB page, the root of the first allocated TLB page ($A_k \dots A_{l+1} = 0$) will be on the first cache block (the translated portion $A'_l = 0$), but the root of the second TLB page (which is a child of the first page) will be on the second cache block ($A_k \dots A_{l+1} = 1$, so $A'_l = 1$) of its page.

It would be enough to apply this correction to those memory-block levels with set-associative caches on the previous level (i.e., level l in the above example, since level $l-1$ has the set associative cache). However, it is simpler to do it on all levels, because then the cache-sensitive algorithms only need knowledge of the block sizes and not any other parameters of the cache hierarchy. Applying the translation on every level increases the time complexity of Algorithm 3.1 to $O(nk^2)$, but this is not a problem in practice, since k is very small (e.g., $k = 2$ was discussed in Section 3.1).

3.4 Local relocation

When insertions and deletions are performed on a tree which has been relocated using the global algorithm of the previous section, each update may disrupt the cache-sensitive memory layout at the nodes that are modified in the update. This section discusses modifications to the insert and delete algorithms that try to preserve a good memory layout without increasing the time complexity of insertion and deletion in a binary search tree that uses rotations for balancing (such as an AVL tree or red-black tree). These algorithms can be used either together with the global relocation algorithm of the previous section (which could be run periodically) or completely independently.

3.4.1 The invariant

Local relocation will preserve the following memory-layout property:

Invariant 3.1 *For all non-leaf nodes x, either the parent or one of the children of x is located on the same B_1-sized cache block as x.*

This property sets an upper limit for the average B_1-block path length (averaged over all root-to-leaf paths in the tree), and so improves the worst-case memory layout. For simplicity, the proof only considers a complete binary tree of height h. To see that Invariant 3.1 improves the memory layout of, e.g., a red-black tree, remember that the top part of a red-black tree of height h is a complete tree of height at least $h/2$ (the bottom part also contains smaller complete trees).

Theorem 3.5 *Assume that a complete binary tree of height h has been laid out in memory so that Invariant 3.1 is satisfied. Then the expected value of the B_1-block path length of a random search is*

$$E[P_1] \leq 2h/3 + 1/3.$$

Proof. In a specific memory layout, $E[P_1]$ is counted over all root-to-leaf paths in the tree. Thus, an upper bound for $E[P_1]$ can be derived from the worst-case memory layout, in which each B_1-sized cache block contains only the nodes prescribed by Invariant 3.1, i.e., a single leaf or a parent and child.

Assume that a node p with height h has its child c in the same cache block in order to satisfy Invariant 3.1. In the worst-case memory layout, the parent of p is in a different cache block. Consider all possible paths down from p. In a complete tree, exactly one half of the paths do not go through c, and on all of these paths, the next cache block encountered after the one containing p contains a node of height $h-1$ (specifically, the other child of p). The other half of the paths go through the child c, and on all of these paths, the next cache block contains a child of c, i.e., a node with height $h-2$.

Starting with $p =$ the root of the tree and proceeding recursively leads to the following recurrence for the expected value of the B_1-block path length:

$$\begin{aligned} P(0) &= 0 \\ P(1) &= 1 \\ P(h) &= \frac{1 + P(h-2)}{2} + \frac{1 + P(h-1)}{2}. \end{aligned}$$

Solving this gives $E[P_1|\text{worst-case memory layout}] = P(h) = 2h/3 + 2/9 - 2(-1)^h/(9 \cdot 2^h) \leq 2h/3 + 1/3$. In any memory layout, $E[P_1] \leq E[P_i|\text{worst-case memory layout}]$. □

Though it is not required for Invariant 3.1, the memory layout can be optimized somewhat further with a simple heuristic: in insertion,

a new node should be allocated in the cache block of its parent, if it happens to have enough free space.

The aliasing problem discussed in Section 3.3.2 does not apply to local relocation, because the layout produced here is only one-level (and also less regular than the layout of Section 3.3).

3.4.2 Preserving the invariant

We will call a node x *broken* if Invariant 3.1 does not hold for it. To analyze how this can happen, denote $N(x)$ = the set of "neighbors" of node x, i.e., the parent of x and both of its children (if they exist). Furthermore, say that x *depends* on y if y is the only neighbor of x that keeps x non-broken (i.e., the only neighbor on the same cache block).

Assume that the invariant initially holds in a binary search tree T. After any structure modification operation (actual insertion, actual deletion, or rotation) is done in T, some of its nodes can potentially be broken. To re-establish the invariant, the algorithm given in the next section is executed after each such operation (i.e., after every rotation and after nodes are inserted or deleted). The algorithm is given a list of 1–6 nodes that could potentially have been broken by the (single) structure modification.

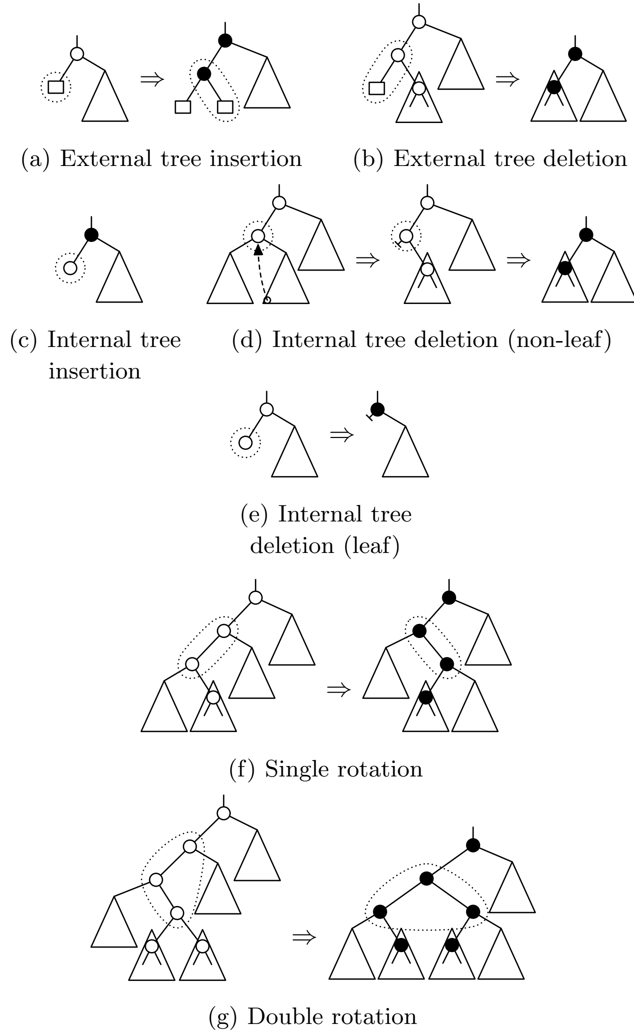
The nodes that can break in a structure modification are exactly those whose parent or either child changes, since a node will break if it depends on a node that is moved away or deleted. Figure 3.3 shows these nodes in detail. Please refer to Section 2.4 for a more detailed explanation of the various cases in the figure.

3.4.3 Fixing broken nodes

After every rotation, and after each actual insertion or actual deletion, Invariant 3.1 is re-established by invoking Algorithm 3.3 using the potentially broken nodes given in Figure 3.3.

Algorithm 3.3 uses an additional definition: $D(x)$ is defined to be the set of neighbors of node x that depend on node x (i.e., will be broken if x is moved to another cache block). Thus, $D(x) \subset N(x)$ and $0 \leq |D(x)| \leq |N(x)| \leq 3$. The algorithm uses the property is that a broken node b can be moved freely, because $D(b) = \emptyset$: all of its neighbors $N(b)$ are on different cache blocks, as otherwise b would not be broken.

Each iteration in Algorithm 3.3 first removes any non-broken nodes from the set B of potentially broken nodes. Then the first available



(a) External tree insertion (b) External tree deletion

(c) Internal tree insertion (d) Internal tree deletion (non-leaf)

(e) Internal tree deletion (leaf)

(f) Single rotation

(g) Double rotation

Figure 3.3 Broken nodes in actual insertion, actual deletion and rotations. Potentially broken nodes are filled black; the dotted lines indicate the nodes that the operation works on.

FIX-BROKEN(B):

```
1  while B ≠ ∅ do
2    Remove any non-broken nodes from B (and exit if B becomes empty).
3    if a node in N(B) has free space in its cache block then
4      Select such a node x and a broken neighbor b ∈ B. Prefer the x
       with the most free space and a b with no broken neighbors.
5      Move b to the cache block containing x.
6    else if a node b ∈ B has enough free space in its cache block then
7      Select the neighbor x ∈ N(b) with the smallest |D(x)|.
8      Move x and all nodes in D(x) to the cache block containing b.
9    else
10     Select a node x ∈ N(B) and its broken neighbor b ∈ B. Prefer a
       broken x, and after that an x with small |D(x)|. If there are multiple
       choices for b, prefer one with N(b) \ x non-broken.
11     Move b, x and all nodes in D(x) to a newly-allocated cache block.
```

Algorithm 3.3 The local relocation algorithm. B is a set of potentially broken nodes which the algorithm will make non-broken; $N(B) = \bigcup_{b \in B} N(b)$, where $N(b)$ is the set of neighbors of node b. An implementation detail is that the algorithm needs access to the parent, grandparent and great grandparent of each node in B, since the grandparent may have to be moved in lines 8 and 11.

one of three options is executed, and these two steps are repeated until B becomes empty.

The first option of the three examines the cache blocks of all neighbors of the broken nodes to find a neighbor x with free space in its cache block. If such a neighbor is found, a broken node $b \in N(x)$ is fixed by moving it to this cache block. If no such neighbor exists, the second option is to examine the cache blocks of the nodes in B: if one of them has enough space for a neighbor x and its dependants $D(x)$, they are moved to this cache block.

Otherwise, the third option is to forcibly fix a broken node b by moving it and some neighboring nodes to a newly allocated cache block. At least one neighbor x of b needs to be moved along with b to make b non-broken; but if x was not broken, some of its other neighbors may depend on x staying where it is – these are exactly the nodes in $D(x)$, and they are all moved to the new cache block. It is safe to move the nodes in $D(x)$ together with x: since they depend on x, none of their other neighbors are on the same cache block.

Clearly, applying these options repeatedly fixes any amount of broken nodes:

Theorem 3.6 *Assume that Invariant 3.1 holds in all nodes of a tree, except for a set B of broken nodes. Then giving B to Algorithm 3.3 will establish Invariant 3.1 everywhere.*

When reading the following theorem, please recall that $|B| \leq 6$ when Algorithm 3.3 is executed after a structure modification.

Theorem 3.7 *Algorithm 3.3 moves at most $4|B| = O(|B|)$ nodes in memory. The total time complexity of the algorithm is $O(|B|^2)$.*

Proof. Each iteration of the loop in Algorithm 3.3 fixes at least one broken node. Line 5 does this by moving one node; line 11 moves 2–4 nodes (b, x, and at most two other neighbors of x), and line 8 moves 1–3 nodes (x and at most two neighbors). There are at most $|B|$ iterations, and thus at most $4|B|$ nodes are moved.

Each iteration looks at $O(|B|)$ nodes, making the total time complexity $O(|B|^2)$, assuming that the amount of free space in a cache block can be found in constant time. However, a naïve implementation finds free space by looking at every node in the B_1-block to locate free positions for nodes. This increases the time complexity to $O(|B|^2 \cdot \lfloor B_1/B_0 \rfloor)$, but may actually be preferable with the small B_1 of current processors. The implementation described in Section 3.5 did this, with $B_1/B_0 = 4$.

With larger B_1/B_0, the bound of the theorem is reached by keeping track of the number of free positions for nodes in an integer stored in a fixed location inside the B_1-sized block.* To find a free node in constant time, a doubly-linked list of free nodes can be stored in the otherwise unused free nodes themselves, as is done in [64], and a pointer to the head of this list stored in a fixed location inside the B_1-block. □

A slight optimization is possible in the special case in which a leaf node n with parent p is deleted from an internal AVL tree (Figure 3.3(e)) – broken nodes can then be fixed in a simpler way. If p and n are on the same cache block, then look at the other child n' of p. If n' is not on the same cache block as p, move n' to the location where n was deleted from (in an AVL tree, n' must be a leaf node). Otherwise – if p and n are on different cache blocks, or if n' does not exist – nothing needs to be done. However, this optimization does not work for red-black trees, since there the node n' might not be a leaf.

* A few bits are enough to store this: $\log_2 \lfloor B_1/B_0 \rfloor$ must be much smaller than, e.g., the size of a single child pointer.

3.4.4 Space usage

Algorithm 3.3 includes a space-time tradeoff: it sometimes allocates a new cache block to get two nodes on the same cache block (thus improving cache locality), even though two existing cache blocks have space for the nodes. Since the algorithm always prefers an unused location in a previously allocated cache block, it is to be hoped that the cache blocks do not become very empty on average.

Instead of moving nodes to a new cache block, it would in some cases be possible to rearrange all nodes on the existing cache blocks to re-establish the invariant. However, moving unrelated nodes on the existing cache blocks "out of the way" of the currently broken nodes is not practical: moving a node x in memory needs access to its parent to update the link that points to x. But there is no simple way to find the parent, because the trees do not contain parent links, as noted in Section 2.6.

A lower limit for the cache block fill ratio can be obtained from the property that local relocation preserves: each non-leaf node has at least the parent or one child accompanying it on the same cache block. Empty cache blocks should of course be reused by new allocations.

3.4.5 Fixing broken nodes less frequently

In theory it would be possible to execute Algorithm 3.3 less frequently than was done above. For instance, Algorithm 3.3 could be run after a full sequence of rotations is done by one insertion or deletion, and not after each individual rotation. In this way fewer nodes might need to be moved, since a larger amount of broken nodes gives more freedom in how to re-establish the invariant. Also, fixing broken nodes after individual rotations has the disadvantage that the very next rotation may break some of the nodes that were just carefully fixed.

However, the complexity of Algorithm 3.3 increases in the square of the number of broken nodes. This approach would thus increase the total time complexity of insertion and deletion, which is something that the local relocation approach wanted to avoid.

3.5 Experiments

This section reports on experiments performed on the algorithms of Sections 3.3 and 3.4 on internal AVL and red-black trees. The imple-

mentations were compared to cache-sensitive B^+-trees (specifically a reimplementation of the "full CSB^+-tree" of [71]) and to standard B^+-trees with cache-block-sized (B_1-sized) nodes – the latter is called a "cB^+-tree" below. The effect of global relocation on the B-trees was also examined.

The experiments were run on a 2167 MHz AMD Athlon XP processor, with a 64 Kb level-1 data cache (2-way associative) and a 512 Kb level-2 cache (8-way associative).* Each experiment was repeated 15 times; all values given are averages of these.

As noted in Section 3.1, the cache parameters were as follows: $k = 2$, $B_0 = 16$, $B_1 = 64$, $B_2 = 4096$, $B_3 = \infty$. As noted in Section 2.6, none of the tree implementations had parent links – rebalancing was done using an auxiliary stack.

The binary tree node size $B_0 = 16$ bytes was reached by using 4-byte integer keys, 4-byte data fields and 4-byte pointers to the left and right children. The AVL tree balance and red-black tree color information was encoded in the otherwise unused low-order bits of the child pointers.

The nodes of the B-trees were structured as simple sorted arrays of keys and child pointers. The branching factor of a non-leaf node (with size $B_1 = 64$ bytes) was 7 in the cB^+-tree and 14 in the CSB^+-tree.

3.5.1 Search time, path length and space usage

In the experiments reported in Figures 3.4 and 3.5 and Table 3.1, the tree was first initialized by inserting the keys $1, \ldots, n$ in a random order (with each permutation equally likely) using single insertions. Then the cache was "warmed up" with 10^4 successful searches of random keys (uniformly distributed in the range $[1, n]$). Finally, the time taken by 10^5 searches (again of random keys that were present in the tree) was measured. The experiment was performed on different kinds of search trees (each using the same 15 sets of insert and search operations, whose results were averaged) with and without the relocation algorithms. The AVL trees are not shown in Figure 3.4, since they performed almost identically to red-black trees (as is seen in Table 3.1).

Figure 3.4 and Table 3.1 show that the search performance of red-black and AVL trees relocated using the global algorithm was close to the cB^+-tree. The local algorithm was not quite as good, but still a

* The implementations were written in C, compiled using the GNU C compiler version 4.1.1, and ran under the Linux kernel version 2.6.18.

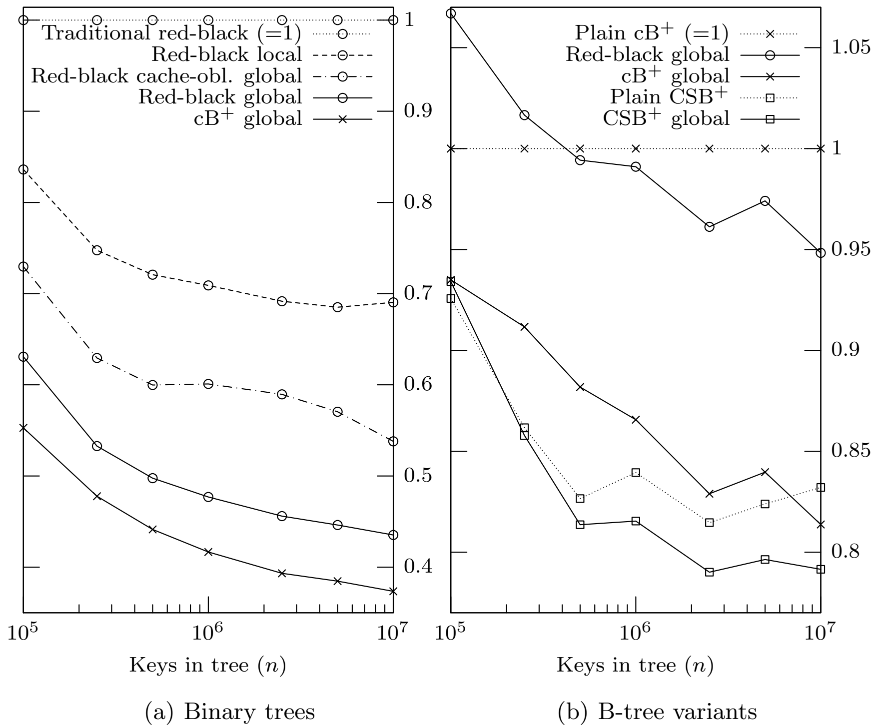


Figure 3.4 Effect of global and local relocation on search time. The figures give the search time relative to (a) the traditional red-black tree, (b) the cB$^+$-tree. The trees marked "global" have been relocated using the global algorithm (with aliasing correction where applicable), and "Red-black local" uses local relocation; the others use neither global nor local relocation.

	Search time, ns	*Average path length Nodes*	P_1	P_2	*Memory used, MB*
Traditional red-black	2313	22.77	22.61	18.06	153
Red-black local	1597	22.77	13.77	12.72	180
Red-black global cache-obl.	1244	22.77	11.93	6.19	216
Red-black global no ac	1105	22.77	10.56	3.37	173
Red-black global with ac	1007	22.69	10.54	3.38	174
Traditional AVL	2303	22.69	22.51	18.01	153
AVL local	1589	22.69	13.76	12.80	180
AVL global cache-obl.	1240	22.69	11.90	6.17	216
AVL global no ac	1100	22.69	10.54	3.38	174
AVL global with ac	1005	22.69	10.54	3.38	174
Plain cB$^+$	1062	9.00	9.00	8.52	149
cB$^+$ global no ac	962	9.00	9.00	3.03	192
cB$^+$ global with ac	864	9.00	9.00	3.03	192
Plain CSB$^+$	883	7.13	7.13	7.04	206
CSB$^+$ global no ac	851	7.13	7.13	5.08	206
CSB$^+$ global with ac	840	7.13	7.13	5.08	206

Table 3.1 Search time, path lengths and space usage for various trees, all with $n = 10^7$ keys. The trees marked "global" have been relocated using the global algorithm (with or without aliasing correction = "ac"), and the ones marked "local" use local relocation; the others use neither.

large (about 30%) improvement over a traditional non-cache-optimized binary tree. The cache-oblivious layout produced by the global algorithm (Section 3.2.2) was somewhat worse than a cache-sensitive layout, but about 40–45% better than a non-cache-optimized tree.

Table 3.1 also shows the average path lengths, counted by examining all paths in the tree, and the amount of memory occupied by the tree (in megabytes or 2^{20} bytes) after the insertions and possible relocation. It is seen that the B_1-block path length P_1 explains much of the variation in search time. In the binary trees, local relocation increased memory usage by about 18% and global relocation by about 14%. The implementation of global relocation included the space optimization given in Section 3.3.1.

Figure 3.5 (as well as Table 3.1) examines the impact of aliasing correction on global relocation. Aliasing correction had about 10–15% impact on binary trees and cB$^+$-trees, and about 5% on CSB$^+$-trees (which do not always access the first B_1-sized node of a TLB page). Especially in the B-trees, global relocation was not very useful without aliasing correction.

In summary, the multi-level cache-sensitive layout produced by global relocation improved the search time of binary search trees by 50–55%, cB$^+$-trees by 10–20% and CSB$^+$-trees by 3–5% in these experiments. The local relocation algorithm improved red-black and AVL trees by about 30%.

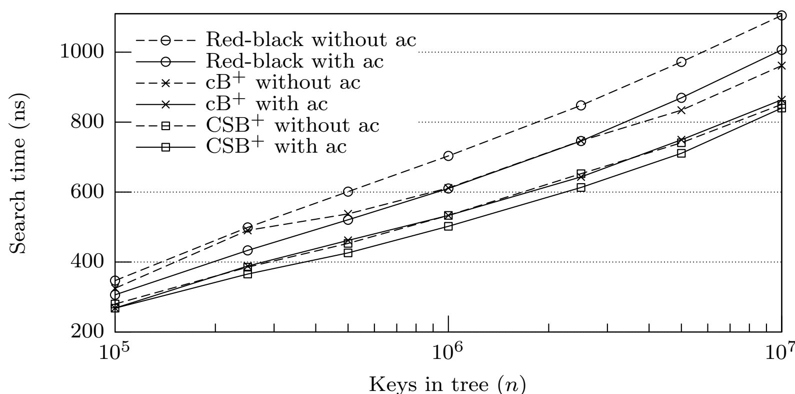


Figure 3.5 Effect of aliasing correction (= "ac") on search time.

3.5.2 Insertion and deletion time

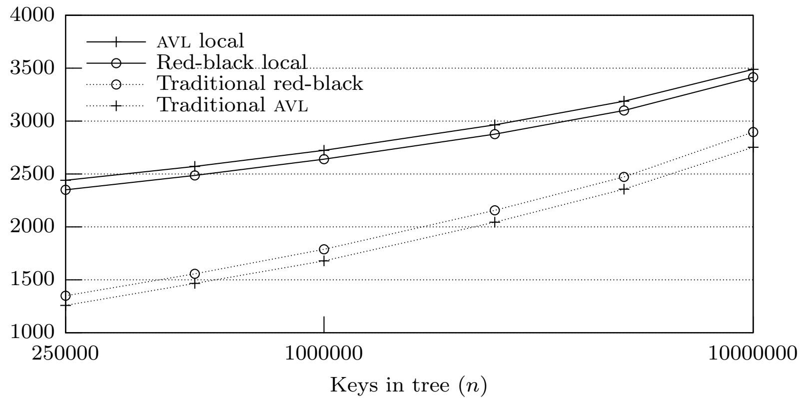
Figure 3.6 examines the running time of updates when using the local algorithm. Here the tree was initialized with n random insertions, and then $10^4 + 10^5$ uniformly distributed random insertions or deletions were performed. The times given are averaged from the 10^5 updates (the 10^4 were used to "warm up" the cache).

The local algorithm increased the insertion time by about 20–60% for $10^6 \leq n \leq 10^7$ (more with smaller n). The deletion time was affected less (about 10% faster to 10% slower for $10^6 \leq n \leq 10^7$). Random deletions in binary search trees produce less rotations than random insertions, and the better memory layout produced by the local algorithm decreases the time needed to search for the key to be inserted or deleted.

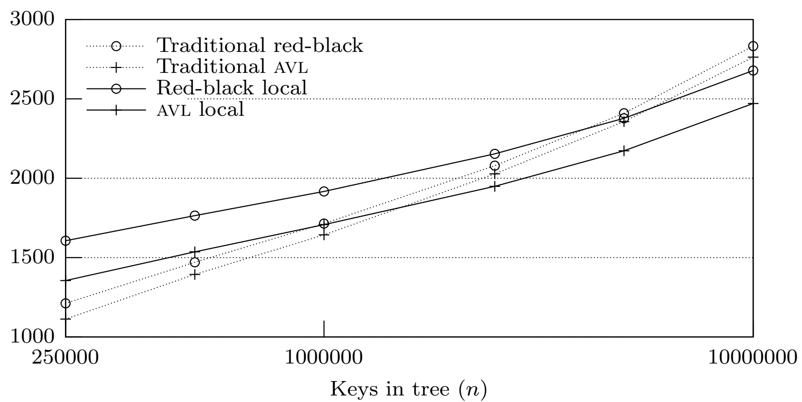
3.5.3 Degradation of locality after updates

As described above, the global relocation algorithm produces a good multi-level layout by making a copy of the tree. A natural question to ask is how well this layout is preserved when insertions and deletions are performed, either without regard to cache-sensitivity or when the local algorithm is used to retain some of the cache-sensitivity.

In the experiment reported in Figure 3.7, the tree was initialized with $n = 10^6$ random insertions. Then the global algorithm was run once, and a number of random updates (half insertions and half dele-



(a) Nanoseconds per insertion

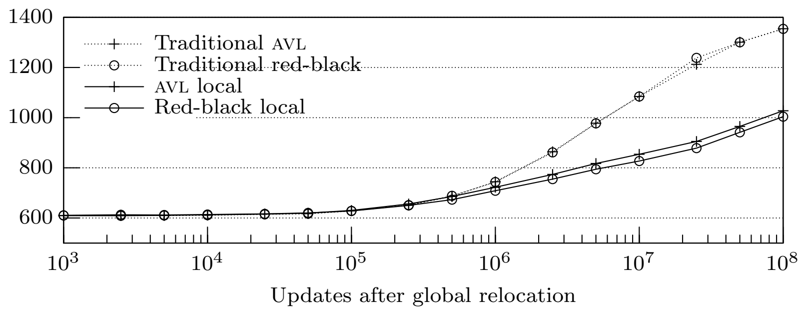


(b) Nanoseconds per deletion

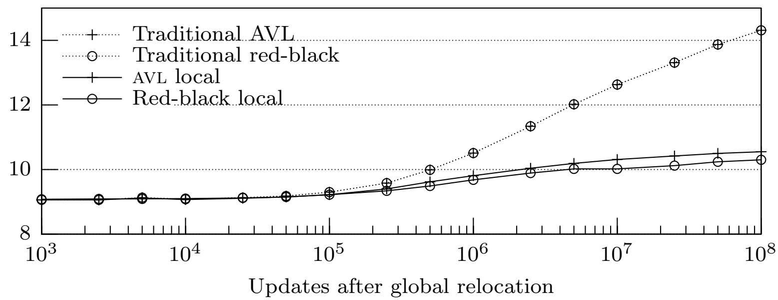
Figure 3.6 Effect of the local relocation algorithm on the time taken by (a) insertions and (b) deletions.

tions) were performed. Finally, the average search time from 10^5 random searches was measured (after a warmup period of 10^4 random searches), along with the average B_1- and B_2-block path lengths at the end of the experiment. Before about 10^5 updates after relocation, the values changed very little; for clarity, the x-axis in Figure 3.7 begins from 10^3 updates.

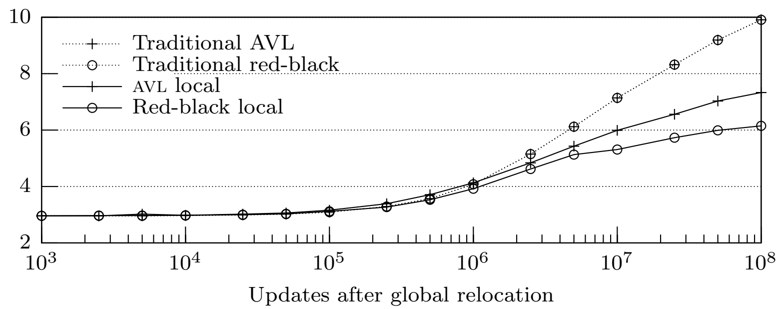
The results indicate that the cache-sensitivity of the tree decreases significantly only after about n updates have been performed. Though the local algorithm does not quite match the cache efficiency of the global algorithm, it keeps a clearly better memory layout than if traditional non-cache-sensitive updates were used.



(a) Search time (nanoseconds)



(b) Average P_1 (B_1-sized blocks)



(c) Average P_2 (B_2-sized blocks)

Figure 3.7 Degradation of locality when random insertions and deletions are performed after global relocation of a tree with $n = 10^6$ initial keys: (a) average search time from 10^5 searches, (b) B_1-block path length, (c) B_2-block path length.

CHAPTER 4

Balancing AVL trees

This chapter begins the topic of bulk updates by discussing an algorithm that balances an unbalanced AVL-tree node whose children are in balance but may have a large height difference. The algorithm is used as a subroutine in the bulk update algorithms of the following chapters, but can also be used on its own to bring an AVL tree into balance after a sequence of insertions and deletions.

Some of the material in this chapter appears in preliminary form in the author's Master's thesis [72].

4.1 The node balancing algorithm

The AVL-tree node balancing algorithm, first presented in [55], balances an unbalanced node (i.e., a node that does not currently satisfy the AVL-tree balance criterion) provided that both of its children are in balance (i.e., are valid AVL trees). The algorithm can be used, for instance, to rebalance an AVL tree which has unbalanced nodes – with any amount of imbalance – on a single-root-to-leaf path: simply run the algorithm bottom-up on all nodes on the path.

As will be shown below, an invocation of the node balancing algorithm on a node n executes $O(d)$ standard rotations, where d is the absolute height difference of the children of n. The height of the resulting subtree is the same or one lower than the original height of n.

The *height difference* at a node is here defined as $h_r - h_l$, where h_r is the height of the right child of the node and h_l is the height of the left child. The term *absolute height difference* is used for $|h_r - h_l|$. Thus, a node in an AVL tree is in balance if the absolute height difference at it is less than two.

```
BALANCE-NODE(n):
 1 P ← empty stack
 2 while n is not in balance do
 3   if the current height difference at n > 1 then
 4     if the height difference at the right child of n < 0 then
 5       DOUBLE-ROTATE-RIGHTLEFT(n)
 6     else
 7       SINGLE-ROTATE-LEFT(n)
 8   else      {the height difference at n < −1}
 9     if the height difference at the left child of n > 0 then
10       DOUBLE-ROTATE-LEFTRIGHT(n)
11     else
12       SINGLE-ROTATE-RIGHT(n)
13   push(the new parent of n, P)
14 Update the height value of n.
15 {P now contains the path from n to where n was originally}
16 while P is not empty do
17   n ← pop(P)
18   BALANCE-NODE(n)
```

Algorithm 4.1 The node balancing algorithm.

Node balancing, Algorithm 4.1, performs a sequence of rotations at the unbalanced node n. Each rotation is selected as the one that most improves the balance at n. The rotation will move node n downward in the tree: any rotation moves the node at which the rotation is performed (node x in Figure 2.2) to be a child of one of its original descendants. In addition, each rotation decreases the height difference at n (this will be shown in the proof of Theorem 4.2 below). These rotations are performed until node n is in balance or becomes a leaf (which is always in balance). At that point, some further imbalance may remain at nodes on the path up from n to the point where n was originally. This final imbalance is corrected by calling the algorithm recursively on each node on this path. As is shown in Section 4.4 below, each recursive call performs at most one rotation.

4.2 Standard insertion and deletion

The rebalancing strategy of the standard AVL-tree insertion and deletion algorithms (as given by Knuth [43], for example) can be described very concisely using the node balancing algorithm.

The standard insertion algorithm performs at most one rotation.

SINGLE-INSERTION(k):
1 Search for k, saving the search path in $P[1..n]$ (where $P[1]$ is the root).
2 Insert a new node with key k as the appropriate child of $P[n]$.
3 **for** each node $P[i]$ in P from down to up **do**
4   Update the height value of $P[i]$.
5   **if** $P[i]$ is in balance and the height value did not change **then**
6   **return**   {*because nodes* $P[1..i-1]$ *are already in balance*}
7   BALANCE-NODE($P[i]$)

Algorithm 4.2 The AVL tree single insertion algorithm in terms of Algorithm 4.1.

This rotation is the same that would be done if the node balancing algorithm were executed bottom-up on every node on the path from the newly inserted node up to the root of the tree. Similarly, the standard deletion algorithm performs rotations as if the node balancing algorithm were executed bottom-up on every node on the path from the parent of the deleted node up to the root.

Algorithms 4.2 and 4.3 give a more complete description of the standard algorithms (for internal trees) in terms of the node balancing algorithm. The standard algorithms perform slightly less computation than these, because the condition where the algorithm ends can be made more exact when using a specialized rebalancing algorithm instead of the generic node balancing algorithm. For instance, the standard single insertion algorithm can exit immediately after the first rotation is done. However, Algorithms 4.2 and 4.3 perform exactly the same rotations as the standard algorithms.

4.3 An implementation of relaxed balancing

The concept of relaxed balancing, described in Section 2.7, brings up another use for the node balancing algorithm. Relaxed balancing of AVL trees can be implemented using the node balancing algorithm as follows [55,84]. Each individual insertion and deletion operation should mark the parent of each new or deleted node by invalidating its height value (by setting it to -1, for example), but do no rotations.

The rebalancing process (executed periodically or as a concurrent background process) should traverse the tree in a bottom-up fashion, e.g., in post-order, and execute the node balancing algorithm on any nodes whose height value is found to be invalid. After each execution of the node balancing algorithm on a node n, the height value of the

SINGLE-DELETION(k):

```
1  Search for k, saving the search path in P[1..n] (where P[1] is the root).
2  if k was not found then
3    return        {nothing to delete}
4  if P[n] is a leaf node then
5    l ← n
6  else
7    Starting from the right child of P[n], descend left as far as possible to
     find the next-larger node P[l]. Save this search path in P[n + 1..l].
8    Copy the key and data from P[l] to P[n], overwriting those in P[n].
9  Delete P[l] from the tree and from P.
10 for each node P[i] in P from down to up do
11   Update the height value of P[i].
12   if P[i] is in balance and the height value did not change then
13     return      {because nodes P[1..i − 1] are already in balance}
14   BALANCE-NODE(P[i])
```

Algorithm 4.3 The AVL tree single deletion algorithm in terms of Algorithm 4.1. An equivalent implementation finds the next-smaller node instead of the next-larger one on line 7.

parent of n should be marked as invalid for rebalancing to propagate upwards. If necessary, e.g., for concurrency, the rebalancing process may be aborted after any invocation of the node balancing algorithm (after invalidating the height value of the parent) and later continued or restarted.

Since relaxed balancing needs to take several insertions and deletions into account, it is closely related to the concept of bulk updates. This relation will be further explored in Section 5.7.

4.4 Analysis of node balancing

The following two theorems formalize the properties described in Section 4.1. Similar results appear in [55], but here the proofs are more detailed and give some additional properties that are needed in the bulk update algorithms.

Theorem 4.1 shows that Algorithm 4.1 balances the subtree rooted at the unbalanced node n. The theorem also gives the new height of this subtree, as well as some more specific properties that will be used later. Theorem 4.2 shows that the number of rotations performed is proportional to the original absolute height difference at n.

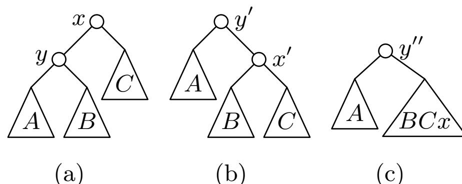


Figure 4.1 Single rotation case in proof of Theorem 4.1. (a) Before the rotation. (b) After a single right rotation at x. (c) After the subtree rooted at x' has been balanced, resulting in a subtree BCx containing node x' and the nodes of B and C.

Theorem 4.1 *Executing the node balancing algorithm on a subtree S, balanced elsewhere but possibly not at its root, results in a subtree T, all of whose nodes are in balance. The height of T is the same or one less than the height of S: $h_S - 1 \leq h_T \leq h_S$. Also, $h_T = h_S$ only if either the root of S is initially in balance or both children of the higher child of the root have the same height. If $h_T = h_S$ and the root of S is not in balance, then the children of the root of T will have different heights.*

Proof. The proof is by induction on the height of the subtree given to the algorithm. The base case is trivial: all trees of height less than 2 are in balance (when the height of a leaf is 0), so the algorithm does nothing. Thus, $T = S$ is in balance and $h_T = h_S$. The same applies to any larger subtrees where the root is initially in balance.

If the root of the subtree is not in balance, the algorithm does one of four rotations, giving four cases to consider. We first consider the case shown in Figure 4.1, where a single right rotation is performed – the single left rotation case is a mirror image of this one. This case is executed when $h_A \geq h_B$, using the notation of Figure 4.1, and the height difference at the root x of the subtree is less than -1 (i.e., $h_y > h_C + 1$), which together imply that $h_B \geq h_C$.

The single right rotation results in the tree shown in Figure 4.1(b). The algorithm proceeds to rebalance the node x', which is the same node as the original x, but now lower down in the tree. The height $h_{x'} = h_B + 1$, which is smaller than the original height $h_S = h_x \geq h_B + 2$ (see Figure 4.1(a)). We can thus use the induction hypothesis to conclude that the algorithm balances the subtree rooted at x', resulting in a balanced subtree BCx that contains node x' and the nodes of B and C (Figure 4.1(c)). Here h_{BCx} is either h_B or $h_B + 1$.

We also need to consider node y'', the parent of the root of the subtree BCx. Because $h_A \geq h_B$ and node y is in balance, h_A is either

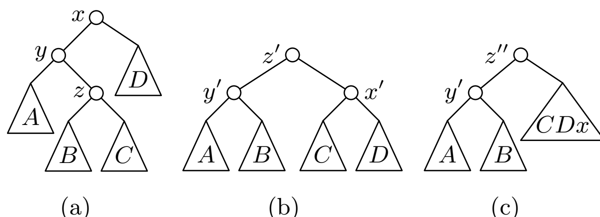


Figure 4.2 Double rotation case in proof of Theorem 4.1. (a) Before the rotation. (b) After a left-right double rotation at x. (c) After the subtree rooted at x' has been balanced, resulting in a subtree CDx containing node x' and the nodes of C and D in some order.

h_B or h_B+1. Thus, y'' is in balance and the upward phase (lines 16–18) of the algorithm will do nothing at y''.

To complete the proof of the single rotation case, we need to examine the height of the resulting tree $h_T = h_{y''}$.

If $h_A = h_B$, the height of the original tree is $h_S = h_A+2 = h_B+2$, and $h_{y''}$ is either $h_A+1 = h_S-1$ (from its left child) or $h_{BCx}+1$ (from the right child). The latter is either $h_B+2 = h_S$ or $h_B+1 = h_S-1$.

If $h_A = h_B+1$, the height of the original tree $h_S = h_A+2 = h_B+3$. As h_{BCx} is either h_B or h_B+1, we conclude that $h_{y''}$ is $h_B+2 = h_S-1$. Therefore, the height h_T is either h_S or h_S-1. Also, as claimed in the theorem, $h_T = h_S$ is possible only in the case where the heights of the children of the higher child of S are the same, i.e., $h_A = h_B$, and then the children of T have different heights: one of the children is A (height h_A) and $h_T = h_S = h_A+2$, so the other child must have height h_A+1. This completes the proof of this case and its mirror image.

The remaining two cases use double rotations and are mirror images of each other. The case where a left-right double rotation is performed is shown in Figure 4.2(a) and, after the rotation, in Figure 4.2(b). In this case, $h_y \geq h_D+2$ in the terminology of Figure 4.2(a). Because y is in balance and this rotation is performed only if the right subtree of y is one higher than its left subtree, $h_y = h_z+1 = h_A+2$. Thus, $h_A+2 \geq h_D+2 \Rightarrow h_A \geq h_D$, and $h_S = h_x = h_A+3$.

Since $h_z = h_A+1$, either both h_B and h_C are equal to h_A, or one of them is h_A and the other one is h_A-1. After the rotation, $h_{y'} = h_A+1$ and y' is in balance for both possible values of h_B.*

Next, the algorithm moves on to the subtree rooted at x'. Since

* This is true also when a double rotation is done after standard single insertion or deletion – the only difference here is that $h_y - h_D$ may be greater than two.

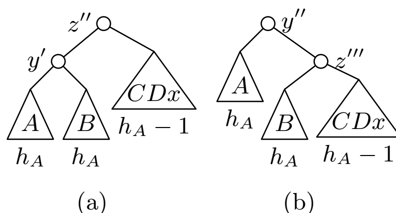


Figure 4.3 Special case of double rotation in proof of Theorem 4.1. (a) This is Figure 4.2(c) with heights given below the subtrees for this special case. (b) After a single right rotation done at z'' in the upward phase.

h_C is either h_A or $h_A - 1$, and $h_A \geq h_D$, it follows that $h_{x'}$ is either $h_A + 1$ or h_A. Thus, if x' is in balance, z' is also in balance and we are done. If x' is not in balance, we can conclude by the induction hypothesis that the node balancing algorithm balances the subtree rooted at x', resulting in a subtree that we shall call CDx (because it contains node x' and the nodes of subtrees C and D, see Figure 4.2(c)). The height h_{CDx} can be $h_A + 1$ or h_A or $h_A - 1$.

Now the tree will be in balance and $h_T = h_A + 2 = h_S - 1$, except in the special case where $h_{CDx} = h_A - 1$, which happens only if $h_{x'} = h_A$ and the height of the subtree rooted at x' is decreased by the next iteration of the downward phase. Since $h_{x'} = h_A$, h_C must be $h_A - 1$, which implies that $h_B = h_A$. This special case is shown in detail in Figure 4.3(a). Node z'' is not in balance, but the upward phase of the algorithm will correct this by performing a single right rotation in the recursive call at z'', resulting in the tree shown in Figure 4.3(b). The tree is now in balance, with height $h_T = h_{y''} = h_A + 2$. This is $h_S - 1$, which completes the proof of the theorem. Note that h_T is always $h_S - 1$ in the double rotation cases. □

It has been shown in [55] that the node balancing algorithm performs at most $2d$ rotations, where d is the absolute height difference at the root of S. The following theorem improves this bound to $d - 1$, and gives a lower bound.

Theorem 4.2 *The node balancing algorithm performs at least* $\lceil (d-1)/3 \rceil = \Omega(d)$ *rotations and at most* $d - 1 = O(d)$ *rotations, where* d *is the absolute height difference at the root of the subtree given to the algorithm.*

Proof. We will first show that each rotation executed in the downward phase (lines 2–13 of Algorithm 4.1) decreases the absolute height dif-

ference at the node that the algorithm examines. There are again two cases and corresponding mirror images.

In the right single rotation case, Figures 4.1(a) and 4.1(b), the absolute height difference at x before the rotation is either h_B+1-h_C or h_B+2-h_C. This is clear from the figure, since node y is in balance and this case is applied only when $h_A \geq h_B$ and the subtree rooted at y is higher than C, so $h_B \geq h_C$. After the rotation, the absolute height difference at x', where the algorithm continues, is $h_B - h_C$, which is clearly smaller than the above.

In the left-right double rotation case, Figures 4.2(a) and 4.2(b), the absolute height difference at x before the rotation is either h_C+2-h_D or h_C+3-h_D. In the former case $h_C \geq h_B$, and $h_C \geq h_D$ because otherwise x would be in balance. Then the absolute height difference at x' ($|h_C-h_D|$) is two smaller than at x. In the latter case $h_B = h_C+1$ and $h_C \geq h_D - 1$. If $h_C \geq h_D$, the absolute height difference at x' is three smaller than at x. If $h_C = h_D - 1$, the absolute height difference at x is 2 and at x' 1, as desired.

Thus, every rotation performed by the downward phase makes the absolute height difference at the node examined in the loop 1 to 3 levels smaller. The loop ends when the absolute height difference is smaller than 2, i.e., the node is in balance. If the original absolute height difference is d, at most $d-1$ rotations (and at least $\lceil (d-1)/3 \rceil$ rotations) are performed in the downward phase.

As noted in the proof of Theorem 4.1, the upward phase of the algorithm performs at most one rotation for each iteration of the downward phase. This rotation is performed only at some of the nodes in which a double rotation was performed in the downward phase. Furthermore, the rotation is performed only when the double rotation made the absolute height difference at least two smaller – the only case where a double rotation decreases the height difference by only one is when $h_C = h_D-1$ (above), and then node x' is in balance, so its height cannot decrease in the next iteration of the downward phase. The parent of x' will then be in balance, and the upward phase will not perform a rotation at the parent.

Thus, the above figures are not affected by taking into account the rotations done in the upward phase. $\square$

CHAPTER 5

Bulk update algorithms for AVL trees

This chapter examines bulk insertion and bulk deletion in AVL trees. The chapter focuses on the rebalancing algorithms, since they are the most complex part of the bulk update operations.

A few of the algorithms given in this chapter (but none of the proofs) appear in preliminary form in the author's Master's thesis [72]. As is described in more detail below, this work builds upon the articles [55, 77]. The idea of detaching entire subtrees in bulk deletion (Section 5.5.2) is based on the article [54] (by, among others, the author).

5.1 Single-bulk insertion

A bulk-insertion algorithm takes as input a set of new keys and a search tree, and inserts the keys into the search tree as a single operation. The bulk-insertion algorithms described in this chapter first sort the new keys using any sorting algorithm. Then they search in the tree for the smallest new key, as if doing a single insertion of this key. Instead of inserting only one key, however, the algorithms will collect all keys that go to the same place in the tree as the first one (i.e., are smaller than the next-larger key already present in the tree), and insert them together.

We will first examine a restricted situation called *single-bulk insertion*, where all of the keys to be inserted go in the same location in the tree, which is called a *position leaf*. This requires that the tree has no keys whose values are between the smallest and largest keys to be inserted. The restriction will be removed in Section 5.4 below.

The tree search for the first new key gives the position leaf where the new keys are to be inserted. In an external tree, this is an actual

BULK-INSERT(A):

1 Sort the array A of keys to be inserted.
2 Search in the tree for the smallest key s to be inserted. Save the search path in P.
3 Form an update tree S from the keys in A.
4 Insert S into the position leaf found in the search.
5 REBALANCE(S, P)

Algorithm 5.1 Single-bulk insertion of an array A of keys.

leaf node; in an internal tree, it is a null child pointer where a new node can be inserted.

The bulk insertion algorithm forms a new balanced AVL tree, called the *update tree*, from the bulk of new keys (and, for an external tree, the position leaf itself). The update tree is then inserted at the position leaf (for an internal tree) or in place of it (for an external tree). This is called the *actual insertion*.

Creating the update tree is simple. A balanced binary search tree is formed from a sorted array of keys by placing the middle key at the root of the generated tree and proceeding recursively with the left and right halves of the array. This kind of binary tree is a valid AVL tree, since it has leaves on at most two levels: some root-to-leaf paths can be one node longer than the others, but this is inevitable with an arbitrary number of nodes.

The next and final step is to *rebalance* the tree. The next section presents a very simple rebalancing algorithm, which will be used as a component in the more efficient algorithm presented afterwards. Algorithm 5.1 gives an outline of the bulk-insertion algorithm.

5.2 Log-squared rebalancing algorithm

After actual insertion has been done as specified in the previous section, the produced AVL tree is in balance except possibly at the nodes on the path from the root of the update tree to the root of the whole tree. This is because the update tree itself is in balance, and inserting it in the original tree only affects the heights of the ancestors of the update tree. Therefore, in single-bulk insertion, the rebalancing algorithm only needs to make this path balanced.

The node balancing algorithm of Section 4.1 suggests a simple strategy for rebalancing: execute the node balancing algorithm on each node

REBALANCE(S, P):
1 **for** each node n in P from down to up **do**
2 Update the height value of n.
3 **if** n is in balance and the height value did not change **then**
4 **return**
5 BALANCE-NODE(n)

Algorithm 5.2 The log-squared rebalancing algorithm for bulk insertion. The argument P is the path up from the parent of the root of the update tree to the root of the whole tree. This algorithm does not use S, which is a pointer to the root of the update tree.

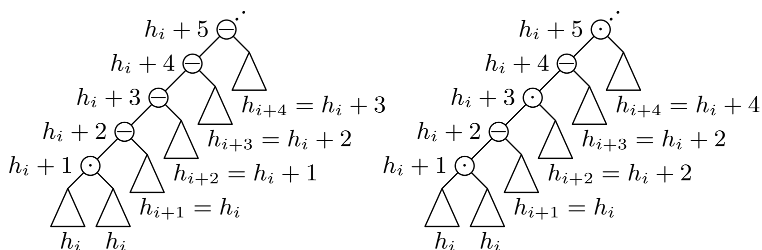


Figure 5.1 Two possibilities for minimum possible growth of consecutive siblings of a path in an AVL tree. Note that $h_{i+4} \geq h_i + 3$ in both cases. The balancing direction is displayed inside each node.

on the path up from the parent of the update tree to the root of the whole tree. Algorithm 5.2 includes the optimization that it stops early if it reaches a node which is in balance and whose height is the same as in the original tree, since then the ancestors are already in balance.

We will see below that this rebalancing strategy uses $O(\log^2 m)$ rotations in the worst case, where m is the number of keys that are inserted. Note that the number of rotations is log-squared relative to the height of the update tree, not to the height of the (possibly much larger) tree in which the insertion is performed. This algorithm was first presented in [55]. However, the analysis below is more precise than in that article, including constant factors and giving a sharper bound than can be inferred from the proofs in [55]. These more precise results are needed in the analysis of the more efficient bulk-insertion algorithm presented in Section 5.3.

Figures 5.1 and 5.2 show examples of the extreme cases of the following lemma. The lemma gives a fact about AVL trees which will be frequently used in the proofs below.

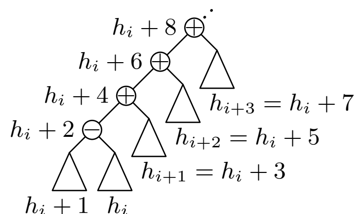


Figure 5.2 Maximum possible growth of consecutive siblings of a path in an AVL tree. Note that if $h_{j+1} = h_j + 3$, then h_{j+2} is at most $h_{j+1} + 2$. The balancing direction is displayed inside each node.

Lemma 5.1 *Consider a path $q_1, \ldots, q_n$ from a leaf q_1 to the root q_n of an AVL tree. The following hold for the siblings $r_1, \ldots, r_{n-1}$ of the nodes on this path: (a) $h_{r_i} \leq h_{r_{i+1}} \leq h_{r_i} + 3$, and (b) $h_{r_{i+j}} \geq h_{r_i} + j - 1$.*

Proof. Because of the balance condition of AVL trees, the height of a child is one or two smaller than the height of its parent: $h_{q_i} + 1 \leq h_{q_{i+1}} \leq h_{q_i} + 2$, and also $h_{r_i} + 1 \leq h_{q_{i+1}} \leq h_{r_i} + 2$. Part (a) follows directly from this. Part (b) is also trivial if we note that $h_{q_{i+1}} \geq h_{q_i} + 1$ implies $h_{q_{i+j}} \geq h_{q_i} + j$. Then $h_{r_{i+j}} \geq h_{q_{i+j+1}} - 2 \geq h_{q_{i+1}} + j - 2 \geq h_{r_i} + j - 1$. □

Lemma 5.2 *Assume that, in a balanced AVL tree, a subtree with height h is replaced by a balanced subtree with height $h + d$, where $d \geq 0$. Then Algorithm 5.2 will use at most $d^2 + 7d/2 + 4$ rotations to rebalance the tree, when called with an argument containing the path upward from the parent of the replaced subtree.*

Proof. First note that if $d = 0$, the parent is in balance, so Algorithm 5.2 does no rotations. We consider $d > 0$ in the following.

Denote by n_i the node n on iteration $i = 1, \ldots, N$ of the loop in Algorithm 5.2. To find out the number of rotations, we need to consider the heights of the children of n_i. On the first iteration ($i = 1$), one child is the replaced subtree and has height $s_1 = h + d$. The other child is the sibling of the replaced tree, with height $h - 1 \leq h_1 \leq h + 1$. On each subsequent iteration ($i > 1$), one child of n_i will be the result of the previous iteration (and its height $s_{i-1} \leq s_i \leq s_{i-1} + 1$ due to Theorem 4.1), and the other child will be the original sibling of n_{i-1}, i.e., the ith sibling on the path up from the original replaced subtree – denote its height by h_i (see Figure 5.3).

By Lemma 5.1 the heights h_i must grow by at least so much that $h_{i+j} \geq h_i + j - 1$. Assuming that node n_i is not yet in balance, The-

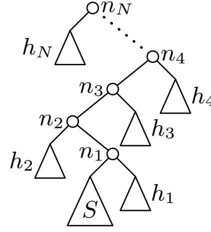


Figure 5.3 Notation used in Lemmas 5.2 and 5.3.

orem 4.1 implies that if $s_{i+1} = s_i + 1$ ($h_T = h_S$ in terms of Theorem 4.1), then $s_{i+2} = s_{i+1}$ ($h_T = h_S - 1$ on the next iteration), and thus $s_{i+2j} \leq s_i + j$ and $s_{i+j} \leq s_i + \lceil j/2 \rceil$.

When n_i is not yet in balance, $s_i - h_i > 1$, since initially $s_1 > h_1$, and $s_i - h_i$ decreases by at most 3 in one iteration, because $s_{i+1} \geq s_i$ and $h_{i+1} \leq h_i + 3$ (Lemma 5.1). If $s_{i-1} - h_{i-1} > 1$, then $s_i - h_i > -2$, which means that n_i must be in balance before $s_i - h_i \leq -2$ is possible.

On the first iteration ($i = 1$), the height difference $s_1 - h_1 \leq d + 1$. The height difference on iteration $i > 1$ is

$$\begin{aligned} s_{1+i-1} - h_{1+i-1} &\leq s_1 - h_1 + \lceil (i-1)/2 \rceil - (i-1) + 1 \\ &\leq d + 2 - \lfloor (i-1)/2 \rfloor \\ &\leq d - i/2 + 3. \end{aligned}$$

This is ≤ 1 when $i \geq 2d+4$; thus, the iteration numbered $i = 2d+3$ must be the last one. By Theorem 4.2, the number of rotations performed by each iteration is at most one less than the height difference. Then the total number of rotations executed before n_i is in balance is at most

$$\sum_{i=1}^{2d+3} (s_i - h_i - 1) \leq \sum_{i=1}^{2d+3} (d - i/2 + 2) = d^2 + 7d/2 + 3.$$

After the first iteration where n_i is in balance, the height value of n_i may still need to be increased by one from its original value, and thus one further rotation may be necessary somewhere higher up in the tree – the situation is now analogous to single insertion. After this one rotation, n_i will be in balance and the height value will not increase, and so the next iteration will be the last. Thus, the algorithm executes at most $d^2 + 7d/2 + 4$ rotations in total. □

Theorem 5.1 *Algorithm 5.2 uses at most* $\lfloor \log_2 m \rfloor^2 + 11\lfloor \log_2 m \rfloor/2 + 17/2 = O(\log^2 m)$ *rotations in the worst case to rebalance the tree after an update tree with* m *keys has been inserted in an* AVL *tree.*

Proof. The result is implied by Lemma 5.2: an empty subtree (in an internal tree) or a single leaf (in an external tree) is replaced by the update tree, with $d = \lfloor \log_2 m \rfloor + 1$ in both cases. □

It is instructive to compare Algorithm 5.2 with the single-insertion algorithm in Algorithm 4.2. The rebalancing strategy is essentially the same, because the node balancing algorithm does not care how much imbalance the insertion produced. This implies that if only one key is inserted using bulk insertion, the log-squared rebalancing algorithm performs the same (zero or one) rotations as the single-insertion algorithm.

5.3 Logarithmic rebalancing algorithm

The simple rebalancing algorithm of the previous section used $O(\log^2 m)$ rotations in the worst case, where m is the number of keys that were inserted. This can be improved by observing that this algorithm repeatedly modifies the structure of the update tree: at each step, the node balancing algorithm uses $O(\log m)$ rotations to merge only a relatively small number of keys from the original tree into the update tree.

The rebalancing procedure in Algorithm 5.3 uses only $O(\log m)$ rotations in the worst case. The idea is to first move the update tree upward in the tree using rotations that do not modify the update tree. The rotation selected on each iteration is the lowest possible rotation that moves the update tree closer to the root – see Table 5.1. After each such rotation, the node balancing algorithm is used on the node that was moved off the path from the root of the update tree S to the root of the whole tree T, to fix any imbalance created by the rotation outside this path.

This idea was presented in [77], but the algorithm of [77] does not work correctly in all cases. To fix this problem, the rotation selected by Algorithm 5.3 is different from [77], where the rotation was always done at the great grandparent of the update tree: the grandparent needs to be used instead in cases LLL, RRR, LRR and RLL of Table 5.1. This change corrects a subtle error in [77]. Figure 5.4 gives an example where Algorithm 5.3 works better.

Algorithm 5.3 moves the update tree S upward as long as its height is larger than that of a neighboring unmodified subtree of the whole tree. This is also necessarily different from [77], which looked only at

REBALANCE(S, P):

1 **while** the sibling of the great grandparent of the root of S exists and its height $\leq h_S - 1$ **do**
2   Execute the rotation from Table 5.1.
3   Do the BALANCE-NODE operation given in Table 5.1.
4   Update the beginning of P to be in accordance with the performed rotation.
5 **for** each node n in P from down to up **do**
6   Update the height value of n.
7   **if** n is in balance, the height value did not change and n is not the lowest or second-lowest node in P **then**
8   **return**
9   BALANCE-NODE(n)

Algorithm 5.3 The logarithmic rebalancing algorithm for bulk insertion. As in Algorithm 5.2, the argument S is a pointer to the root of the update tree, and P is the path up from the parent of S to the root of the whole tree.

	Tree	*Operations*	*Result*		*Tree*	*Operations*	*Result*
L L L		Single-rotate-right(y) Balance-node(y)		R R R		Single-rotate-left(y) Balance-node(y)	
L L R		Single-rotate-right(z) Balance-node(z)		R R L		Single-rotate-left(z) Balance-node(z)	
L R L		Double-rotate-leftright(z) Balance-node(z)		R L R		Double-rotate-rightleft(z) Balance-node(z)	
L R R		Single-rotate-left(y) Balance-node(y)		R L L		Single-rotate-right(y) Balance-node(y)	

Table 5.1 Rotations performed in the logarithmic rebalancing algorithm. The rotation is selected by looking at the left/right directions of the three child links above the update tree S, giving eight cases (four of which are mirror images). After the rotation, the node balancing algorithm is called on the node that was removed from the path up from the root of S.

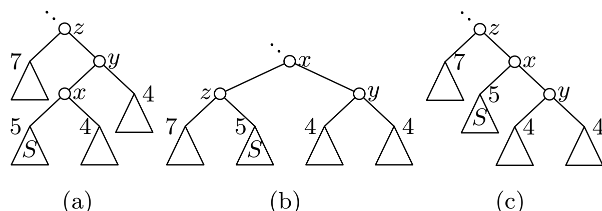


Figure 5.4 Example showing how the logarithmic rebalancing algorithm given here differs from the algorithm in [77]. Heights are given beside the subtrees. (a) Before an iteration of the upward phase. (b) The algorithm of [77] performs a right-left double rotation at node x, and now needs to continue moving the update tree S upward. (c) Algorithm 5.3 executes a single rotation to the right at node y, after which the tree is in balance.

the height difference at the grandparent. Specifically, the neighboring subtree is the sibling of the current great grandparent of the root of S (labeled h_4 in Figure 5.3) – it is the subtree closest to S that is guaranteed to be unmodified by the previously performed rotations.

After the update tree has been moved high enough, the rebalancing algorithm of Section 5.2 is used to correct any final imbalance (any remaining imbalance is on the path from the root of the update tree S to the root of the whole tree T). A minor detail is that the algorithm of the previous section (Algorithm 5.2) would finish immediately if it does not need to do any rotations and the height value does not change on an iteration. Here this optimization is not valid at the parent and grandparent of the update tree, since the preceding rotations may have changed the tree at these nodes. Thus, the code in Algorithm 5.3 removes this optimization from the first two iterations of the loop in Algorithm 5.2.

The rather lengthy proof below shows that Algorithm 5.3 performs $O(h_S)$ rotations in total, where $h_S = O(\log m)$ is the height of the update tree. The basic structure of the proof is as follows. We will see that the phase which moves the update tree upward (lines 1–4) has $O(\log m)$ iterations, each of which consists of one case of Table 5.1. Each iteration performs "amortized $O(1)$" rotations in the node balancing operation, so that the total number of rotations is $O(\log m)$. Finally, any rotations done in the second phase of the algorithm (lines 5–9) will be shown to fit in this $O(\log m)$ bound.

5.3.1 Complexity: Number of iterations

Lemma 5.3 *The upward phase (lines 1–4) of Algorithm 5.3 executes at most h_S iterations of the while loop, where h_S is the height of the update tree.*

Proof. Consider the path P up from the root of the update tree S to the root of the whole tree. Denote the heights of the direct siblings of the nodes on this path by $h_1, \ldots, h_k$ (see Figure 5.3).

The upward phase of the algorithm considers the height of the sibling of the great grandparent of S. This is initially h_4. Each iteration of the loop in the algorithm moves the update tree S upward, removing either the parent, grandparent or great grandparent of S from P (see Table 5.1). Thus, on iteration $j = 0, 1, \ldots$ of the upward phase, the height of the sibling of the current great grandparent of S will be h_{4+j}.

Because the update tree is inserted at a leaf, $h_4 \geq 1$ in an internal tree, or $h_4 \geq 2$ in an external tree, since the height of a leaf node is defined to be 0.* The first phase of the algorithm ends on the first iteration j with $h_{4+j} \geq h_S$, unless the root of the tree is reached before this. By Lemma 5.1, $h_{4+j} \geq h_4 + j - 1$, and the first phase must thus end when $j \geq h_S - h_4 + 1$. Therefore, the number of iterations in the first phase is at most h_S in an internal tree, and at most $h_S - 1$ in an external tree. □

When the update tree contains only one new key, no iterations are done in the first phase, and Algorithms 5.2 and 5.3 both execute the same (zero or one) rotations as the single-insertion algorithm.

5.3.2 Complexity: Height differences

This section will examine the heights of the siblings of the update tree and its parent, grandparent and great grandparent, at the beginning of any iteration of the first phase of the rebalancing algorithm. Name these siblings (subtrees) A, B, C and D from down to up, and their heights h_A, h_B, h_C and h_D – see Figure 5.5. (For example, before the first iteration, $h_A = h_1$, $h_B = h_2$, etc., using the terms of Lemma 5.3 and Figure 5.3.) Further define $d_1 = h_B - h_A$, $d_2 = h_C - h_B$, and $d_3 = h_D - h_C$. Note that these are all integers.

* This can be seen from Figure 5.1 by setting $h_i + 1 = 0$ and removing the subtrees with height h_i.

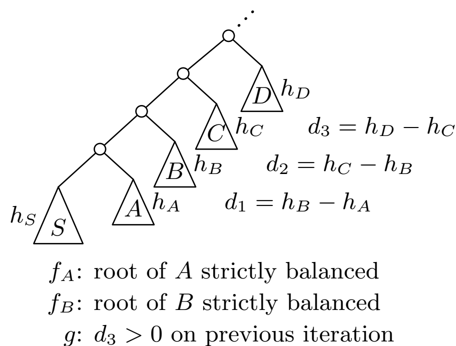


Figure 5.5 Definitions used in Section 5.3.2.

The goal of the proofs in this section is to show that the following properties hold before and after each iteration of the loop in lines 1–4 of Algorithm 5.3:

$$\begin{aligned} -2 \leq d_2 \leq 3 & \qquad \text{and} \\ d_1 \geq -2 & \qquad \text{and} \\ d_1 < 0 \Rightarrow |d_1 + d_2| \leq 2. & \end{aligned}$$

These properties are essential for the complexity proof in the next section. The properties follow from certain invariants, given below in Lemmas 5.6 and 5.7, that hold before and after each iteration.

Each iteration of the loop in lines 1–4 of Algorithm 5.3 essentially executes one of the eight cases given in Table 5.1. We will only need to consider how the subtrees A–C change, and this reduces the eight cases to the three labeled AB, BC and AC in Table 5.2. The first and fourth cases in the table lead to the same changes in the subtrees A–C.

The heights calculated in Table 5.2 refer to a variable i, whose value is either 0 or 1: $i = 0$ if the height of the subtree given to BALANCE-NODE was decreased in the BALANCE-NODE operation, and $i = 1$ otherwise. In the terminology of Theorem 4.1, $i = 1$ if and only if $h_T = h_S$.

A few additional definitions are needed for the invariants. The Boolean value f_A is defined to be true if both children of the root of the subtree A have the same height. Similarly f_B is true if both children of the root of B have the same height. The value g is defined to be true on a particular iteration if, on the previous iteration, d_3 was larger than 0. As will be seen, the value of g restricts the possible values of d_3 on the current iteration – g describes a detail of the shape of the original tree that may have been hidden by the changes made to the tree by the previous iteration.

	Before	*After*	*New heights*	*Name*
L L L			$h'_A = \max\{h_A, h_B\} + i$ $h'_B = h_C$ $h'_C = h_D$	AB
L L R			$h'_A = h_A$ $h'_B = \max\{h_B, h_C\} + i$ $h'_C = h_D$	BC
L R L			$h'_A = h_B$ $h'_B = \max\{h_A, h_C\} + i$ $h'_C = h_D$	AC
L R R			$h'_A = \max\{h_A, h_B\} + i$ $h'_B = h_C$ $h'_C = h_D$	AB

Table 5.2 Cases for Lemmas 5.6 and 5.7. This is an expanded version of Table 5.1, describing how the heights of the subtrees A–D change. Mirror images (RRR, RRL, RLR, RLL) have been omitted. The subtrees in the second column that contain two subtrees and one node have been balanced by a call to BALANCE-NODE. The left/right direction of subtree D is immaterial: D could just as well be the right child of its parent. AB, BC and AC are symbolic names for the cases; note that the two cases named AB in the figure result in the same changes in subtree heights.

In the following, h_A, h_B, h_C, h_D, d_1, d_2, d_3, f_A, f_B and g will denote the values before an iteration, and h'_A, h'_B, h'_C, h'_D, d'_1, d'_2, d'_3, f'_A, f'_B and g' after the one iteration has been executed. In these terms, the definition of g can be written $(d_3 > 0 \Leftrightarrow g')$.

We begin with a couple of facts about d_3 and g. All of the cases in Table 5.2 modify only the first two subtrees A and B – the rest are always unmodified subtrees of the AVL tree as it was before the bulk insertion. Since $h'_C = h_D$ in all cases, Lemma 5.1 implies:

Lemma 5.4 *The value d_3 is in the range $0 \leq d_3 \leq 3$ before every iteration of lines 1–4 of Algorithm 5.3.*

Moreover, by Lemma 5.1, $h_{i+2} \geq h_i + 1$, so d_3 cannot be 0 on two consecutive iterations. This gives us the following relation for g and g'.

Lemma 5.5 *The following implications are true: $\neg g \Rightarrow g'$ and $\neg g' \Rightarrow g$.*

Careful examination of Tables 5.1 and 5.2 reveals that case AB cannot appear after any instance of case BC or AC, because of the left/right directions possible in the eight cases. Case AB is entered only when the left/right directions above the update tree S and its parent are the same: either S is the left child of its parent, which is also the left child of the grandparent of S (cases LLL and RLL of Table 5.1), or both are right children (cases RRR and LRR). However, cases BC and AC (i.e., LLR, RRL, LRL, RLR) all result in situations where the left/right directions above S and its parent differ. For instance, in the result of case LRL in Table 5.1, S is the right child of its parent y, but y is the left child of its parent x. Therefore, lines 1–4 of Algorithm 5.3 actually execute zero or more instances of case AB first, and some instances of cases BC and AC afterwards.

We will first see that a relatively simple invariant holds before and after case AB.

Lemma 5.6 *When lines 1–4 of Algorithm 5.3 are executed, the following invariant holds before and after case AB in Table 5.2 is applied, assuming that the great grandparent of the update tree exists:*

$$\begin{gathered}(-2 \leq d_1 \leq 3) \wedge (0 \leq d_2 \leq 3) \\ \wedge\ (d_1 = -2 \wedge d_2 = 0 \Rightarrow \neg f_A \wedge \neg g)\end{gathered}$$

Proof. At the beginning of the algorithm, $0 \leq d_1 \leq 3$ and $0 \leq d_2 \leq 3$, which is sufficient to satisfy the invariant. This follows trivially from Lemma 5.1, as subtrees A–C are unmodified subtrees of the AVL tree.

A short calculation using the heights given in Table 5.2 shows that case AB (i.e., cases LLL, RRR, LRR, RLL of Table 5.1) affects d_1 and d_2 as follows:

$$\begin{aligned} d_1' &= \begin{cases} d_2 - i & \text{if } d_1 \geq 0 \\ d_1 + d_2 - i & \text{if } d_1 < 0 \end{cases} \\ d_2' &= d_3 \end{aligned}$$

For instance, $d_1' = h_B' - h_A' = h_C - \max\{h_A, h_B\} - i = h_C - \max\{h_A, h_A + d_1\} - i = h_C - h_A - \max\{d_1, 0\} - i = d_1 + d_2 - \max\{d_1, 0\} - i$, which gives the above values.

The requirement $0 \leq d_2' \leq 3$ holds trivially, since $0 \leq d_3 \leq 3$ by Lemma 5.4. If $d_1 \geq 0$, then $-1 \leq d_1' \leq 3$, which satisfies the invariant. If $-2 \leq d_1 < 0$, then $-2 \leq d_1' \leq 2$, except in the case where $d_1 = -2$ and $d_2 = 0$ and $i = 1$ (where d_1' would be -3). But this last case is not possible, since $(d_1 = -2 \wedge d_2 = 0) \Rightarrow \neg f_A$, which means that i must be 0 due to Theorem 4.1: since $d_1 = -2$, the subtree A is higher than B (and their common parent is not in balance), and $\neg f_A$ means that the children of the higher subtree have different heights. By Theorem 4.1, the height of the subtree given to BALANCE-NODE will then be decreased $(h_T = h_S - 1)$ and thus $i = 0$.

The invariant finally requires establishing $\neg f_A'$ and $\neg g'$ in the special case where $d_1' = -2 \wedge d_2' = 0$. This occurs only when $d_1 < 0$ and $d_3 = 0$ (which gives the required $\neg g'$ by definition) and $d_2 > 0$ (because $\neg g' \Rightarrow g$ by Lemma 5.5, and d_2 is equal to the d_3 of the previous iteration). Thus, $d_1' = -2$ only when $d_1 = -2 \wedge d_2 = 1 \wedge i = 1$. Theorem 4.1 now implies $\neg f_A'$, since $d_1 = -2$ means that the root of the subtree given to BALANCE-NODE was not in balance and $i = 1$ that the height of the subtree was not decreased by the node balancing operation ($h_T = h_S$ in the terms of Theorem 4.1). □

A somewhat more complicated invariant than the one above is needed to prove that the properties discussed in the start of this section also hold for the part of the algorithm that executes cases BC and AC.

Lemma 5.7 *When lines 1–4 of Algorithm 5.3 are executed, the following invariant holds before and after case BC or AC in Table 5.2 is applied, assuming that the great grandparent of the update tree exists:*

$$\begin{aligned} & 0 \leq d_2 \leq 3 \wedge d_1 \geq -2 \wedge (d_1 = -2 \wedge d_2 = 0 \Rightarrow \neg f_A \wedge \neg g) \\ \vee\ & d_2 = -1 \wedge d_1 \geq 0 \wedge (d_1 = 0 \Rightarrow \neg g) \\ \vee\ & d_2 = -2 \wedge d_1 \geq 2 \wedge \neg f_B \wedge \neg g \end{aligned}$$

Proof. Name the three lines of the invariant conditions 1°, 2° and 3°, respectively:

$$\begin{array}{ll} 1^\circ & 0 \le d_2 \le 3 \ \wedge \ d_1 \ge -2 \ \wedge \ (d_1 = -2 \wedge d_2 = 0 \Rightarrow \neg f_A \wedge \neg g) \\ 2^\circ & d_2 = -1 \ \wedge \ d_1 \ge 0 \ \wedge \ (d_1 = 0 \Rightarrow \neg g) \\ 3^\circ & d_2 = -2 \ \wedge \ d_1 \ge 2 \ \wedge \ \neg f_B \ \wedge \ \neg g \end{array}$$

Note that only one of these needs to hold to satisfy the invariant. Condition 1° holds initially (i.e., before the first application of case BC or AC), since the invariant in Lemma 5.6 implies condition 1°.

First consider whether case BC (i.e., cases LLR and RRL of Table 5.1) preserves the invariant. A short calculation using the heights given in Table 5.2 shows that this case affects d_1 and d_2 as follows:

$$\begin{array}{lll} \text{If } d_2 \ge 0\text{: } d_1' = d_1 + d_2 + i & \text{and} & d_2' = d_3 - i \\ \text{If } d_2 < 0\text{: } d_1' = d_1 + i & \text{and} & d_2' = d_2 + d_3 - i \end{array}$$

If condition 1° holds before case BC is applied, then $d_2 \ge 0$, so $d_1' = d_1 + d_2 + i \ge -2$ and $-1 \le d_2' = d_3 - i \le 3$ (since $0 \le d_3 \le 3$ by Lemma 5.4). If $d_2' \ge 0$, then condition 1° will be satisfied also after case BC is executed. The special situation $d_1' = -2 \wedge d_2' = 0$ is not possible, since $d_1' = -2$ only if $d_1 = -2$ and $d_2 = 0$ and $i = 0$. However, $d_2 = 0$ means that the subtree given to BALANCE-NODE is in balance, and thus BALANCE-NODE will not perform any rotations, which implies that $i = 1$ when $d_2 = 0$ in case BC.

Otherwise, if condition 1° holds before case BC and $d_2' = -1$, then condition 2° will be satisfied: $d_2' = d_3 - i = -1$ only if $i = 1$ and $d_3 = 0$, and the last gives $\neg g'$ by definition. Then $d_1' \ge -1$ (since $i = 1$), but to satisfy condition 2°, we need to show that $d_1' \ge 0$. Assuming $d_1' = -1$ leads to a contradiction: $d_1' = d_1 + d_2 + i = -1$ only if $d_1 = -2$ and $d_2 = 0$, which would together imply $\neg g$ (due to condition 1°). By Lemma 5.5, $\neg g \Rightarrow g'$, but we have derived $\neg g'$ above – thus, d_1' cannot be -1. We have thus established that if condition 1° holds before case BC is applied, either condition 1° or condition 2° will hold afterwards.

If condition 2° holds before case BC is applied, then $i = 1$ since the subtree given to BALANCE-NODE is already in balance ($d_2 = -1$), so BALANCE-NODE will not perform any rotations. Hence, $d_1' = d_1 + 1 \ge 1$ and $-2 \le d_2' = d_3 - 2 \le 1$. If $d_2' \ge -1$, these values clearly satisfy either condition 1° or condition 2°. If $d_2' = -2$, then condition 3° is satisfied as follows. Here $d_2' = -2$ only if $d_3 = 0$, which gives $\neg g'$ by definition. In addition, $d_1' = d_1 + 1 \ge 2$, because d_1 cannot be 0 when

$d_3 = 0$: if $d_1 = 0$, the invariant gives us $\neg g$, but this conflicts with the $\neg g'$ that we just derived, because $\neg g' \Rightarrow g$ (Lemma 5.5). Finally, $\neg f'_B$ holds because the subtree given to BALANCE-NODE was in balance with children of different heights (this subtree will be subtree B on the next iteration, as shown in Table 5.2).

If condition 3° holds before case BC is applied, then $i = 0$ due to Theorem 4.1 ($d_2 = -2$, so the subtree given to BALANCE-NODE is not in balance, and $\neg f_B$). Then $d'_1 = d_1 \geq 2$ and $d'_2 = d_3 - 2$. The $\neg g$ of condition 3° implies by $\neg g \Rightarrow g'$ (Lemma 5.5) that $d_3 > 0$. Thus, $-1 \leq d'_2 = d_3 - 2 \leq 1$. These values satisfy either condition 1° or condition 2°.

Case BC thus preserves the invariant. Next consider case AC (i.e., cases LRL and RLR of Table 5.1). This affects d_1 and d_2 as follows (again calculated from the heights given in Table 5.2 – note that $d_1 + d_2 = h_C - h_A$):

If $d_1 + d_2 \geq 0$: $d'_1 = d_2 + i$ and $d'_2 = d_3 - i$
If $d_1 + d_2 < 0$: $d'_1 = i - d_1$ and $d'_2 = d_1 + d_2 + d_3 - i$

If condition 1° holds before case AC is applied, we need to consider three subcases depending on $d_1 + d_2$. If $d_1 + d_2 \geq 0$, then $0 \leq d'_1 = d_2 + i \leq 4$ and $-1 \leq d'_2 = d_3 - i \leq 3$. These values clearly satisfy either condition 1° or condition 2°, since the combination $d'_1 = 0 \wedge d'_2 = -1$ is not possible using either value of i.

If $d_1 + d_2 = -1$, then $i = 1$ and $\neg f'_B$, since the subtree given to BALANCE-NODE is already in balance, with children of different heights. (The subtree resulting from BALANCE-NODE will be subtree B on the next iteration, as shown in Table 5.2.) Thus, $2 \leq d'_1 = 1 - d_1 \leq 3$ (since $d_1 < 0$) and $-2 \leq d'_2 = d_3 - 2 \leq 1$. When $d'_2 \geq -1$, condition 1° or condition 2° is clearly satisfied. When $d'_2 = -2$, condition 3° is satisfied, since d_3 must be 0, which gives $\neg g'$.

If $d_1 + d_2 = -2$, then $d_1 = -2$ and $d_2 = 0$, and the invariant gives us $\neg f_A$ and $\neg g$. Then $i = 0$ by Theorem 4.1. Thus, $d'_1 = 2$, and, because $d_3 > 0$ from $\neg g \Rightarrow g'$ (Lemma 5.5), the condition $-1 \leq d'_2 = d_3 - 2 \leq 1$ holds. This clearly satisfies either condition 1° or condition 2°.

If condition 2° holds before case AC is applied, and $d_1 = 0$, then $d_1 + d_2 = -1$. Thus, the subtree given to BALANCE-NODE is already in balance, so $i = 1$. Then $d'_1 = 1$ and $-1 \leq d'_2 = d_3 - 2 \leq 1$ (here $d_3 > 0$, because $d_1 = 0$ implies $\neg g$ due to the invariant, and $\neg g \Rightarrow g'$). These values clearly satisfy either condition 1° or condition 2°.

If condition 2° holds before case AC and $d_1 > 0$, then $d_1 + d_2 \geq 0$ and there are two subcases depending on i. If $i = 0$, then $d'_1 = -1$ and

$0 \le d_2' = d_3 \le 3$, which satisfies condition 1°. If $i = 1$, then $d_1' = 0$ and $-1 \le d_2' = d_3 - 1 \le 2$. Condition 2° is satisfied if $d_2' = -1$, because $d_3 = 0 \Leftrightarrow \neg g'$. Otherwise, if $0 \le d_2' \le 2$, condition 1° is satisfied.

Finally, if condition 3° holds before case AC is applied, then $d_1 + d_2 \ge 0$, so $-2 \le d_1' = i - 2 \le -1$ and $0 \le d_2' = d_3 - i \le 3$ ($d_3 > 0$ due to $\neg g \Rightarrow g'$). Condition 1° will clearly be satisfied, since the combination $d_1' = -2 \wedge d_2' = 0$ is not possible using either value of i.

We have now established that the invariant in the theorem holds initially and that the cases BC and AC preserve it. □

The above invariants can now be used to conclude that the properties described in the start of this section hold:

Lemma 5.8 *The following properties hold when Algorithm 5.3 is begun and after each iteration of lines 1–4 of the algorithm, assuming that the great grandparent of the update tree exists:* $-2 \le d_2 \le 3$ *and* $d_1 \ge -2$ *and* $d_1 < 0 \Rightarrow |d_1 + d_2| \le 2$.

Proof. All three properties follow directly from both of the invariants in Lemmas 5.6 and 5.7. □

5.3.3 Complexity: Number of rotations

We are now ready to examine the number of rotations performed by the rebalancing algorithm.

Theorem 5.2 *After an update tree with* m *keys and height* $h_S = \lfloor \log_2 m \rfloor$ *has been inserted in an* AVL *tree, Algorithm 5.3 uses at most* $7h_S + 92 = O(h_S) = O(\log m)$ *rotations in the worst case to rebalance the tree.*

Proof. First consider the case where no iterations are done in the first phase (lines 1–4) of the algorithm, i.e., the ending condition $h_D > h_S - 1$ holds initially. Because the update tree is inserted at a leaf, $h_D \le 6$ in an internal tree or $h_D \le 7$ in an external tree.* Since the ending condition holds, $h_S \le 6$ in an internal tree and $h_S \le 7$ in an external tree. The second phase of the logarithmic rebalancing algorithm (Algorithm 5.3) will do the same rotations as the log-squared

* The maximum h_D occurs when the update tree is inserted in place of the subtree marked with h_i in Figure 5.2. Then $h_i = 0$ in an external tree, since the update tree is inserted in place of a single leaf. In an internal tree, $h_i = -1$, because the position where the update tree is placed contains no nodes, and its height must be defined to be -1 if the height of a single node is 0.

algorithm (Algorithm 5.2), so Lemma 5.2 (with $d = 7$ in both kinds of trees) can be used to to conclude that at most $\lfloor 7^2 + 7 \cdot 7/2 + 4 \rfloor = 77$ rotations are done.

Then assume that the first phase executes at least one iteration. We will see that each iteration requires at most 7 rotations plus some rotations taken from a "potential" Φ in which previous iterations can save their leftover rotations.*

Lemma 5.8 claims that, before any iteration, $d_1 \geq -2$ and $-2 \leq d_2 \leq 3$. Each iteration performs one rotation to move the update tree upward and a number of rotations in the node balancing operation (see Table 5.1). By Theorem 4.2, the balancing requires at most $d - 1$ rotations, where d is the height difference of the subtrees to be balanced. Here, $d = |d_1|$ for case AB in Table 5.2, $d = |d_2|$ for case BC, and $d = |d_1 + d_2|$ for case AC.

Since d_1 does not have a constant upper limit, we will use the potential Φ to save the $d_1 - 1$ rotations necessary for cases AB and AC. The potential will always be at least $\max\{d_1 - 1, 0\}$ (if $-2 \leq d_1 \leq 1$, the potential is unnecessary). Initially (at the beginning of the first iteration), $d_1 \leq 3$, and we will save two rotations in the potential: $\Phi = 2$.

If an iteration executes case AB from Table 5.2, and $d_1 \geq 0$, the $d_1 - 1$ rotations necessary for balancing are taken from the potential, possibly reducing Φ to 0. After the iteration, $-2 \leq d'_1 \leq 3$ because of the invariant of Lemma 5.6. The potential may thus need to be increased by at most 2. One rotation is used to move the tree upward, so a total of at most 3 rotations in addition to those from the potential are used by the iteration. This is less than the 7 required for the result of the theorem.

If the iteration executes case AB but $d_1 < 0$, then $-2 \leq d_1 \leq -1$, and at most one rotation is used in the balancing. After the iteration, $d'_1 \leq 3$ because of the invariant of Lemma 5.6, and at most 2 rotations need to be saved in the potential. Thus, this case uses a total of at most 4 rotations, including the one for raising the tree upward.

An iteration that executes case BC needs at most 2 rotations for the balancing, since the invariant of Lemma 5.7 gives $-2 \leq d_2 \leq 3$. The proof of Lemma 5.7 details that either $d'_1 = d_1 + d_2 + i$ or $d'_1 = d_1 + i$. The resulting d'_1 may thus be 4 larger than d_1 (when $d'_1 = d_1 + d_2 + i$ and $d_2 = 3$ and $i = 1$), and 4 rotations need to be saved in the potential in the worst case. Thus, at most 7 rotations are used.

* This proof technique resembles the potential method for amortized analysis [81].

An iteration that executes case AC needs at most $|d_1 + d_2| - 1 \leq |d_1| + |d_2| - 1$ rotations for the balancing. If $d_1 \geq 0$, the $d_1 - 1$ rotations can be taken from the potential, and since $-2 \leq d_2 \leq 3$, at most 3 additional rotations are needed. If $d_1 < 0$, then $|d_1 + d_2| \leq 2$ (Lemma 5.8) and at most one rotation is needed. After the iteration, $d'_1 \leq 4$ (since $d'_1 = i - d_1 \leq 3$ or $d'_1 = d_2 + i \leq 4$, as described in the proof of Lemma 5.7), and at most 3 rotations may need to be saved to the potential. At most $3 + 1$ other rotations were needed, so at most 7 rotations are used in total.

There are at most h_S iterations in the first phase, where h_S is the height of the update tree (Lemma 5.3). Since each iteration uses at most 7 rotations (when the potential is taken into account), and 2 rotations are saved to the initial potential, a total of at most $7h_S + 2$ rotations are performed by lines 1–4 of Algorithm 5.3.

The first phase of the algorithm finishes when $h_D > h_S - 1$. Because h_D can increase by at most 3 in one iteration (Lemma 5.1), $h_S - 3 \leq h_C \leq h_S - 1$ or otherwise we would have finished on the previous iteration. Lemma 5.8 implies that $h_C - 3 \leq h_B \leq h_C + 2$ and $h_A \leq h_C + 2$; thus, $h_S - 6 \leq h_B \leq h_S + 1$ and $h_A \leq h_S + 1$.

The first iteration of the second phase merges the update tree S with subtree A (the current sibling of S). According to Theorems 4.1 and 4.2, this uses at most $|h_S - h_A| - 1$ rotations and results in a tree SA with height $h_{SA} = \max\{h_S, h_A\} + i_{SA}$, where $0 \leq i_{SA} \leq 1$ (and $i_{SA} = 1$ if the height was not decreased in the node balancing operation). Thus, $h_A \leq h_S + 1$ implies that $h_S \leq h_{SA} \leq h_S + 2$. Only six rotations are needed in addition to some that can be taken from the remaining potential (which is $d_1 - 1$, if $d_1 > 0$), which can be seen as follows. If $h_A \leq h_S$, then $h_S - h_A - 1$ rotations are needed. If $d_1 > 0$, then $d_1 - 1 = h_B - h_A - 1 \geq h_S - 6 - h_A - 1$, and at most 6 rotations in addition to those from the potential are needed. If $-2 \leq d_1 \leq 0$, the potential will not help, but then $h_S - 6 \leq h_A \leq h_S + 3$ (because here $h_B \leq h_A \leq h_B + 2$) and only 5 or fewer rotations are needed. Finally, if $h_A > h_S$, then $h_A = h_S + 1$ and no rotations are needed, since the subtree SA is already in balance.

The second iteration merges tree SA with its sibling, subtree B, and here the height difference is at most 8 (the maximum occurs when $h_{SA} = h_S + 2$ and $h_B = h_S - 6$); thus, at most 7 rotations are needed. The resulting subtree SAB has height $h_S \leq h_{SAB} = \max\{h_{SA}, h_B\} + i_{SAB} \leq h_S + 3$, where $0 \leq i_{SAB} \leq 1$.

Since subtree C (the sibling of SAB) has not been modified by this algorithm, Lemma 5.2 tells us how many rotations the remaining

iterations use. The original sibling of the root of subtree C had height at least $h_C - 1$. The current subtree SAB has height at most $h_C + 6$ (the maximum occurs when $h_C = h_S - 3$ and $h_{SAB} = h_S + 3$), so the d of Lemma 5.2 is at most 7, and the remaining iterations will perform at most 77 rotations. The lemma additionally requires that $d > 0$, i.e., h_{SAB} is at least as large as the height of the original subtree at this position. This is true, since the height of the original subtree is at most $h_C + 1 \leq h_S$ (because it was a sibling of C), and $h_{SAB} \geq h_S$.

In summary, the first phase (lines 1–4) of the algorithm uses at most $7h_S + 2$ rotations, and the second phase uses at most $6+7+77 = 90$ rotations (in addition to those taken from any leftover potential). Thus, the whole algorithm performs at most $7h_S + 92 = O(h_S)$ rotations. If no iterations are done in the first phase, at most 77 rotations are done. □

The above proof reveals a subtle restriction for the ending condition of the first phase of the rebalancing algorithm. Intuitively, it would seem that the only requirement for the ending condition is that, at the end of the first phase, the height of the update tree S differs by at most a constant from the height of an unmodified subtree somewhere at most a constant distance away from the update tree.

However, it can be seen from the above proof that $h_{SAB} \geq h_C + 1$ is an essential, more restrictive requirement for the ending condition of the first phase. Since C is an unmodified subtree, the original height of its parent was either $h_C + 1$ or $h_C + 2$. Now, $h_{SAB} = h_C + 1$ is the smallest value for h_{SAB} which guarantees that the subtree resulting from the third iteration of the second phase has height $\geq h_C + 2$, which means that that the bulk insertion does not decrease the height of the subtree at this location. This is important since if the height were decreased, Lemma 5.2 would not apply, and the situation would be analogous to single deletion, where $O(\log n)$ additional rotations are needed in the worst case (with n nodes in the whole tree). A height decrease after a bulk insertion is counterintuitive, but possible: for example, if $h_{SAB} < h_C - 1$, and the children of the root of C have different heights, Theorem 4.1 states that the height must decrease. However, this cannot happen with the ending condition chosen in Algorithm 5.3.

5.3.4 Height value changes

To find out the total rebalancing complexity of the bulk-insertion algorithms, we also need to examine the changes that the rebalancing

algorithms make to the height values of the nodes in the AVL tree. We will begin with an upper bound for the number of height values that change in one single-bulk insertion.

In the discussion below, a *strictly balanced node* is a node whose children have equal heights.

Lemma 5.9 *The log-squared and logarithmic rebalancing algorithms change $O(r) + b$ height values, where r is the number of rotations performed by the rebalancing algorithm, and $b \leq B$, where B is the number of strictly balanced nodes in the search path (i.e., the path from the root of the whole tree to the parent of the root of the update tree). Moreover, the b height value changes are done on strictly balanced nodes which will not be strictly balanced after rebalancing.*

Proof. The height value changes done by the rebalancing algorithms can be classified into four categories: (1) A rotation may change the height value of the nodes that it modifies. (2) The height value of a node that was modified by a rotation can change a second time later (but only once). (3) The height value of a strictly balanced node on the search path may need to be changed, but the node will not be strictly balanced after this change. (4) The height value of at most one additional node on the search path may change.

To classify the height value changes, we need to analyze the individual parts of the rebalancing algorithms. We assume that rotations update the height values of the nodes that they modify – these changes are in category (1).

In addition to rotations, the node balancing algorithm updates the height value of the node n that it gets as an argument, and of each node considered in the upward phase of the algorithm. Any node considered in the upward phase has taken part in a rotation of the downward phase, so these changes fall in category (2). The height value change of node n is classified below.

Each iteration of the first phase of the logarithmic rebalancing algorithm executes a rotation to move one node off the search path. All height value changes done here are in category (1). In addition, the node balancing algorithm may update the height value of the node that was moved off the path (i.e., the node n given to node balancing algorithm), but this node was just involved in a rotation, so this change falls in category (2).

Each iteration of the log-squared rebalancing algorithm, and of the second phase of the logarithmic algorithm, executes the node balancing algorithm on a node n on the path from the position leaf to the root.

If n is not initially in balance, a rotation is done at n, and the additional height value change done by the node balancing algorithm is in category (2).

However, if the node n is already in balance, then its height value may need to be changed outside of a rotation. In other words, the problematic case occurs when the log-squared algorithm, or the second phase of the logarithmic algorithm, reaches a node which is still in balance after a height increase in one of its children.

There are two ways in which the height can change. If the node n is originally strictly balanced and the bulk insertion increases the height of either child by one, then the height of the resulting node (which is still in balance, but now not strictly balanced) will increase by one. This is category (3).

The other situation where the height can change is when the children of node n originally have a height difference of one, and the bulk insertion increases the height of the smaller child by two. After this, node n will be in balance and the height of n will be one greater. But this situation can occur only once per single-bulk insertion, since the resulting height increase is only one, which means that on later iterations, the height of a subtree cannot increase by two. This one change is classified into category (4).

Since a rotation modifies $O(1)$ nodes, categories (1) and (2) contain a total of $O(r)$ height value changes. Category (3) contains exactly b changes, and category (4) includes only one change. Thus, a total of $O(r) + b$ height values are changed. □

We are now ready to analyze the amortized rebalancing complexity of the algorithms for single-bulk insertion. It was noted earlier that when only one key is inserted using single-bulk insertion, the rebalancing operations performed are the same as for single insertion. The following theorem thus also applies when some of the bulk insertions are replaced by single insertions.

Theorem 5.3 *Assume that k single-bulk insertions with $m_1, \ldots, m_k$ keys are performed to an initially empty* AVL *tree. If the logarithmic rebalancing algorithm is used, the time complexity of rebalancing after the ith insertion is $O(\log n_i + \log m_i)$ in the worst case, and $O(\log \max\{m_1, \ldots, m_k\})$ when amortized over the k insertions, where $n_i = m_1 + \cdots + m_{i-1}$ is the number of nodes in the tree before the ith insertion. If the log-squared algorithm is used instead, the bounds are $O(\log n_i + \log^2 m_i)$ (worst case) and $O(\log \max\{m_1, \ldots, m_k\} + \log^2 m_i)$ (amortized).*

Proof. Consider the number of height value changes performed by the algorithms. Neither rebalancing algorithm does more than $O(1)$ work without either performing a rotation or changing a height value afterwards, so the amortized time complexity of rebalancing (i.e., of Algorithms 5.2 and 5.3) is the same as the amortized number of height value changes.

Theorems 5.1 and 5.2 give the number of rotations that are done in the worst case. This gives the r of Lemma 5.9.

In the worst case, the B of Lemma 5.9 is $O(\log n_i)$, i.e., the length of the search path, and the worst-case figures then follow directly from Lemma 5.9 and Theorems 5.1 and 5.2.

The following potential function is used in the amortized analysis: $\Phi(p) = 1$, if node p is strictly balanced and a bulk insertion (either the actual insertion or a rotation done in rebalancing) has been performed anywhere in the subtree rooted at the node. Otherwise, $\Phi(p) = 0$. In other words, $\Phi(p) = 1$ in all strictly balanced nodes, except for those that are part of an update tree which has not been touched by a subsequent bulk insertion.

On the ith insertion, all strictly balanced nodes on the path from the root of the tree to the position leaf have either $\Phi(p) = 1$ or are part of a single update tree from a previous insertion $j < i$. The strictly balanced nodes with $\Phi(p) = 0$ cannot be part of more than one update tree, because then the lower update tree would be the result of a bulk insertion done in the subtree rooted at the upper update tree.

Thus, the only strictly balanced nodes with $\Phi(p) = 0$ are part of one update tree from a previous insertion $j < i$. On insertion i, there are then at most $O(\log m_j) = O(\log \max\{m_1, \dots, m_{i-1}\})$ such nodes.

Lemma 5.9 states that some strictly balanced nodes may need their height values changed outside of any rotation, but the nodes will not be strictly balanced after the height value update. For any node with $\Phi(p) = 1$, the work needed for the height value update can be taken from the potential, since the resulting $\Phi(p) = 0$. The new nodes in the update tree of this bulk insertion will have $\Phi(p) = 0$, since no bulk insertions have yet been performed below them. As the result of a rotation, some non-strictly balanced nodes may become strictly balanced, but then the potential increase $\Phi(p) = 0 \to 1$ can be included in the $O(1)$ work performed by the rotation.

Therefore, on the ith insertion, the only nodes not accounted for by the potential are the $O(\log \max\{m_1, \dots, m_{i-1}\})$ nodes in an update tree from a previous insertion. Then the B of Lemma 5.9 is $O(\log \max\{m_1, \dots, m_k\})$ in the amortized case. The amortized figures now follow directly from Theorems 5.1 and 5.2. □

BULK-INSERT(A):
1 Sort the array A of keys to be inserted.
2 **while** there are keys left **do**
3 Search in the tree for the smallest key s to be inserted. Save the search path P and the key m of the lowest node where the search descended to the left child.
4 Form an update tree S from the keys k with $s \leq k < m$, and insert it into the position leaf found in the search.
5 REBALANCE(S, P)

Algorithm 5.4 Simple approach for bulk insertion with multiple bulks. A is an array of keys to be inserted.

5.4 Inserting multiple bulks

As noted in Section 5.1, the previous sections assumed that the keys to be inserted go to the same location ("position leaf"); in other words, that the tree has no keys whose values are between the smallest and largest keys to be inserted.

A simple way to remove this restriction is to process each position leaf separately. Algorithm 5.4 gives the full bulk-insertion algorithm using this simple strategy. Since the algorithm executes consecutive single-bulk insertions, the results of Theorem 5.3 apply to it.

Disadvantages of this simple approach are that all of the new keys are not available in the tree before any rebalancing is done (which could be important in a concurrent application), and that the algorithm may do unnecessary rebalancing when several update trees are close to each other.

The total time complexity of the simple approach can be stated as follows.

Theorem 5.4 *Assume that I bulk insertions that form a total of $K \geq I$ bulks with $m_1, \ldots, m_K$ keys are performed to an initially empty* AVL *tree using the logarithmic rebalancing algorithm and the simple approach for multiple bulks. If the tree has n_i keys before the ith bulk insertion, and the ith bulk insertion forms k_i bulks with $o_1, \ldots, o_{k_i}$ keys, then the ith bulk insertion uses $O(k_i \log(n_i + \sum_j o_j))$ time for tree traversal, $O(k_i \log \max\{m_1, \ldots, m_K\})$ amortized time for rebalancing (amortized over all I bulk insertions), and $O(\sum_j o_j)$ time for creating the new nodes.*

Proof. For each of the k_i bulks in the ith bulk insertion, the simple approach for multiple bulks traverses a root-to-leaf path of length

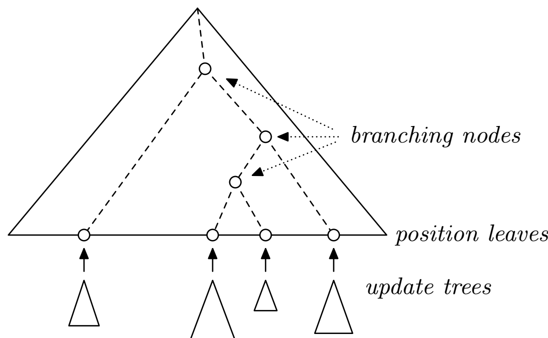


Figure 5.6 A position tree for bulk insertion with multiple bulks. The dashed lines represent chains of nodes that form the position tree together with the nodes drawn in the figure.

$O\big(\log\big(n_i + \sum_{j=1}^{k_i} o_j\big)\big)$; more precisely, the length of the jth path is $O(\log(n_i + (o_1 + o_2 + \cdots + o_{j-1})))$.

For the rebalancing complexity, we need to ignore the partitioning of the K bulks into I bulk insertions with $k_1, \ldots, k_I$ bulks, $\sum_j k_j = K$, because the amortization in Theorem 5.3 is over all single-bulk insertions, not just the ones executed by the ith bulk insertion. Thus, in terms of rebalancing, the ith bulk insertion performs k_i single-bulk insertions, and Theorem 5.3 gives the stated rebalancing complexity.

There are $o_1 + o_2 + \cdots + o_{k_i} = \sum_j o_j$ new nodes. □

The traversal time $O(k_i \log(n_i + \sum_j o_j))$ is, of course, close to $O(k_i \log n)$ if the number of inserted keys is much smaller than the number of keys already present ($\sum_j o_j \ll n_i$).

A more advanced method that extends the log-squared rebalancing algorithm is given in [55]. The nodes visited by searching for all of the position leaves form a tree-shaped subgraph called the *position tree*, which consists of all ancestors of the position leaves (see Figure 5.6). The basic idea of this method is to do rebalancing by traversing the position tree in a bottom-up manner (e.g., in post-order), applying the node balancing algorithm at each node. The node balancing algorithm can also be used at a node where the position tree branches (called a *branching node*), as long as the children are processed first: the approach used by the log-squared rebalancing algorithm works even if update trees have been inserted under both children of an unbalanced node.

However, the approach of [55] does not directly work with the logarithmic rebalancing algorithm. The upward phase of Algorithm 5.3

requires that the siblings it considers are unmodified subtrees of the original tree, which is not true at the branching nodes. The solution is to stop the upward phase before each branching node. In other words, the logarithmic algorithm is used to rebalance the path from an update tree to the nearest ancestor that is a branching node. When both children of a branching node have been rebalanced using the logarithmic algorithm, the node balancing algorithm is used to rebalance the branching node itself. After this, the subtree resulting from rebalancing the branching node is treated as a new update tree, and the logarithmic algorithm is used to rebalance the path from its parent to the next branching node (or the root of the tree, if no branching nodes remain).

Algorithm 5.5 gives the method in detail. The algorithm saves the search path in an array P, and uses two other arrays for additional book-keeping: $m[i]$ is a maximum limit for the keys that can be placed below node $P[i]$ in the tree (i.e., the key of the lowest node in $P[1..i-1]$ in which the search descended to the left child), and $b[i] = 1$ if the left child of the branching node at $P[i]$ has already been processed.

The advanced approach traverses each node in the position tree only a constant number of times: each ancestor a is traversed once when searching downwards in the tree, and the saved path entry associated with a may be traversed at most a constant number of times when searching for subsequent position leaves.

The number of nodes in the position tree is analyzed in [76]:

Lemma 5.10 *Let $l_1, \ldots, l_k$ be some leaf nodes, from left to right, in an* AVL *tree T with n nodes, and let d_i denote the distance between the leaves l_i and l_{i+1} (specifically, the number of nodes with keys between the keys of l_i and l_{i+1}). The total number of ancestors of $l_1, \ldots, l_k$ is*

$$O\left(\log n + \sum_{i=1}^{k-1} \log d_i\right).$$

Proof. The result is essentially given in Theorem 2 of [76]. There the distance d'_i is defined as the number of leaves l such that $\text{key}(l_i) \leq \text{key}(l) \leq \text{key}(l_{i+1})$. Here the distance d_i is counted in the number of intervening nodes (both leaves and internal nodes), which is more appropriate in an internal tree. The result is asymptotically the same, since $d_i = \Theta(d'_i)$ in an AVL tree. □

The total complexity of the advanced approach can be stated as follows, using the above lemma for analyzing the tree traversal.

BULK-INSERT(A):

1 Sort the array A of keys to be inserted.
2 **while** there are keys left **do**
3 Search in the tree for the smallest key s to be inserted, starting from the lowest node in the saved search path P, or the root if P is empty. Save the full search path in $P[1..n]$, where $P[1]$ is the root and $P[n]$ is the node where the search finished. For each new node $P[i]$ added to P, set $b[i] \leftarrow 0$ and $m[i] \leftarrow m[i-1]$ (if $P[i]$ is the right child of $P[i-1]$) or $m[i] \leftarrow$ key of $P[i-1]$ (otherwise); $m[0] = \infty$.
4 Form an update tree S from the keys k with $s \leq k < m[n]$, and insert it into the position leaf found in the search. Update s.
5 $P[n+1] \leftarrow$ the root of the update tree
6 $u \leftarrow 1$
7 **while** $u = 1$ **do**
8   Search upward in P for the branching node, i.e. the lowest node $P[i]$ such that $b[i] = 1$ or $s < m[i]$.
9   **if** no branching node was found **then**
10   REBALANCE($P[n+1]$, $P[1..n]$)
11   **return**
12   **else** {*the branching node is* $P[i]$}
13   REBALANCE($P[n+1]$, $P[i+1..n]$)
14   Remove $P[i+1..n+1]$ from P.
15   **if** $b[i] = 0$ **then** {*left child of* $P[i]$ *done*}
16     $P[i+1] \leftarrow$ the right child of $P[i]$
17     $b[i] \leftarrow 1$
18     $u \leftarrow 0$
19   **else** {*both children of* $P[i]$ *done*}
20   BALANCE-NODE($P[i]$)
21   Remove $P[i]$ from P.

Algorithm 5.5 Advanced approach for bulk insertion with multiple bulks, using the logarithmic rebalancing algorithm. A is an array of keys to be inserted. This code is for internal trees; for external trees, add "$n \leftarrow n - 1$" after line 4.

Theorem 5.5 *Assume that I bulk insertions that form a total of $K \geq I$ bulks with $m_1, \ldots, m_K$ keys are performed to an initially empty* AVL *tree using the logarithmic rebalancing algorithm and the advanced approach for multiple bulks. If the tree has n_i keys before the ith bulk insertion, and the ith bulk insertion forms k_i bulks with $o_1, \ldots, o_{k_i}$ keys, then the ith bulk insertion uses $O\big(\log n_i + \sum_{j=1}^{k_i-1} \log d_j\big)$ time for tree traversal, $O(k_i \log \max\{m_1, \ldots, m_K\})$ amortized time for rebalancing (amortized over all I bulk insertions), and $O(\sum_j o_j)$ time for creating the new nodes. The value d_j is defined as the number of nodes in T between the jth and $(j+1)$th update tree.*

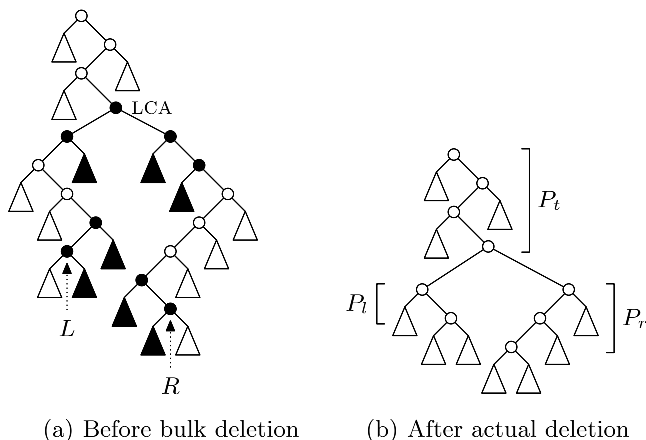
The traversal time of the advanced approach is never larger than that of the simple approach and is smaller when the update trees are close to each other. Consider, for instance, the path P from the root of the tree to the lowest common ancestor of all the position leaves of the ith bulk insertion. The simple approach traverses P (or a similar path, if a rebalancing operation changes it) k_i times, while the advanced approach traverses P only once (except possibly when rebalancing).

In a concurrent implementation, it can be useful to make newly inserted keys available to concurrent readers as soon as possible, before rebalancing has been completed. It is easy to modify Algorithm 5.5 to work in two phases: (1) Create and insert all of the update trees, but do no rebalancing (skip lines 10, 13 and 20 of the algorithm). (2) Do the rebalancing (run the algorithm again, but skip line 4 and modify line 3 to stop the search at the roots of the previously created update trees). The position tree can also be saved in phase (1), so that the second phase does not need to redo the search. However, these modifications have a disadvantage: the locality of the algorithm is not as good, since the position tree is traversed twice and, for example, the first rotations are done close to the leftmost update tree just after the rightmost update tree has been inserted.

5.5 Single-bulk deletion

As explained in Section 2.8, a bulk-deletion (or interval-deletion) algorithm takes as input a set of intervals $[L, R]$ of keys to delete from the search tree. The algorithm deletes all keys in the given intervals and rebalances the tree.

An outline of the bulk-deletion algorithm of this section was given in [77], but without many of the details that are present here, for instance with regard to deleting multiple intervals.



(a) Before bulk deletion (b) After actual deletion

Figure 5.7 Overview of bulk deleting an interval $[L, R]$. The black subtrees will be detached and the black nodes deleted.

5.5.1 Deleting an interval

Single-bulk deletion processes each interval separately, reducing the problem to that of deleting a single interval $[L, R]$ from the tree. A single interval is deleted as follows (see Algorithm 5.6). First, locate the *lowest common ancestor* (LCA) of the interval – this is the highest node in the tree that will be deleted (see Figure 5.7(a)). The LCA is located at the point where the search paths for L and R diverge to different children, i.e., the highest node whose key is in between L and R. Denote the key in the LCA node by A.

Deletion proceeds by searching for L starting from the left child of the LCA node. Whenever this search descends to a left child from a node n, the subtree rooted at the right child of n needs to be completely deleted: its keys are larger than L and smaller than A. Node n also needs to be deleted (for the same reason), so n is simply replaced with its left child. Freeing up the memory associated with the deleted nodes is discussed in Section 5.5.2 below.

The search for L ends when it reaches a leaf or, in an internal tree, when a node with key L is reached. In the latter case, the node can be replaced by its left child. When the search ends, the interval $[L, A]$ has been deleted. The algorithm then starts over from the LCA node, now searching for R in the same way to delete $[A, R]$.

In an external tree, actual deletion is now done and only rebalancing remains. However, in an internal tree, the LCA node itself still needs to

BULK-DELETE(L, R):

1 Search for LCA: search in the tree for L and R until the search diverges or either L or R is found.
2 $a \leftarrow$ the LCA node
3 $P_t \leftarrow$ path $[r, \ldots, a]$ from root r to LCA
4 $P_l \leftarrow$ empty path
5 $P_r \leftarrow$ empty path
6 **if** a.key $< L$ or a.key $> R$ **then**
7   **return**   {*There is nothing to delete*}
8 **if** $L \neq a$.key **then**   {*Delete left half*}
9   Search for L starting from a.left.
10   **if** the search goes left from a node n or n.key $= L$ **then**
11   Mark node n and the subtree n.right as detached from the tree.
12   Replace n with its left child.
13   **if** n.key $= L$ **then**
14   Stop the search here.
15     $P_l \leftarrow$ the search path
16     $d_l \leftarrow$ the parent of the highest deleted node
17 **if** $R \neq a$.key **then**   {*Delete right half*}
18   As above, but search for R and detach when going right.
19     $P_r \leftarrow$ the search path
20     $d_r \leftarrow$ the parent of the highest deleted node
21 **if** either child of a is now empty **then**
22   Replace a with the other child.
23   Remove a from P_t.
24 **else**   {*Need to delete* LCA *by replacing it with its successor*}
25   Search for the successor s of a starting from the lowest node in P_r.
26   Delete s as in single deletion, and extend P_r to end at the parent of s.
27 REBALANCE(P_l, d_l)
28 REBALANCE(P_r, d_r)
29 REBALANCE$(P_t$, lowest node in $P_t)$

Algorithm 5.6 Actual bulk deletion of an interval $[L, R]$ in an internal tree.

REBALANCE(P, c):
1 **for** each node n in P from down to up **do**
2 Update the height value of n.
3 **if** node c has been reached and n is in balance and the height value did not change **then**
4 **return**
5 BALANCE-NODE(n)

Algorithm 5.7 Rebalancing for bulk deletion. The argument P is part of a search path (see Algorithm 5.6), and $c \in P$ is a node above which rebalancing can be stopped if nothing changes in the tree.

be deleted. If either child of the LCA was deleted completely, the LCA node can be replaced with its remaining child. But if nodes remain in both children of the LCA, it needs to be replaced by its successor node, i.e., the node s with the next-larger key. This is done in the same way as in single deletion (see Section 2.4): move the key and associated data fields of s to the LCA node and replace s by its right child (if any). As in single deletion, the predecessor node, found under the left child of the LCA, could just as well be used.

Rebalancing needs to be done on the following paths (see Figure 5.7(b)): (1) the path down from the LCA to the position where the search for L ended (call this P_l); (2) the corresponding path for R (P_r); and (3) the path up from the LCA to the root of the tree (P_t). In an internal tree, if a successor node s is located and deleted as described above, the path P_r should be extended to include the parent of s.

The rebalancing algorithm (Algorithm 5.7) is simple: simply use the node balancing algorithm bottom-up for P_l, P_r and P_t. Above the highest point where subtrees were detached, rebalancing can stop early if a node n is already in balance and its height did not change – then nothing has been done on the path up from n to the beginning of the path P_l, P_r or P_t.

The rebalancing algorithm is essentially the same as the log-squared algorithm for bulk insertion (Algorithm 5.2), but, as we will see below, its complexity is logarithmic when used for deletion instead of insertion.

5.5.2 Detaching subtrees

A major advantage of bulk deletion is that it is not necessary to visit each individual node that is to be deleted, since actual deletion can often detach entire subtrees from the tree. However, the allocation of

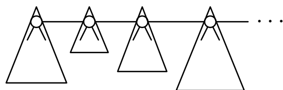


Figure 5.8 Storing detached subtrees. Each of the roots includes pointers to its children and to the next node in the linked list.

new nodes needs to be changed slightly to let the nodes of detached subtrees be available for future insertions. The following discussion transfers ideas mentioned in [54] to AVL trees and out of a database context.

Conceptually, the root nodes of the detached subtrees need to be saved in an auxiliary structure, e.g., a linked list. Whenever a new node needs to be allocated, this structure is first consulted. If it is non-empty, one of the roots, say node p, is reused as storage for the new node (otherwise the new node is allocated normally). Node p is removed from the structure and its children are saved back as new subtrees.

The auxiliary structure consists of a set of roots of subtrees that contain only deleted nodes. However, the size of the structure can get quite large, so it is useful to store it implicitly, using the deleted nodes themselves as storage. The roots in the structure should then form a linked list, where each root stores, in addition to its left and right children, a pointer to the next root in the list (e.g., in place of the key, which is unused in a deleted node) – see Figure 5.8. It is then easy to remove a node from this linked list and add its children to the front of the list, in constant time. The order of the roots in the list does not matter.

In external trees, internal nodes and leaves can be treated in the same way if they have the same size. If they have different sizes, a simple solution is to store leaf nodes in a separate linked list. When adding new subtrees or children to the structure, any leaf nodes can be added to the front of the leaf-node list instead of the internal-node list.

5.5.3 Analysis

The complexity of the bulk-deletion algorithm is analyzed below to be logarithmic in the number of deleted keys, when amortized over several deletions. The article [77] contains a similar result, but the following proof is more detailed.

Amortization is necessary because of the properties of AVL trees. In the worst case even single deletion may need to execute $O(\log n)$

rotations, where n is the number of nodes in the tree. This can be amortized to $O(1)$ rotations per deletion in a sequence of deletions, but not with mixed insertions and deletions – see Table 2.1.

Definition 5.1 *The (deletion) potential $\Phi_D(n)$ for a node n is:*

$$\Phi_D(n) = \begin{cases} 0, & \text{if } n \text{ is strictly balanced} \\ 1, & \text{otherwise.} \end{cases}$$

The potential $\Phi_D(T)$ of an AVL *tree T is the sum of the potentials of its nodes.*

Lemma 5.11 *Assume that an interval $[L, R]$ containing m keys is deleted from an* AVL *tree T with potential $\Phi_D(T)$. The bulk-deletion algorithm will use $O(\log m) + O(\Phi_D(T) - \Phi_D(T'))$ rotations to rebalance the tree, resulting in an* AVL *tree T' with potential $\Phi_D(T')$.*

Proof. Consider the path $p_1, \ldots, p_l$ from the position of L to the left child of the LCA node after actual deletion has been performed. (The rightward path is similar.) The siblings of $p_1, \ldots, p_l$ are consecutive siblings of a path in an AVL tree, except that some of the original siblings have been removed (detached) in the actual deletion.

At each point p_i where siblings have been removed (including p_1), the height difference d_i of two consecutive remaining siblings is limited by the number of siblings removed at this point: $d_i = h_{p_i} - h_{p_{i-1}} \leq 2j_i + 1$, where j_i is the number of siblings (subtrees) that have been removed (see Lemma 5.1). In addition, the height of the previous node p_{i-1} may have been reduced by one (thus increasing d_i) by the previous iteration of Algorithm 5.7. Therefore, at most $2j_i + 1$ rotations need to be done to fix the imbalance at p_i (Theorem 4.2). Each rotation can increase the potential of a constant number of nodes by 1, so the amount of work required at p_i is $O(j_i)$ when the potential is taken into account.

The height of the subtree rooted at p_i after the iteration may be reduced by one in the balancing (Theorem 4.1). Therefore, even if no further siblings were removed, a rotation may be needed at every level above p_i, but each of these rotations decreases the potential of the tree, since they are needed only when the height of the lower child of a non-strictly balanced node reduces by one.

By the above discussion, the total amount of work required below the LCA (including all j_is on both the leftward and rightward paths) is $O(J)$, where J is the total number of subtrees removed in actual deletion.

Consider then the LCA node l, having height h_l before actual deletion. If neither child node of l was deleted by the actual deletion, then the height of either child can decrease only by one in the balancing, and one rotation (and a possible potential increase of 1) is enough to correct the possible imbalance at l. If one or both child nodes of l were deleted, which means that a subtree of height at least $h_l - 3$ was detached, then there may be a height difference of at most h_l at l. The work needed to fix the imbalance at l is then at most $O(h_l) = O(h_d)$, where h_d is the height of the largest detached subtree (which is one of the grandchildren of l). If one child of l was completely emptied in actual deletion, Algorithm 5.6 will delete node l altogether, and in this special case no work is needed to rebalance l.

If both child nodes of l were deleted, it is possible that, after actual deletion and rebalancing at l, the height of the subtree originally rooted at l may be much lower than it was before the deletion. Then the height difference at the parent of l can be at most $h_l + 2$ (the extreme case occurs when the subtree rooted at l is completely deleted), and $O(h_l) = O(h_d)$ work is needed to rebalance the parent of l.

The height of the parent of l (or l itself, if both child nodes of l were not deleted) can decrease by only one (Theorem 4.1). Thus, above the parent of l, any further rotations decrease the potential of the tree (they are, again, needed only when the height of the lower child of a non-strictly balanced node reduces by one).

The total amount of rotations required for rebalancing, when the potential is taken into account, is then $O(J) + O(h_d)$. If m keys are deleted in total, the largest detached subtree can have height $h_d \leq \log_\Phi(m+1) = O(\log m)$ (where $\Phi = (1+\sqrt{5})/2$, Theorem 2.2). Subtrees are detached only from the two paths from the positions of L and R to the LCA node l, so there are two sets of consecutive siblings of paths in the original tree T which can be removed. Due to Lemma 5.1, there can be at most four siblings of each given height (two on either path). Then the number of subtrees $J = O(\log m)$. The amount of work needed is then $O(J) + O(h_d) = O(\log m)$. □

Next consider the total time complexity of rebalancing. Recall that three paths are rebalanced: P_l, P_r and P_t (lines 27–29 of Algorithm 5.6). On each path, above the highest detached subtree, rebalancing ends if a node on the path is in balance and its height value does not change. The discussion in the above proof showed that below the highest detached subtrees rebalancing takes time $O(\log m)$. Above this point on each of the three paths, the rebalancing algorithm does not do more

than a constant amount of work without either decreasing the potential accordingly or performing a rotation. This can be seen as follows.

At the original grandparent g of the highest detached subtree (the parent of a detached subtree is always deleted by line 11 of Algorithm 5.6), the height of g may decrease by at most one if a rotation does not need to be performed at g, because one child of g is unmodified. The same is true of the parent p of any node where a rotation was performed. If no more rotations are performed, the only way for a height decrease of one to propagate upward on the path is when the height of the higher child of a non-strictly balanced node q decreases. This, however, makes q strictly balanced, thus decreasing the potential of q by 1 to account for the necessary height value change.

Therefore, the amortized time complexity of rebalancing after bulk deletion is asymptotically the same as the number of rotations, and can be stated as follows.

Theorem 5.6 *Assume that a sequence of k single-bulk deletions is performed on an AVL tree with n keys, and that $m_1, \dots, m_k$ keys are deleted from each interval, with $\sum_i m_i = n$. The amortized complexity of rebalancing after deleting an interval with m keys is $O(\log m)$.*

If the interval to be deleted contains only one node, this bulk-deletion algorithm normally makes the same changes to the tree as the standard single-deletion algorithm, as did both of the bulk-insertion algorithms discussed above. (In an internal tree, only the LCA is deleted; in an external tree, only one leaf node.) However, there is one exception to this: lines 21–23 of Algorithm 5.6 that replace the LCA with its child if the other child is empty do not have an equivalent in the single-deletion algorithm, though it would be possible to add the same optimization to single deletion.

The total worst-case time complexity of bulk deletion of an interval $[L, R]$ with m keys from a tree with $n \geq m$ keys is $O(\log n)$, i.e., it is dominated by the time taken to traverse the tree searching for L and R. Contrast this with the $O(m \log n)$ time that would be used if each key was deleted using single deletion.

5.6 Deleting multiple intervals

Consider the case where multiple intervals $\{[L_1, R_1], \dots, [L_k, R_k]\}$ are to be deleted. Two approaches, similar to the simple and advanced approaches in bulk insertion, are possible.

With either approach, it is useful to begin by sorting the intervals to be deleted and checking for overlap between the intervals. Sorting makes the tree searches more localized, and the number of intervals can be reduced if they overlap. For instance, intervals $[L_1, R_1]$ and $[L_2, R_2]$ with $L_1 \leq L_2 \leq R_1 \leq R_2$ can be combined into a single interval $[L_1, R_2]$. Thus, bulk deletion with multiple intervals should begin with a preprocessing phase that sorts the list of intervals and merges any overlapping ones.

The straightforward approach is to process each interval separately: rebalance the tree fully after deleting an interval and before doing actual deletion for the next interval. In this simple approach, tree traversal is restarted from the root for each interval.

The remainder of this section will give a sketch of the advanced approach, which works similarly to Algorithm 5.5, the advanced approach for bulk insertion. Since rebalancing for bulk deletion is done using only the node balancing algorithm, as in the log-squared algorithm for bulk insertion, the additional complications in Section 5.4 arising from the logarithmic bulk-insertion algorithm are avoided. However, more bookkeeping is involved than in bulk insertion, since bulk deletion needs to process several paths (in the left and right children of the LCA).

Assume first that the algorithm would do all actual deletions one by one before any rebalancing. Consider the subtree formed by the union of all paths P_t, P_l and P_r produced by actual deletions of all the intervals (similar to the position tree in bulk insertion, but with more paths). Rebalancing can be done by executing the node balancing algorithm bottom-up on each node in this subtree, for example in post-order, as in the relaxed-balancing implementation of Section 4.3.

In Algorithm 5.6 for bulk deletion, downward tree traversal searched for both ends L and R of the next interval, starting from the root of the tree. Recall from Section 5.4 that the advanced approach for insertion avoids repeating root-to-leaf searches by looking bottom-up at the search path of the previous operation to find a node where the downward traversal can be begun. Because the (sorted) intervals to be deleted are processed from left to right, the search path used in this case is the rightmost path traversed when deleting the previous interval, i.e., P_t concatenated with P_r. The upward traversal can be stopped when the key of a node is larger than both L and R of the next interval: such a node is always located above the LCA of the next interval.

To improve locality in the tree searches, and to avoid having to keep track of the whole position tree, the actual bulk deletions and rebalancing should be interleaved, as was done in the advanced approach

for bulk insertion. After actual deletion of the interval $[L_i, R_i]$, rebalance the produced path P_l as usual. Then look for the next interval using the saved path $P_t P_r$. Assume that the downward traversal starts from a node $p \in P_t \cup P_r$; then the portion of the path $P_t P_r$ below p can be rebalanced before the downward traversal begins. Rebalancing for the next interval can proceed in the same way, until a node in the path $P_t P_r$ is reached, at which point rebalancing can be continued on the remainder of the path $P_t P_r$. Thus, at most two paths (leading to L and R) need to be saved at any one time.

A final point for consideration in this sketch of the advanced approach is the optimization that bottom-up rebalancing can be stopped early if the tree is not changed at some point (lines 3–4 of Algorithm 5.7). The optimization is also valid here, but as in bulk insertion, rebalancing needs to be restarted from the next branching node on the path, i.e., the node p mentioned in the discussion above.

5.7 Comparison to relaxed balancing

When using relaxed balancing, as described in Section 4.3, rebalancing is decoupled from the actual updates (single insertions and single deletions). A single rebalancing operation may then need to take into account several insertions or deletions done in the same part of the tree. This is closely related to the rebalancing done by a bulk-update algorithm, and it is perhaps instructive to compare the algorithm described in Section 4.3 with the bulk-insertion and bulk-deletion algorithms of this chapter.

Bulk update operations have some advantages compared to the using single insertions and deletions combined with relaxed balancing. First, bulk insertion directly creates a balanced tree that contains all insertions to a specific position, while relaxed balancing needs to create nodes in positions dictated by the order of the elements in the "bulk". Relaxed balancing may need to traverse long paths inside the "update tree" (since the newly created subtree has not yet been balanced) and even do rebalancing inside the update tree. For instance, if keys are inserted in sorted order, the new subtree created by relaxed balancing degenerates into a linked list before rebalancing is performed. Second, single insertions require a root-to-leaf search for the position of every new key, unless some kind of finger tree (see Section 2.11) is used. These advantages are slightly offset by bulk insertion needing to sort the new keys first.

Third, bulk insertion can also make use of the more efficient logarithmic rebalancing algorithm. If single insertions and the relaxed balancing approach were used to insert keys that would form one bulk, the rebalancing implied by Section 4.3 would behave like the log-squared rebalancing algorithm (after the update tree itself has been balanced).

For bulk deletion, the primary advantages are that the deletion of individual nodes in large subtrees can be deferred because of the detach-subtree operations, and that the relaxed balancing approach needs to traverse a subtree many times in order to single-delete all of its nodes. If the algorithm described in Section 4.3 were used for rebalancing after actual bulk deletion, the rotations performed would actually be the same as with the bulk-deletion algorithm, though slightly more bookkeeping would need to be done to keep track of which parts of the tree need rebalancing.

5.8 Experiments

This section briefly reports on experiments made with the bulk-insertion and bulk-deletion algorithms. The experiments used internal height-valued AVL trees and the experimental setup described in Section 3.5): a C implementation running on an AMD Athlon XP at 2167 MHz.

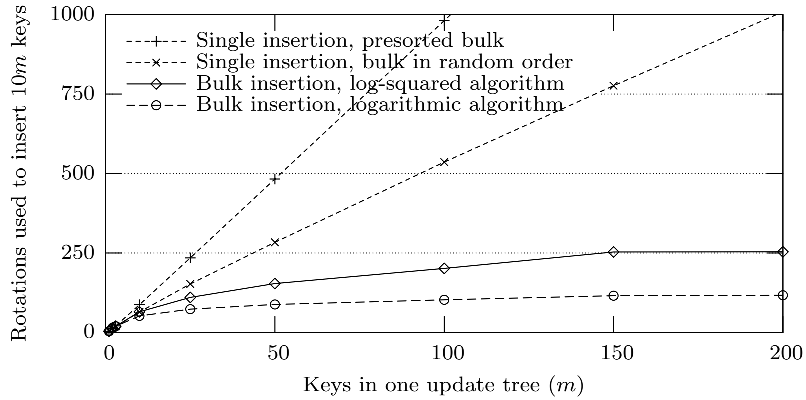
In all cases, an initial tree was first formed using 10^6 single insertions in random order. Then bulk insertion or bulk deletion was performed using keys that create 10 bulks (or intervals), each of size m, for $m = 1, \ldots, 50000$. The experiments were repeated 15 times for each m using new random keys.

5.8.1 Bulk insertion

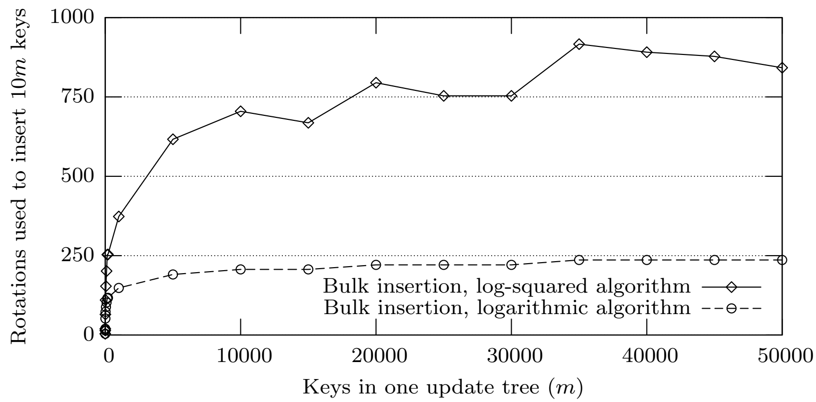
The log-squared and logarithmic rebalancing algorithms were compared with each other and with a trivial algorithm that inserts a bulk using repeated single insertions. Repeated single insertions perform differently if the keys of a bulk are in random order than if they are first sorted; both cases were examined here.

Figure 5.9 shows the average number of rotations performed when inserting 10 bulks of m keys each into the tree (using the simple approach of Algorithm 5.4). Figure 5.10 gives the average CPU time used when inserting these $10m$ keys.

The results in Figure 5.9 are not surprising. The logarithmic algorithm clearly outperformed the log-squared algorithm, and both were



(a) Small bulks



(b) Large bulks

Figure 5.9 Rotations used by bulk insertion.

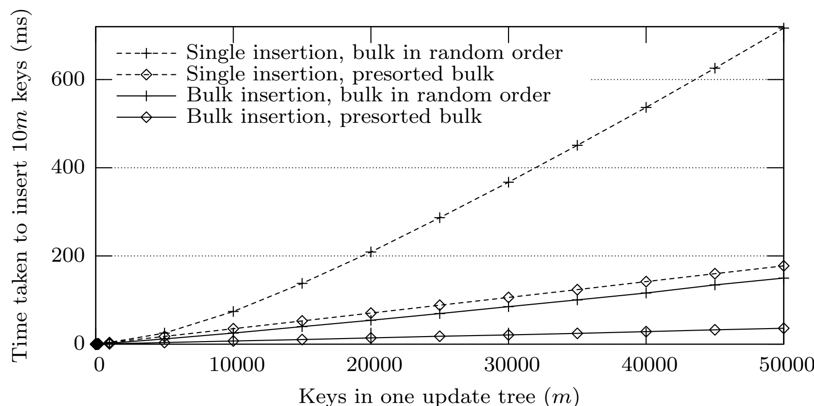


Figure 5.10 Time spent in bulk insertion. The logarithmic algorithm for bulk insertion is shown; the log-squared algorithm was only slightly slower.

much better than the linear cost of single insertions. Bulk insertion used fewer rotations than single insertion for all bulk sizes with more than a few keys. The CPU times in Figure 5.10 are dominated by creating the m new nodes; nevertheless, bulk insertion performed considerably better than single insertion.

Figure 5.11 compares the simple and advanced approaches for multiple bulks given in Algorithms 5.4 and 5.5. In the previous experiments, the 10 update trees for each bulk insertion were placed at (uniformly distributed) random positions in the tree. Here, the distance of one update tree to the next is fixed: new keys are selected so that there are exactly a given number of keys in the initial tree between two position leaves. The results are averaged for 15 random initial trees (each containing $n = 10^6$ keys) and random positions for the leftmost update tree.

The values in Figure 5.11 estimate the traversal cost by measuring the number of key comparisons performed for an update tree size of $m = 10000$ ($m = 1000$ was similar). Comparisons performed while searching for the end of each bulk were not counted – when creating an update tree, both implementations compared each new key except for the first to the maximum key possible at the current position leaf, both doing $10m - 1$ comparisons which are here ignored.

The advanced approach clearly used much fewer key comparisons for small update tree distances. With large distances, the simple ap-

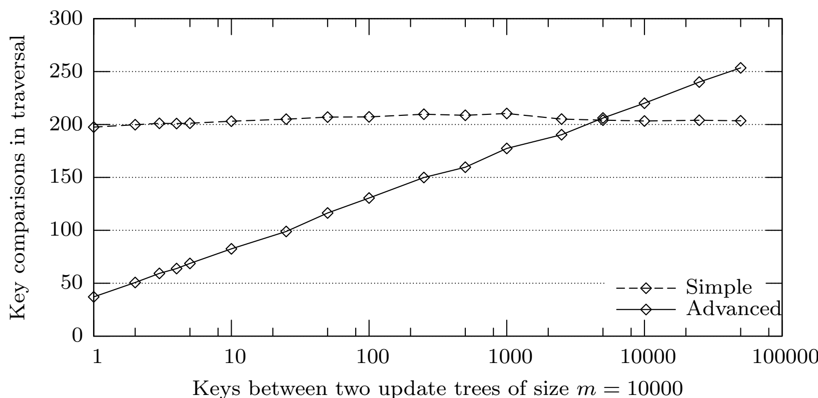


Figure 5.11 Comparison of simple and advanced approaches for multiple bulks. The logarithmic rebalancing algorithm was used in all cases. The y-axis gives the number of key comparisons done during tree traversal when inserting $10m$ keys.

proach was somewhat better; the advanced approach traverses search paths upwards when looking for the next position leaf, but when the update trees are far from each other a simple root-to-leaf search would be more efficient.

5.8.2 Bulk deletion

Figure 5.12 gives the average number of rotations performed when deleting 10 non-overlapping intervals of m keys from random locations from a tree with $n = 10^6$ keys, using single-bulk deletions. Figure 5.13 gives the average CPU time used when deleting the 10 intervals (having a total of $10m$ keys), for two versions of bulk deletion: one which detaches whole subtrees, and another that visits each individual node to delete it. The latter is called "Bulk deletion with node deallocation" in Figure 5.13 and in Table 5.3.

The experiments compare bulk deletion to a simple method in which single deletions are used to delete the same keys in order from smallest to largest. Each key is deleted as a separate deletion operation – that is, a root-to-leaf search is performed for every key. Some sort of finger tree (see Section 2.11) could be used to make tree traversal in the single deletions somewhat more efficient, but the number of rota-

	Number of nodes read			*Bulk deletion*	
m	*Single deletion*	*Bulk deletion*	*Bulk del. w/dealloc.*	*Subtree detachments*	*Node deletions*
1	237	242	242	0.0	10.0
10	384	328	383	34.4	44.4
100	1384	458	1380	67.3	77.3
1000	10476	577	10467	100.3	110.3
10000	100555	695	100549	136.5	146.5
25000	250606	763	250603	150.8	160.8
50000	500615	773	500601	161.9	171.9

Table 5.3 Read-set sizes and subtree detachments in bulk deletion of $10m$ keys.

tions would not change. Single-deleting the keys in random order could make the number of rotations somewhat smaller (as in insertion, see Figure 5.9), but naturally still linear in the number of deleted keys.

Table 5.3 shows the number of separate nodes read (i.e., the size in nodes of the read set of the algorithm) during deletion of the $10m$ keys, as well as the number of subtree detachments performed by bulk deletion. Bulk deletion with subtree detachment still deletes some individual nodes (e.g., node n in line 11 of Algorithm 5.6) – these are given in the rightmost column of the table.

The results show that bulk deletion is very efficient: fewer than 120 rotations and a fraction of the single-deletion time was used to delete 500000 keys. The extremely small running time is explained by not having to visit most of the individual nodes – the version of bulk deletion that deallocates each individual node was only about 30% faster in running time than single deletion, and had about the same read-set size. For instance, for $m = 50000$, bulk deletion needed to read about 773 nodes on average and performed 162 subtree detachments to delete $10m$ keys, compared to the 500615 nodes that were read by the $10m = 500000$ single deletions.

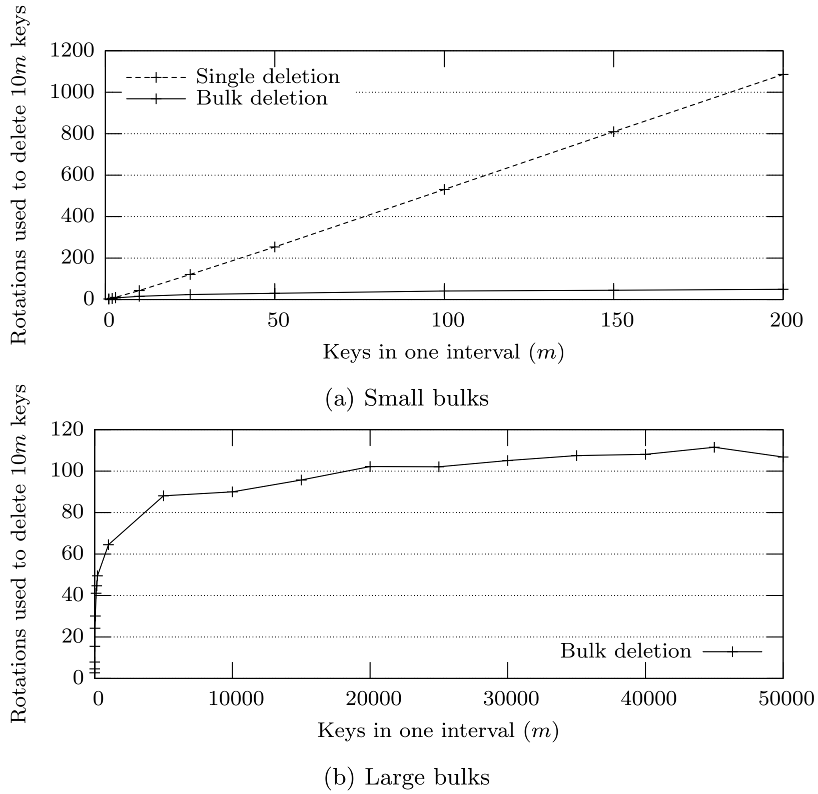


Figure 5.12 Rotations used by bulk deletion.

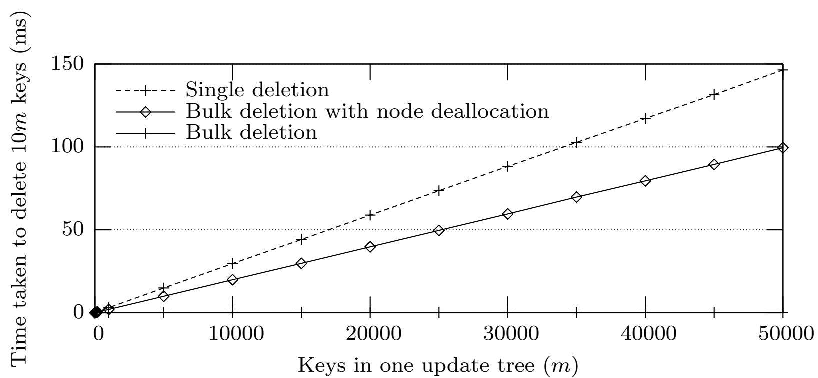


Figure 5.13 Time spent in bulk deletion. The values for “Bulk deletion” are very small, in the range 0.03–0.10 ms.

CHAPTER 6

Using bulk insertion in adaptive sorting

This chapter examines the application of bulk insertion in an adaptive sorting algorithm. The chapter begins by discussing a finger structure adapted to AVL trees, and then explains how to find bulks to be inserted using the bulk-insertion algorithm. The next topic is an alternative approach that uses bulks but does not need an actual bulk-insertion algorithm. The chapter finishes with analytical and experimental results about the new algorithms.

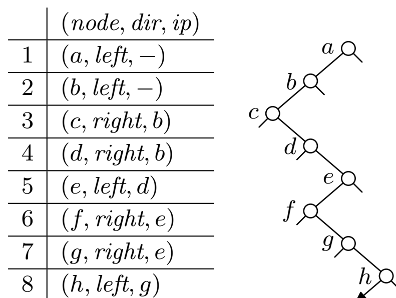
The reader is referred to Sections 2.10 and 2.11 for brief introductions to adaptive sorting and finger trees.

Some of the results presented in this chapter appear in [74] by the author and supervisor of this thesis. However, the published article has much less detail, especially regarding the implementation, and does not include the bulk tree (but includes some analytical results on so-called composite measures of presortedness that are not present here).

6.1 Simplified finger: Binary saved path

Section 2.11 explained that a finger is a pointer to a node in the search tree, using which other nearby nodes can be accessed without starting the search from the root. In search-tree-based adaptive sorting, the finger is used as a starting point for finding the place of the next key to be inserted.

As described in Section 2.11, this chapter will present a simplified finger structure that is less complex to implement than a full finger, but not quite as effective. This simplified finger structure for AVL trees (actually for any binary search tree) will be called the *binary saved path*, because it resembles the saved path used in database systems.



	$(node, dir, ip)$
1	$(a, \mathit{left}, -)$
2	$(b, \mathit{left}, -)$
3	(c, right, b)
4	(d, right, b)
5	(e, left, d)
6	(f, right, e)
7	(g, right, e)
8	(h, left, g)

Figure 6.1 Binary saved path construction: When the search advances to the left child of node h, the new entry pushed onto the binary saved path is $(h, \mathit{dir}(h), \mathit{ip}(h)) = (h, \mathit{left}, g)$.

The saved path [61] is a stack that stores the last accessed root-to-leaf path of a B-tree. For each node on the root-to-leaf path, the saved path contains a pointer to the node and the minimum and maximum router keys of the node (as well as some additional data related to concurrency). The saved path is used by all search tree operations: a search for a key k begins by examining the saved path in a bottom-up fashion, looking for the lowest node that *covers* k, i.e., k is greater than or equal to the minimum router and smaller than the maximum. A standard downward tree search is then begun from the node from this entry (in the worst case, at the root).

Examining an entry in the traditional saved path requires two comparisons: k is compared to the minimum and maximum routers. In internal binary trees, the number of comparisons can be reduced to one as follows. (Internal trees are better suited for adaptive sorting than external trees because of their smaller space usage, as noted in Section 2.4.)

Like the saved path, the *binary saved path* is a stack that describes a root-to-leaf path in the binary tree. Each entry in the binary saved path describes a parent-to-child edge in the tree – for instance, "the left child of node p". Specifically, each entry is a triple

$$(p, \mathit{dir}(p), \mathit{ip}(p)),$$

where $\mathit{dir}(p)$ is either *left* or *right* and denotes the direction of the edge down from node p (i.e., which child of p is the next entry on the path), and $\mathit{ip}(p)$, the *indirect parent* of p, is the lowest node above p where the direction is different from $\mathit{dir}(p)$. (See Figure 6.1.) The lowest entry of the path generally points to the node that was last inserted.

The maximum and minimum router keys are found directly from each entry. For an entry $(p, \text{left}, ip(p))$, the maximum router is the key of p and the minimum is the key of the indirect parent. For an entry $(p, \text{right}, ip(p))$, the minimum router is the key of p and the maximum is the key of the indirect parent.

Theorem 6.1 *Using the binary saved path, the lowest entry in the path that covers a key k can be found using at most s key comparisons, where s is the number of entries in the path.*

Proof. For an entry in the saved path to cover key k, k needs to be between the key of the node p in the entry (the "direct parent") and the key of the indirect parent $ip(p)$ (which is one of the nodes higher up on the saved path). The key k is compared with the indirect parent first, because if this comparison fails, the saved path entries between the indirect and direct parents can be skipped: none of them can cover k because of the indirect parent.

Each entry on the saved path thus gives a new minimum or maximum limit for k, so k needs to be compared to each entry at most once. □

A rotation done on a node p which is on the saved path can make the saved path structure invalid. However, it is straightforward to update the saved path after each rotation to reflect the change in the tree structure: the saved path should finally contain the path from the root to the previously inserted node. For example, if the saved path contains an entry "left child of node p", a right single rotation done at p should remove this entry from the path.

6.2 Finding ascending or descending sequences

A subproblem that arises in using bulk insertion for sorting is how to find the sequence of keys to be inserted. Given a list of keys $K_1, \ldots K_n$, and minimum and maximum key values s and g, where $s < K_1 < g$, the problem is to find the longest ascending or descending prefix $K_1, \ldots, K_m$ where each key is in the given range: $s < K_i < g$ for $i = 1, \ldots, m$, $1 \leq m \leq n$. When s and g are consecutive keys already present in a search tree, $K_1, \ldots, K_m$ can all be inserted in the same location in the tree. As is usual in adaptive sorting, it is assumed that the input data does not contain duplicate keys.

If $K_1 < K_2$, we look for an ascending prefix. A trivial way of finding the end m of the prefix uses two comparisons per element: one to see if the key is larger than the previous one, and another to see if the key is smaller than g. This requires at most $2m - 2$ comparisons to find a prefix with m keys. If $K_1 > K_2$, a descending prefix is found in the same manner. To simplify the presentation, $K_1 < K_2$ is assumed below.

After finding one prefix $K_1, \ldots, K_m$ and bulk-inserting it, the algorithm is repeated for the remaining keys $K_{m+1}, \ldots, K_n$, now possibly with different values of the limits s and g. In this way, the keys $K_1, \ldots, K_n$ are split into ascending or descending sequences (called *bulks*) within the given limits.

The number of comparisons can be reduced from $2m - 2$ to $m + O(\log m)$ per sequence of length m as follows. Assuming that the prefix is ascending, it is not necessary to compare every element to the maximum. Instead, a variation of *exponential and binary search* [58] is applied, first comparing only the second, fourth, 8th, 16th, 32nd, ... key to the maximum. When one of these comparisons fails (say, the 32nd key is above the maximum), a binary search is done in the last interval (e.g., keys 17 to 31) to find the last element below the maximum. Whenever an element K_i that is smaller than the maximum is found, in either the exponentially increasing search or in the binary search, we need to ensure that the prefix $K_1, \ldots, K_i$ is ascending before continuing. To achieve $m + O(\log m)$ comparisons, the "ascending-comparisons" $K_{i-1} < K_i$ must not be repeated. This is done simply by saving the index a before which the sequence $K_1, \ldots, K_a$ is known to be ascending.

However, the above approach may use more than m ascending-comparisons when finding a single prefix of m keys. For example, if the 32nd key is below the maximum in the exponentially increasing search, we then check that keys 17 to 32 are ascending. Assume that the 31st key is not larger than the 30th; then a binary search for the maximum is done in keys 17 to 30. We then find that the last element below the maximum is actually the 17th key, and create a bulk of size $m = 17$, but we have used $31 + O(\log m)$ comparisons.

When finding all ascending or descending sequences in $K_1, \ldots, K_n$, repeating any of the ascending-comparisons is avoided by saving the index a for the next sequence. In the example, when searching for the next sequence starting from the 18th key, we know from the current value of a that all keys up to the 30th are ascending. A flag is also saved noting whether the sequence up to a is ascending or descending.

Theorem 6.2 *Given limits s and g and a list of keys $K_1, \ldots, K_n$, finding maximal ascending or descending sequences of consecutive keys where each key k satisfies $s < k < g$ can be done with $m + O(\log m)$ comparisons per sequence of length m. The limits s and g may be different for each sequence.*

Proof. Because of the saved index a, every key is compared to the previous one (to find out if the sequence is ascending or descending) only once. All remaining comparisons are comparisons with the maximum (ascending) or minimum (descending) element. In the ith bulk, the exponentially increasing search does $O(\log m_i)$ comparisons – the second-to-last element compared is always included in the bulk. The binary search is done in an interval of size $O(m_i)$ and thus also uses $O(\log m_i)$ comparisons. □

6.3 Bulk-insertion sort in an AVL tree

Recall from Section 2.10 that the local insertion sort algorithm [56] inserts the values to be sorted in a finger search tree one by one. Implementing the finger using the binary saved path in an AVL tree leads to an AVL-tree-based variation of this algorithm.

Applying bulk insertion to the AVL-tree-based algorithm is simple: after finding the position of the next key to be inserted, create a bulk of as many keys as fit in this position, and insert them all using the bulk-insertion algorithm of Section 5.1 with the logarithmic rebalancing algorithm of Section 5.3. In this application, bulks are inserted one at a time, so single-bulk insertion is enough and the methods of Section 5.4 for inserting multiple bulks are not needed.

Finding the keys that fit in the insertion position is done using the algorithm of the previous section. The values s and g required by bulk-finding should be the next-smaller and next-larger key values present in the tree after searching for the position where the first key is to be inserted. They are found from the saved-path entry $(p, dir(p), ip(p))$ that points to where the first key should be inserted: s and g are the keys of the direct parent p and indirect parent $ip(p)$.

Algorithm 6.1 gives an outline of the bulk-insertion sorting algorithm. Figure 6.2 shows an example of which keys form bulks when a sequence of keys is sorted using this algorithm.

It was described in Section 5.1 that the bulk insertion algorithm first needs sort the keys of the new bulk. However, in this case sorting

BULK-INSERTION-SORT($A[1..n]$):

1 $k \leftarrow 1$
2 $P \leftarrow$ empty binary saved path
3 $T \leftarrow$ empty AVL tree
4 **while** $k \leq n$ **do**
5   Search bottom-up in the binary saved path P for the lowest position $P[s]$ with $key(P[s]) < A[k] < key(ip(P[s]))$ or $key(P[s]) > A[k] > key(ip(P[s]))$.
6   Search for $A[k]$ in T starting from $P[s]$. Save this path to $P[s+1..d]$.
7   Use the algorithm of Section 6.2 to find the longest ascending or descending sequence of keys $A[k..l]$, where $key(P[d]) < A[i] < key(ip(P[d]))$ or $key(P[d]) > A[i] > key(ip(P[d]))$ for all $i \in [k, l]$.
8   Bulk insert $A[k..l]$ in T at the position pointed by $P[d]$.
9   $k \leftarrow l + 1$
10 **return** T

Algorithm 6.1 Bulk-insertion sort.

$\underbrace{4\ 8\ 9\ 20\ 22\ 23}\ \underbrace{7\ 6\ 5}\ \underbrace{3\ 2\ 1}\ \underbrace{19\ 14\ 13\ 10}\ \underbrace{15\ 16\ 17\ 18}\ \underbrace{21}\ \underbrace{12\ 11}$

Figure 6.2 Example of bulks created in bulk-insertion sort and bulk-tree sort. Bulk-insertion sort uses 7 bulk insertions to sort this sequence of 23 keys.

the bulk is of course unnecessary, since we know that its keys are in ascending or descending order. Therefore, the update tree should be created out of the sequence $A[k..d]$ simply by placing the middle element $A[\lfloor(k+d)/2\rfloor]$ at the root of the update tree and proceeding recursively.

6.4 Lazy bulks

The AVL-tree bulk-insertion algorithm of Chapter 5 uses $O(\log m)$ amortized time to rebalance the tree after a bulk of m keys is inserted (Theorem 5.3) – but creating the m new nodes still requires $O(m)$ time. However, in the case where bulk-insertion sort produces mainly large bulks, the sorting algorithm does not actually visit most of the nodes in each bulk, so creating all of the nodes is not necessary.

This observation can be used to reduce the running time of bulk-insertion sort as follows, assuming that keys $A[1..n]$ are to be sorted. Instead of creating an update tree from keys $A[i]$ to $A[i+m-1]$, create a single "lazy" placeholder node that contains the values i and m instead of the normal key and child pointers.

In the implementation used in the experiments, a special, otherwise unused, address for the left child pointer was used to mark a lazy node. The right child pointer and key were used to store m and a pointer to $A[i]$, as well as a flag that notes whether the sequence $A[i]$ to $A[i+m-1]$ is ascending or descending.

The lazy node is placed where the root of the new update tree should be. Later, the lazy node can be expanded, in constant time, to a normal node (with key $A[i+\lfloor m/2 \rfloor]$) with two new lazy nodes as its children (one with keys $A[i]$ to $A[i+\lfloor m/2 \rfloor - 1]$, and the other with keys $A[i+\lfloor m/2 \rfloor + 1]$ to $A[i+m-1]$). A lazy node p is expanded whenever any part of the bulk-insertion sorting algorithm follows a child pointer that leads to p, either during rebalancing or during a future finger search.

The use of lazy bulks does not change the rebalancing strategy of the tree: the lazy node acts like a normal node that has the height of the expanded update tree. The height is easy to calculate from m.

Lazy bulks have the disadvantage that every time a left or right child pointer is followed, the child node needs to be checked for laziness. Experiments done for Section 6.7 indicated that the slowdown caused by this is very small compared to the advantage of lazy bulk creation.

Strictly speaking, lazy bulks change the output format of the sorting algorithm, since some of the leaves of the AVL tree are placeholder nodes. It is, however, still very easy to access the keys in sorted order: do the usual in-order traversal of the tree, but whenever a placeholder node with values i and m is reached, output the values $A[i]$ to $A[i+m-1]$ before continuing the in-order traversal.

6.5 The bulk tree

It is possible to apply the bulk-finding algorithm of Section 6.2 to adaptive sorting even without using an actual bulk-insertion algorithm like the one in Chapter 5. This can be done by modifying the structure of the binary search tree as follows.

Instead of storing single keys in nodes, each node stores a *bulk* of several keys that are consecutive in the original array A to be sorted. Specifically, an internal AVL tree is used where each node contains a pointer to a key $A[i]$, the number of keys m in the bulk, and a flag noting whether the bulk is ascending or descending (in addition to the left and right child pointers and the AVL-tree balancing direction). Each node then designates the ascending or descending sequence of keys $A[i]$ to $A[i+m-1]$. This modified AVL tree is called a *bulk tree*.

Searching for the next key k to be inserted is done using the binary saved path of Section 6.1. Thus, we first traverse the saved path bottom-up to find the lowest entry that covers k. For the purposes of this search, the router key of a node p in the saved path is either the smallest or largest key of the bulk stored in node p: the smallest key is used if $dir(p) = left$ (since the smallest key sets a maximum for the keys stored in the left child of p), the largest otherwise.

After the lowest saved-path entry that covers k is found, the search proceeds downwards in the tree. The downward search is done by comparing k to the smallest key a and largest key b of the bulk in each node. If $k < a$ and $k < b$, the search descends to the left child, and respectively to the right if $k > a$ and $k > b$. If an empty location in the tree is found in this way, a new bulk containing k and any other keys found using the bulk-finding algorithm of Section 6.2 is inserted (with a possible rotation afterwards, as when inserting a single node to a normal AVL tree).

If, however, $a < k < b$ at some point, the bulk $[a..b]$ needs to be split in two parts to accommodate the new key k. Three things are then done. First, a binary search for k is performed in the bulk (which is an ascending or descending sequence of keys), and the bulk is split into two halves: S (containing keys smaller than k) and L (containing keys larger than k).

Second, the bulk-finding algorithm is run to find a new bulk K containing k and possibly some other keys. Here the minimum key value s in the bulk-finding algorithm is set to the largest key of S, and the maximum g to the smallest key of L (instead of taking these limits from the saved path).

Third, the three bulks S, K and L are inserted in the tree, for example by replacing the original bulk $[a..b]$ with L and inserting S and K as new nodes to the rightmost end of the left child of the original bulk (i.e., the position where the next-smaller node should be – note that no key comparisons are needed to find this). The new nodes S and K are inserted separately, with a possible rotation after both insertions.

Every insertion of a new bulk thus either creates a single new node, or (in the bulk-split case) modifies the contents of one node and inserts two others.

After all keys have been inserted in this manner, the sorted sequence can be retrieved by doing a standard in-order traversal of the bulk tree and printing out the bulks.

Although an AVL tree was used above (and in the experiments below), the bulk tree could just as well be, for instance, a red-black tree.

However, the tree should be internal to avoid special cases having to do with the router keys of external trees.

6.6 Analysis

As noted in Section 2.11, the binary saved path is not a full-fledged finger. More specifically, it does not use worst-case $O(\log d)$ time to move a distance d in the tree. It is thus not optimal with respect to the *logDist* [42] or *Loc* [65] measures used to characterize local insertion sort using a finger tree. However, the binary saved path is optimal with respect to the number of inversions:

Theorem 6.3 *Local insertion sort implemented using the binary saved path takes time $O(n \log(1 + \mathit{Inv}/n))$, which is optimal with respect to Inv (the number of inversions).*

Proof. It is known that an approach that keeps the finger always pointing to the largest key in the tree is *Inv*-optimal [82]. Consider an insertion to a position x elements away from the largest key l, and assume that the previous insertion was one at a position p elements away from the largest key. Moving the binary saved path from p to x costs at most as much as moving a finger from p to l and from l to x (the time complexity is $O(\log p + \log x)$). Therefore, the binary saved path is M-optimal for any measure M for which the "finger at largest key" approach is M-optimal.

The approach of [82] does not actually move the finger from p to l, but the previous insertion has moved from l to p in the tree, so we can move the finger back to l without increasing the cost by more than a constant factor. □

Next consider the bulk-insertion sorting algorithm of Sections 6.3 and 6.4.

Lemma 6.1 *Consider a measure M of sortedness. If local insertion sort implemented using the binary saved path is optimal with respect to M, bulk-insertion sort is also M-optimal.*

Proof. The two algorithms are essentially identical when a bulk of size $m = 1$ is inserted. Consider then a larger bulk containing $m > 1$ elements in an ascending or descending sequence.

After the position of the first key is found, local insertion sort uses $O(m)$ key comparisons in the amortized sense to insert the elements one

by one: since the keys are in ascending or descending order, the location for the next insertion is a constant distance away from the previous one, and each single insertion takes time $O(1)$ when traversing the tree using the binary saved path.

Bulk-insertion sort takes at most the same amount of time: after the position of the first key is found, $O(m)$ comparisons are required for finding the bulk (Theorem 6.2) and amortized $O(\log m)$ time for rebalancing and tree traversal (Theorem 5.3). □

Theorem 6.4 *Bulk-insertion sort is Inv-optimal.*

Proof. Implied by Lemma 6.1 and Theorem 6.3. □

It is currently not known whether bulk-tree sort is *Inv*-optimal or not, although it appears in the experiments in Section 6.7 to be at least strongly adaptive to *Inv*.

Bulk-insertion sort and bulk-tree sort are both optimal with respect to a new measure *Bulk*, first defined in the article [74] (by the author and supervisor of this thesis) mentioned at the start of this chapter. The measure *Bulk* is defined as the number of bulks in the input X, where a *bulk* B is a maximal-length ascending or descending sequence of keys where no keys in X before B are between the smallest and largest keys in B. The bulk-finding algorithm in Section 6.2 is used to find these bulks, so Figure 6.2 also gives an example of the bulks used by the *Bulk* measure.

The lower bound for the measure $k = \mathit{Bulk}(X)$ is $\Omega(n+k\log k)$ [74, Lemma 1].

The *Block* measure – defined in Section 2.10 as the number of blocks of consecutive elements in the original sequence that are also present in the sorted sequence – is closely related to *Bulk*: each block is completely contained in a bulk, and each bulk contains one or more blocks. For example, Figure 6.2 contains 7 bulks and 18 blocks: the blocks are $\{8, 9\}, \{22, 23\}, \{15, 16, 17, 18\}$, and all the other keys as one-element blocks.

Theorem 6.5 *Bulk-insertion sort is Bulk-optimal.*

Proof. Assume that the input data contains n keys that form k bulks. Ignoring the creation of the update tree and the bulk-finding algorithm (Section 6.2), inserting one bulk can be done in $O(\log n)$ time. The total time spent in creating the nodes in the update trees and in bulk-finding is $O(n)$.

The time complexity of bulk-insertion sort is then $O(n + k \log n)$. Since $1 \le k \le n$, it follows that $n = ck$ where $1 \le c \le n$. Then:

$$\begin{aligned} k \log n &= \frac{n}{c} \log(ck) = \frac{n}{c} \log c + \frac{n}{c} \log k \\ &\le n + \frac{n}{c} \log k \\ &= n + k \log k. \end{aligned}$$

Thus $O(n + k \log n) = O(n + k \log k)$, which matches the lower bound of the *Bulk* measure. □

Theorem 6.6 *Bulk-tree sort is Bulk-optimal.*

Proof. Assume that the input data contains n keys that form k bulks. As noted in Section 6.5, inserting each bulk in the bulk tree creates one or two new nodes, so the bulk tree has $O(k)$ nodes after all the insertions. Since the bulk tree is structurally an AVL tree, the $O(k)$ single insertions take time $O(k \log k)$. Finding the bulks (Section 6.2) requires $O(n)$ time in total.

Additionally, each insertion of a new bulk may do one binary search inside a previously inserted bulk. Since the size of the largest bulk is bounded only by $O(n)$, the binary searches may need $O(k \log n)$ time.

The total complexity of bulk-tree sort is then $O(n+k \log n)$, which is *Bulk*-optimal using the argument of Theorem 6.5. □

Note that the use of the simplified finger was not required for *Bulk*-optimality in either algorithm.

Lazy bulks in bulk-insertion sort (Section 6.4) are not required for *Inv*- or *Bulk*-optimality, since creating all nodes in the update tree (or expanding every lazy node) takes time $O(\sum_i m_i) = O(n)$, if the input (of size n) forms k bulks with $m_1, \ldots, m_k$ keys. However, lazy bulks reduce this cost:

Theorem 6.7 *Assume that the input to be sorted forms k bulks containing $m_1, \ldots, m_k$ elements. When lazy bulks are used in bulk-insertion sort, inserting the bulks (without searching or bulk-finding) takes time $O(\sum_i \log m_i)$.*

Proof. Consider the insertion of one bulk of size m. Creating the update tree (i.e., the single lazy node) takes $O(1)$ time. Rebalancing may need to expand some of the nodes in this update tree and in the update trees of previous insertions. However, rebalancing after bulk insertion takes $O(\log m)$ amortized time (Theorem 5.3), and can thus expand at

most $O(\log m)$ lazy nodes (also amortized). Thus, inserting a bulk of m keys to a given position in the tree can be done in $O(\log m)$ amortized time. □

Applying bulk insertion, with or without lazy bulks, naturally reduces the number of rotations. The following follows directly from Theorem 5.2.

Theorem 6.8 *Assume that the input to be sorted forms k bulks containing $m_1, \ldots, m_k$ elements. Then the total number of rotations needed by bulk-insertion sort is $O\big(\sum_{i=1}^{k} \log m_i\big)$.*

Without using bulk insertion or the bulk-tree, the number of rotations is of course $O\big(\sum_{i=1}^{k} m_i\big) = O(n)$.

6.7 Experiments

This section reports on experiments performed comparing the adaptive sorting algorithms described above with each other and with a few other algorithms, on randomly generated input data having a varying number of inversions. Local insertion sort was implemented using the binary saved path and internal height-valued AVL-trees, with and without bulk insertion (these are called Single-insertion sort and Bulk-insertion sort below), and additionally the bulk tree of Section 6.5.

The performance was compared to two *Inv*-optimal algorithms – Splaysort [60] and Splitsort [53] – and to standard Quicksort, Insertion sort (see [58], for example), and the `qsort` function in the C library. Splitsort is known to be efficient especially in terms of running time [23, 26] and Splaysort in the number of comparisons [23, 60].

The experiments were done using the experimental setup described in Section 3.5: C implementations running on an AMD Athlon XP at 2167 MHz. Each experiment was repeated 10 times using newly generated input – the numbers given are averages of these.

6.7.1 Implementations

Despite the name, the `qsort` function is actually a straightforward implementation of Merge sort in current versions of the GNU C library, which is commonly used under Linux. The source code (of version 2.3.6, which was used in these experiments) reveals that `qsort` uses Merge

sort, unless there is a problem allocating memory for the needed additional $O(n)$ space, in which case it falls back to an in-place Quicksort implementation. The fallback was avoided in these experiments.

The Quicksort implementation in these experiments used the deterministic version from [13], specifically [13, Figure 1] with the pivot selection changed to always use the middle element as the pivot. In [13], this version had the smallest number of comparisons when the number of inversions was small. The textbook optimizations of median-of-3 partitioning and fallback to Insertion sort for $n \leq 10$ were also attempted, but this improved performance only very slightly.

The Splaysort implementation was the one from [60]. Splitsort was reimplemented for these experiments, but it was found that the space optimization mentioned in [53] that uses only n pointers of extra space was much slower than a version that uses $2n$ extra pointers to avoid copying data back and forth in the various phases of the algorithm. Thus, the faster $2n$-space version was used in the experiments.

To obtain comparable running times, the output of each sorting algorithm was always written to an array. That is, when using the tree-based algorithms (Single-insertion sort, Bulk-insertion sort, Bulk-tree sort, Splaysort), all nodes in the created tree were finally traversed and the sorted result written back into the original array.

6.7.2 Input data

Both integer and string keys were examined. The integer keys (word-sized integers in the range $[1, n]$ with $n = 2^{25}$) give very fast comparisons. The string keys were taken from a list of about 2 million file names sorted in lexicographic order.* Most of the strings contain similar prefixes so that string comparisons often need to look for more than the first few characters to differentiate between the strings.

The input data was generated so that the full extent of inversion adaptivity was examined, from completely sorted sequences to completely random ones. This required using three different methods to generate input.

The first input-generation method produced small amounts of inversions (0 to about n) using the algorithm of [25] applied to the *Inv* measure. This algorithm exchanges k randomly chosen pairs of adja-

* Specifically, the list of file names in the Debian GNU/Linux distribution release 4.0r2, `http://ftp.debian.org/dists/Debian4.0r2/Contents-i386.gz` with duplicates removed.

cent elements, starting from a sorted sequence, thus generating about k inversions on average (unless k is too large).

The second method was the algorithm described in [23], which was used to produce larger numbers of inversions: about n to $n^2/8$. This algorithm generates about $mn/2$ inversions on average, by first dividing a sorted sequence into $\lceil n/m \rceil$ equal-sized blocks and permuting the elements in each block into random order, and then selecting a random element from each of m equal-sized blocks and permuting the selected elements into random order.

Finally, the third method created random (non-adaptive) sequences where every permutation is equally likely – which gives about $n^2/4$ inversions on average. This non-adaptive case is reported as the rightmost data point in Figures 6.3(b), 6.4(b,c) and 6.5(b), and on the bottom line of Table 6.1.

6.7.3 Results and discussion

Figure 6.3 and Table 6.1 show the number of comparisons performed by the algorithms (divided by n for clarity) using integer keys. Figures 6.4 and 6.5 give running times for integer and string keys. The number of comparisons in the string keys case was very similar to the larger n of the integer key case.

The figures use the actual number of inversions measured from the input data, averaged* over the 10 generated sequences, as the x-axis. Table 6.1 also gives the parameters k and m of the input-generation algorithms.

The results show that applying bulk insertion greatly improved AVL-tree based sorting when the number of inversions was small: with less than 10^5 inversions, single-insertion sorting was about 6 times slower in the integer case (up to 2 times slower in the string case). In this range, the number of comparisons used by Bulk-insertion sort was very close to $1n$, while single insertion needed about $2n$. The number of rotations (Table 6.1) was also low, as the average bulk size is quite large in this range. For single insertion, the number of rotations is larger for sorted sequences than for random ones; recall that sorted sequences are the worst case for AVL-tree single insertion.

For larger numbers of inversions, Single-insertion sort was slightly faster than Bulk-insertion sort, but the difference was small. Bulk-tree

* The standard deviation was small, less than 1% of the average, except for the cases $m = 2$ to $m = 128$ where it was somewhat larger.

sort used almost the same number of comparisons as Bulk-insertion sort, but the bulk tree was much faster for a medium number of inversions (about 10^4 to $10^2 n$).

Comparing to the other sorting algorithms, we see that for up to about 10^5 inversions (10^4 in the string case), Bulk-insertion sort and Bulk-tree sort were the fastest – except for Insertion sort, which was hopelessly slow when the number of inversions was larger than about $10^2 n$.

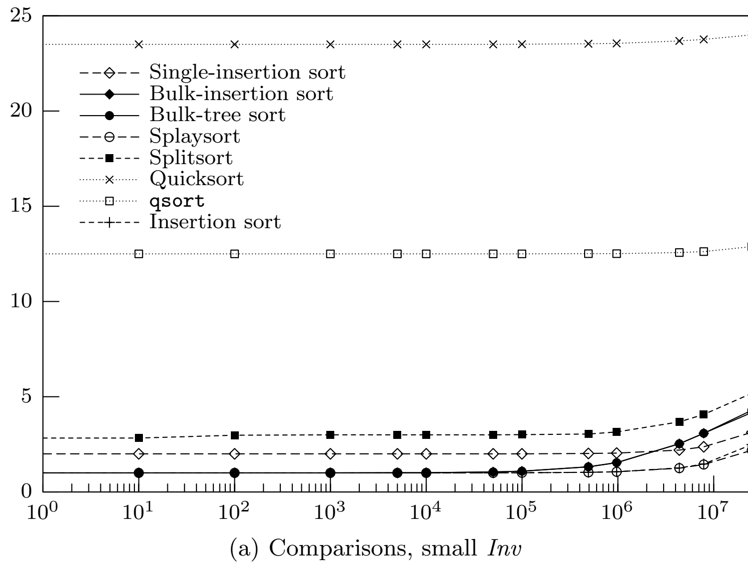
With more than 10^6 inversions, Splaysort, Splitsort and Quicksort were (variously) the fastest. With a very large number of inversions, about $10^6 n$ or more ($10^5 n$ in the string case), Bulk-insertion sort was again faster than Splaysort – though in this range, the array-based Splitsort, Quicksort and `qsort` are much faster. With more than about $10^4 n$ inversions, Bulk-tree sort was significantly slower than Bulk-insertion sort or the others (except for Insertion sort).

Tree-based sorting algorithms are known to be slow for very large numbers of inversions [23,60], and this was also seen here, Figures 6.4(b) and 6.5(b). A similar sharp bend in the running time (at about $10^4 n$ in Figure 6.4(b)) was found in [23] for various tree- and heap-based algorithms; the article gives cache misses and the larger storage requirements of the tree as possible reasons.

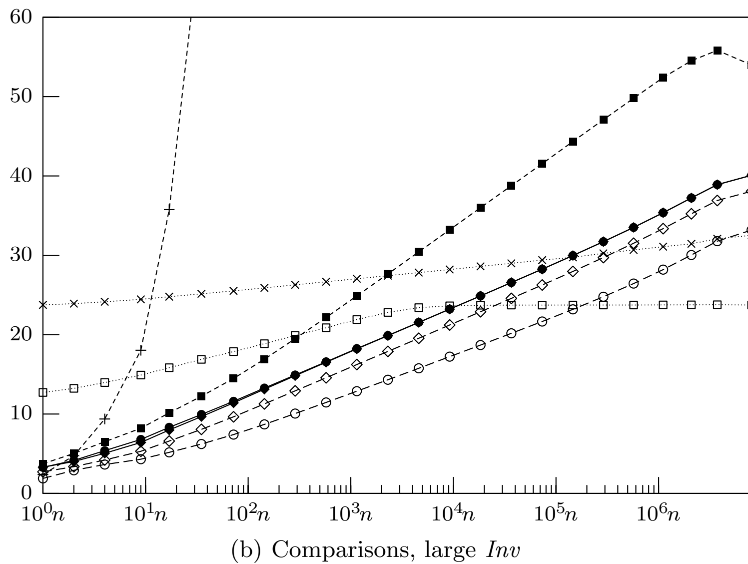
The differences between the algorithms are smaller in the amount of comparisons (Figure 6.3 and Table 6.1). The algorithms that use the least comparisons are Splaysort and (for more than about $10^5 n$ inversions) `qsort` (Merge sort). Up to about 10^5 inversions, Bulk-insertion sort and Bulk-tree sort are as good as Splaysort in the amount of comparisons – and faster in running time, as noted above.

Reasons for the apparent slight *Inv*-adaptivity of Quicksort and `qsort` (Merge sort) are discussed in [13].

These experiments only produced *Inv*-adaptive input data – *Bulk*-adaptivity was not considered. Even adaptivity to the related *Block* measure has apparently never been studied experimentally – of course, neither has the new *Bulk* measure. The current experiments thus actually give an advantage to other methods than Bulk-insertion sort or Bulk-tree sort.

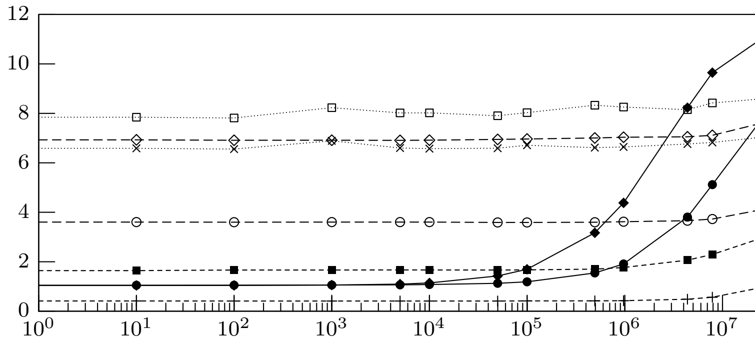


(a) Comparisons, small *Inv*

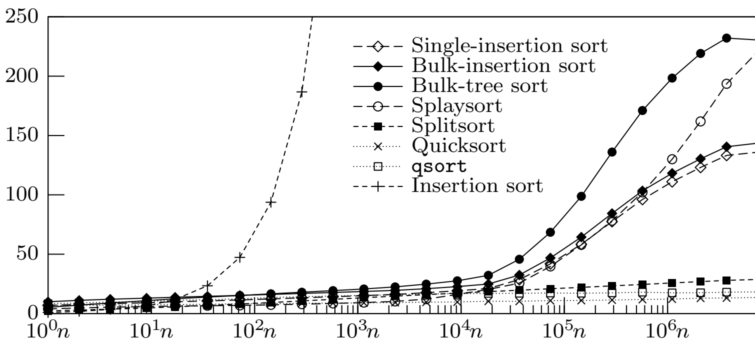


(b) Comparisons, large *Inv*

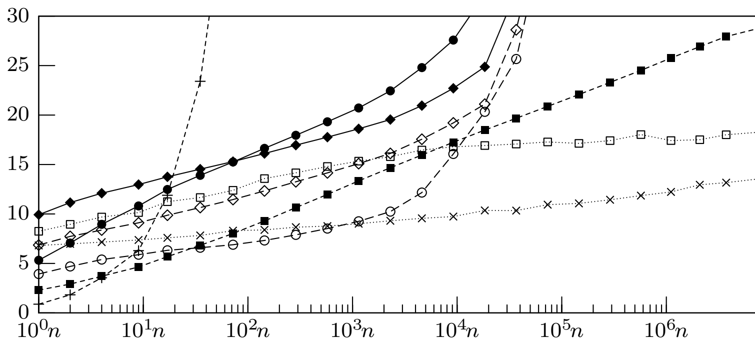
Figure 6.3 Comparisons per element used in sorting, with $n = 2^{25} \approx 34 \cdot 10^6$. The x-axis gives the number of inversions (*Inv*).



(a) Time, small *Inv*

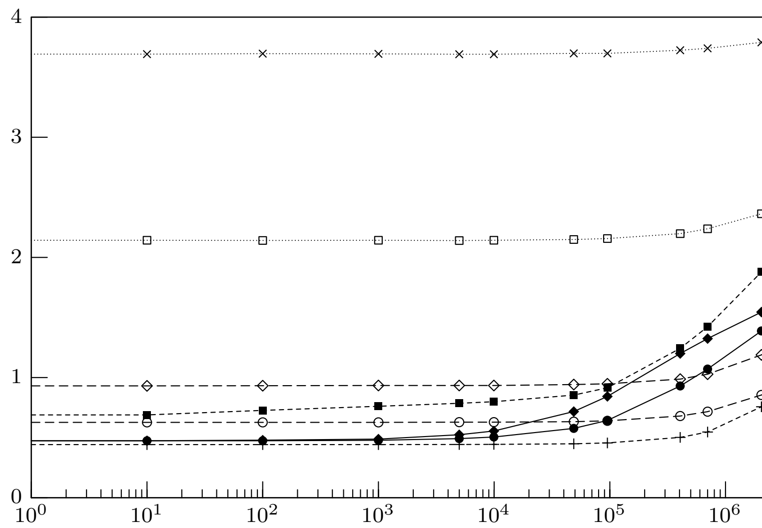


(b) Time, large *Inv*

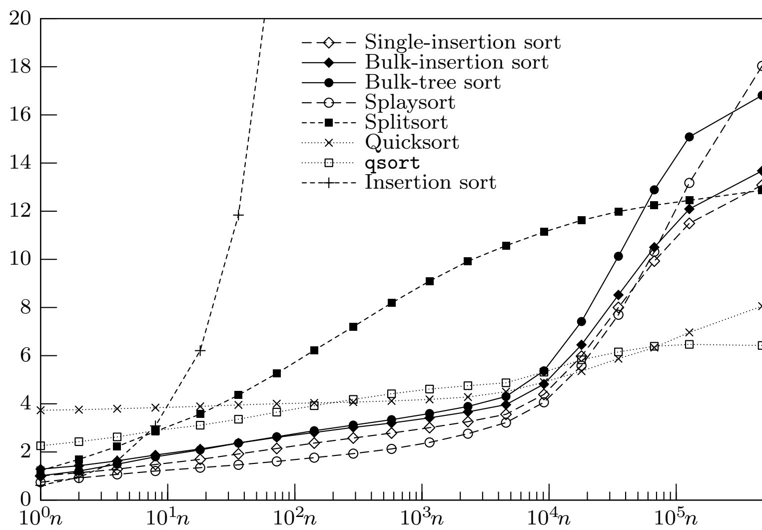


(c) Time, large *Inv* (small times enlarged)

Figure 6.4 Total time (in seconds) used in sorting, with 4-byte integer keys, $n = 2^{25} \approx 34 \cdot 10^6$. The x-axis gives the number of inversions (*Inv*).



(a) Time, small *Inv*



(b) Time, large *Inv*

Figure 6.5 Total time (in seconds) used in sorting, with string keys, $n = 1971968$. The x-axis gives the number of inversions (*Inv*).

	Average inversions	*Comparisons per element* Bulk-tree s.	Bulk-ins. s.	Single-ins. s.	Splay-sort	Split-sort	Insert. sort	Quick-sort	`qsort`
Fully sorted input data (no inversions)									
$k = 0$	0	1.00	1.00	2.00	1.00	1.00	1.00	23.50	12.50
Small amount of inversions									
$k = 10$	10	1.00	1.00	2.00	1.00	2.83	1.00	23.50	12.50
$k = 100$	100	1.00	1.00	2.00	1.00	2.98	1.00	23.50	12.50
$k = 1000$	1000	1.00	1.00	2.00	1.00	3.00	1.00	23.50	12.50
$k = 10000$	9997	1.01	1.01	2.00	1.00	3.00	1.00	23.50	12.50
$k = 100000$	99707	1.09	1.09	2.00	1.01	3.01	1.01	23.51	12.50
$k = 1000000$	971424	1.54	1.54	2.04	1.06	3.15	1.06	23.55	12.51
$k = 10000000$	7875607	3.08	3.08	2.37	1.44	4.07	1.47	23.76	12.62
Large amount of inversions									
$m = 2n$	1	3.28	3.30	2.72	1.89	3.73	2.38	23.75	12.72
$m = 16n$	9	6.80	6.42	5.31	4.31	8.21	18.04	24.45	14.91
$m = 128n$	72	11.60	11.41	9.65	7.41	14.48	144.98	25.52	17.87
$m = 1024n$	575	16.58	16.53	14.57	11.46	22.17	1151.6	26.67	20.88
$m = 8192n$	4600	21.56	21.54	19.55	15.77	30.43	>1152	27.82	23.41
$m = 65536n$	36703	26.57	26.57	24.57	20.17	38.77	>1152	28.99	23.73
$m = 524288n$	289808	31.72	31.72	29.72	24.78	47.09	>1152	30.25	23.74
$m = 4194304n$	2086907	37.23	37.23	35.23	30.04	54.52	>1152	31.45	23.76
Input data in completely random order									
	8064285	40.09	40.09	38.08	33.19	53.98	>1152	32.52	23.74

	Average inversions	*Rotations* Bulk-tree sort	Bulk-ins. sort	Single-ins. sort	*Average bulk size*
Fully sorted input data (no inversions)					
$k = 0$	0	0	0	33554406	33554432
Small amount of inversions					
$k = 10$	10	26	217	33554402	1597830
$k = 100$	100	292	1868	33554357	166937
$k = 1000$	1000	2989	15458	33553905	16770
$k = 10000$	9997	29964	121689	33549410	1679
$k = 100000$	99707	297605	894914	33504640	169.0
$k = 1000000$	971424	2775504	5850949	33079327	18.04
$k = 10000000$	7875607	15612802	22335266	30320503	3.04
Large amount of inversions					
$m = 2n$	1	16778071	25965377	27021115	2.67
$m = 16n$	9	21204866	24485346	22884450	1.36
$m = 128n$	72	17749844	18493838	17962638	1.07
$m = 1024n$	575	16193329	16327473	16219276	1.01
$m = 8192n$	4600	15757134	15778565	15761014	1.00
$m = 65536n$	36703	15654391	15656950	15655066	1.00
$m = 524288n$	289808	15633261	15633548	15633370	1.00
$m = 4194304n$	2086907	15629210	15629568	15629381	1.00
Input data in completely random order					
	8064285	15629022	15629000	15629004	1.00

Table 6.1 A selection of experimental results on sorting (integer keys, $n = 2^{25} \approx 34 \cdot 10^6$). Figure 6.3 has more data points for the number of comparisons.

CHAPTER 7

Conclusion

The concluding remarks in this chapter are divided into three sections according to the main topics of the thesis: cache-sensitive binary search trees, AVL-tree bulk updates, and the application to adaptive sorting.

7.1 Cache-sensitive binary search trees

Chapter 3 examined binary search trees in a k-level cache memory hierarchy with block sizes $B_1, \ldots, B_k$. The chapter presented an algorithm that relocates tree nodes into a multi-level cache-sensitive memory layout in time $O(nk)$, where n is the number of nodes in the tree. However, the main topic was a method for maintaining an improved one-level memory layout for binary search trees by executing a constant-time operation after each structure modification (actual insertion, actual deletion or individual rotation).

As the structure of the tree and rebalancing is not changed, the cache-sensitive methods can be used to improve an existing binary search tree implementation, instead of needing to rewrite it (as, for example, when using cache-oblivious B-trees). More importantly, the methods are applicable to any binary search tree that uses rotations for balancing.

Although the average performance of the cache-sensitive red-black and AVL trees did not quite match the B^+-tree variants in the experiments, in practice there may be other reasons for using binary search trees than average-case efficiency. For instance, the worst-case (as opposed to amortized or average) time complexity of updates in red-black trees is smaller than in B-trees: $O(\log_2 n)$ vs. $O(d \log_d n)$ time for a full sequence of page splits or merges in a d-way B-tree, $d \geq 5$. Rota-

tions are constant-time operations, unlike B-tree node splits or merges, which take $O(B_1)$ time in B-trees with B_1-sized nodes, or $O(B_1^2)$ in the full CSB$^+$-tree. This may improve concurrency: nodes are locked for a shorter duration. In addition, it has been argued in [76] that, in main-memory databases, binary trees are optimal for a form of shadow paging that allows efficient crash recovery and transaction rollback, as well as the group commit operation [32].

A natural extension of the simple invariant of the local algorithm of Section 3.4 would be to handle multi-level caches in some way. However, the local algorithm was designed to preserve the property of binary search trees that individual structure modifications use worst-case constant time. Multi-level cache sensitivity does not seem feasible with this constraint because, for instance, it is not possible to move a cache-block-sized area of nodes to establish an invariant after a structure modification. A direction for future research could be to lift this constraint to create a dynamic multi-level cache-sensitive search tree (based on either a B-tree or a binary search tree) using the technique of preserving a memory-layout invariant while keeping the structure of the nodes and rebalancing strategy otherwise intact.

The aliasing phenomenon reported in Section 3.3.2 presumably also affects other multi-level cache-sensitive search trees. It would be interesting to see the effect of a similar aliasing correction on, for example, the two-level cache-sensitive B-trees of [17].

One way of using the global relocation algorithm is to run it periodically in a background process. The experiments in Section 3.5 indicate that it does not need to be run very often – it appears that a viable approach would be to execute the global algorithm after nk^2 updates have been performed, where n is the number of keys in the tree when the global algorithm was last run. This would bring the amortized cost of the global algorithm down to $O(1)$ per update. These periodic runs could possibly also be combined with relaxed balancing (Section 4.3) to decouple rotations and memory layout updates from the actual insertion and deletion operations.

A limitation of the cache-sensitive approaches of this thesis is that they do not optimize the "scan" or range search operation, which reads a range of consecutive keys in the tree. The current approaches may need to read $O(S)$ cache blocks to perform a scan of S keys, while, for example, the cache-sensitive and some of the cache-oblivious B-trees need only $O(S/B)$ blocks for this operation, where B is a cache block size.

7.2 Bulk updates

Chapter 5 presented and analyzed bulk insertion and deletion algorithms for AVL trees.

When all of the keys to be inserted go to the same location in the tree, the number of rotations performed by the bulk-insertion algorithm was shown to be worst-case logarithmic in the number of inserted keys (even if the number of keys already present in the tree is much larger). The total rebalancing complexity was amortized logarithmic in a sequence of bulk insertions.

In the case where the inserted keys go to multiple locations, the thesis presented an algorithm that improves upon the tree search cost of a simple approach that executes a separate bulk insertion for each location.

Experiments demonstrated that the bulk-insertion algorithm is very efficient in practice, both in the number of rotations and in running time, although the latter is dominated by the creation of new nodes.

The rebalancing complexity of the bulk-deletion algorithm (as also the number of rotations) was shown to be logarithmic in the number of deleted nodes, when amortized over a sequence of bulk deletions. Moreover, the bulk-deletion algorithm was able to avoid visiting most of the nodes to be deleted by detaching entire subtrees from the tree using constant-time operations. This made the algorithm extremely fast in the experiments reported in the chapter.

The amortized complexities noted above do not apply to a sequence of mixed bulk insertions and deletions, because the balancing criterion in AVL trees is such that even mixed single insertions and deletions can require $O(\log n)$ rotations in the worst case, where n is the size of the existing tree. Although this worst case is rare in practice, AVL trees are most at home in a situation where most operations are searches and insertions.

7.3 Application to adaptive sorting

The main objective of Chapter 6 was to show that bulk insertion can successfully be applied to the problem of adaptive sorting.

The chapter presented a simple comparison-efficient finger structure for binary trees based on saving the search path in an auxiliary

data structure, thus creating an AVL-tree-based variation of the adaptive sorting algorithm known as local insertion sort.

A strategy was given for finding bulks to be inserted using only $m + O(\log m)$ comparisons per bulk of size m. This was then applied to both an ordinary AVL tree using the bulk-insertion algorithm, and to a specialized tree called the bulk tree, where the inserted bulks were stored as intervals.

Using the simplified finger structure, with or without applying bulk insertion, was shown to create an inversion-optimal sorting algorithm. In addition, the new algorithms bulk-insertion sort and bulk-tree sort were shown to be optimal with respect to a new measure *Bulk* that counts the number of produced bulks.

The experiments demonstrated that for small amounts of inversions – up to about 10^5 in both integer and string key inputs of length about 10^7 – the bulk-insertion-based algorithms were better than all other measured sorting algorithms (Splaysort, Splitsort, Quicksort, `qsort`), with the exception of Insertion sort, which has quadratic worst-case time bound. Despite the space overhead implied by tree sorting (see [60]), the algorithms are thus a reasonable choice for inputs most of which are known to be nearly sorted.

Even though the focus was on minimizing the number of comparisons, the experiments demonstrated that the new sorting algorithms are quite efficient also in terms of running time, even when comparisons are fast (e.g., when sorting simple integers). For nearly sorted sequences, the bulk-insertion sort and bulk-tree sort algorithms came very close to reaching the lower bound of one comparison per key to be sorted.

Besides sorting, a potential application of the bulk-insertion sort algorithm of Section 6.3 is that of index generation in a main memory database: the algorithm provides a very efficient method for generating an AVL tree from a set of nearly sorted keys.

The comparison-efficient bulk-finding strategy relied on the restriction that the sequences of consecutive keys inserted as bulks must be in ascending or descending order. An alternative could be to collect larger bulks containing consecutive keys that go to the same location in the tree but are not in order, and sort them recursively to produce an update tree. Although Chapter 5 specified that bulk insertion should first sort its input and create the update tree from sorted data, the bulk-insertion algorithm actually works with any AVL tree as the update tree. It would thus be possible to use the tree resulting from a

recursive invocation of bulk-insertion sort as the update tree. However, this approach would be incompatible with the use of lazy bulks.

The way in which bulk insertion was applied to local insertion sort is not specific to AVL trees. It would be possible to, for instance, to apply an (a, b)-tree bulk-insertion algorithm [46, 50, 66, 79] to local insertion sort using (a, b) trees, either with a saved path-type finger or using a full finger tree. However, the comparison-efficient simplified finger structure of Section 6.1 is only applicable to binary trees.

References

[1] G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics*, 3:1259–1263, 1962. (English translation.).

[2] L. Arge, K. H. Hinrics, J. Vahrenhold and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33:104–128, 2002.

[3] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 22(6):1488–1508, 2003.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[5] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono and A. López-Ortiz. The cost of cache-oblivious searching. In *44th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003)*, pages 271–282. IEEE Computer Society, 2003.

[6] M. A. Bender, E. D. Demaine and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 165–173. Springer, 2002.

[7] M. A. Bender, E. D. Demaine and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[8] M. A. Bender, Z. Duan, J. Iacono and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *13th Annual ACM-SIAM*

Symposium on Discrete Algorithms (SODA 2002), pages 29–38. Society for Industrial and Applied Mathematics, 2002.

[9] M. A. Bender, M. Farach-Colton and B. C. Kuszmaul. Cache-oblivious string B-trees. In *25th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS 2006)*, pages 233–242. ACM Press, 2006.

[10] G. E. Blelloch, B. M. Maggs and S. L. M. Woo. Space-efficient finger search on degree-balanced search trees. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 374–383. ACM Press, 2003.

[11] P. Bohannon, P. McIlroy and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *2001 ACM SIGMOD International Conference on Management of Data*, pages 163–174. ACM Press, 2001.

[12] G. S. Brodal, R. Fagerberg and R. Jacob. Cache oblivious search trees via binary trees of small height. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 39–48. Society for Industrial and Applied Mathematics, 2002.

[13] G. S. Brodal, R. Fagerberg and G. Moruz. On the adaptiveness of quicksort. In *7th Workshop on Algorithm Engineering and Experiments (ALENEX 2005)*, pages 130–140. Society for Industrial and Applied Mathematics, 2005.

[14] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[15] S. Carlsson, C. Levcopoulos and O. Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9:629–648, 1993.

[16] S. Chen, P. B. Gibbons and T. C. Mowry. Improving index performance through prefetching. In *2001 ACM SIGMOD International Conference on Management of Data*, pages 235–246. ACM Press, 2001.

[17] S. Chen, P. B. Gibbons, T. C. Mowry and G. Valentin. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *2002 ACM SIGMOD International Conference on Management of Data*, pages 157–168. ACM Press, 2002.

[18] R. Cole. On the dynamic finger conjecture for splay trees, part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

[19] R. Cole, B. Mishra, J. Schmidt and A. Siegel. On the dynamic finger conjecture for splay trees, part I: Splay sorting $\log n$-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

[20] C. R. Cook and D. J. Kim. Best sorting algorithm for nearly sorted lists. *Communications of the ACM*, 23(11):620–624, 1980.

[21] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411. ACM Press, 1989.

[22] A. Elmasry. Adaptive sorting with AVL trees. In *IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (IFIP TCS 2004)*, pages 307–316. Kluwer, 2004.

[23] A. Elmasry and A. Hammad. An empirical study for inversions-sensitive sorting algorithms. In *4th International Workshop of Efficient and Experimental Algorithms (WEA 2005)*, volume 3503 of *Lecture Notes in Computer Science*, pages 597–601. Springer, 2005.

[24] A. Elmasry and A. Hammad. Inversion-sensitive sorting algorithms in practice. *ACM Journal of Experimental Algorithmics*, 13(1.11), 2008.

[25] V. Estivill-Castro. Generating nearly sorted sequences – the use of measures of disorder. *Electronic Notes in Theoretical Computer Science*, 91:56–95, 2004.

[26] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.

[27] R. Fenk, A. Kawakami, W. Markl, R. Bayer and S. Osaki. Bulk loading a data warehouse built upon a UB-tree. In *International Database Engineering and Applications Symposium (IDEAS 2000)*, pages 179–187. IEEE Computer Society, 2000.

[28] J. D. Fix. The set-associative cache performance of search trees. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 565–572. ACM Press, 2003.

[29] C. C. Foster. Information storage and retrieval using AVL trees. In *Proceedings of the 1965 20th National Conference*, pages 192–205. ACM Press, 1965.

[30] C. C. Foster. A generalization of AVL trees. *Communications of the ACM*, 16(8):513–517, 1973.

[31] J. Gil and A. Itai. How to pack trees. *Journal of Algorithms*, 32:108–132, 1999.

[32] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[33] L. J. Guibas, E. M. McCreight, M. F. Plass and J. R. Roberts. A new representation for linear lists. In *9th Annual ACM Symposium on Theory of Computing (STOC 1977)*, pages 49–60. ACM Press, 1977.

[34] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, pages 8–21. IEEE Computer Society, 1978.

[35] S. Hanke, T. Ottmann and E. Soisalon-Soininen. Relaxed balanced red-black trees. In *3rd Italian Conference on Algorithms and Complexity (CIAC 1997)*, volume 1203 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 1997.

[36] S. Hanke and E. Soisalon-Soininen. Group updates for red-black trees. In *4th Italian Conference on Algorithms and Complexity (CIAC 2000)*, volume 1767 of *Lecture Notes in Computer Science*, pages 253–262. Springer, 2000.

[37] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B+-trees. In *2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 283–294. ACM Press, 2003.

[38] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[39] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[40] C. Jermaine, A. Datta and E. Omiecinski. A novel index supporting high volume data warehouse insertion. In *25th International Conference on Very Large Data Bases (VLDB 1999)*, pages 235–246. Morgan Kaufmann, 1999.

[41] W. Jiang, C. Ding and R. Cheng. Memory access analysis and optimization approaches on splay trees. In *7th Workshop on Languages, Compilers and Run-time Support for Scalable Systems*, pages 1–6. ACM Press, 2004.

[42] J. Katajainen, C. Levcopoulos and O. Petersson. Local insertion sort revisited. In *International Symposium on Optimal Algorithms*, volume 401 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 1989.

[43] D. E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching.* Addison-Wesley, second edition, 1998.

[44] S. R. Kosaraju. Localized search in sorted lists. In *13th Annual ACM Symposium on Theory of Computing (STOC 1981)*, pages 62–69. ACM Press, 1981.

[45] T.-W. Kuo, C.-H. Wei and K.-Y. Lam. Real-time data access control on B-tree index structures. In *15th International Conference on Data Engineering (ICDE 1999)*, pages 458–467. IEEE Computer Society, 1999.

[46] S. D. Lang, J. R. Driscoll and J. H. Jou. Batch insertion for tree structured file organizations—improving differential database representation. *Information Systems*, 11(2):167–175, 1986.

[47] K. S. Larsen. Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica*, 35:859–874, 1998.

[48] K. S. Larsen. AVL trees with relaxed balance. *Journal of Computer and System Sciences*, 61:508–522, 2000.

[49] K. S. Larsen. Relaxed red-black trees with group updates. *Acta Informatica*, 38:565–586, 2002.

[50] K. S. Larsen. Relaxed multi-way trees with group updates. *Journal of Computer and System Sciences*, 66:657–670, 2003.

[51] K. S. Larsen, T. Ottmann and E. Soisalon-Soininen. Relaxed balance for search trees with local rebalancing. *Acta Informatica*, 37:743–763, 2001.

[52] K. S. Larsen, E. Soisalon-Soininen and P. Widmayer. Relaxed balance using standard rotations. *Algorithmica*, 31:501–512, 2001.

[53] C. Levcopoulos and O. Petersson. Splitsort – an adaptive sorting algorithm. *Information Processing Letters*, 39:205–211, 1991.

[54] T. Lilja, R. Saikkonen, S. Sippu and E. Soisalon-Soininen. Online bulk deletion. In *23rd International Conference on Data Engineering (ICDE 2007)*, pages 956–965. IEEE Computer Society, 2007.

[55] L. Malmi and E. Soisalon-Soininen. Group updates for relaxed height-balanced trees. In *18th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS 1999)*, pages 358–367. ACM Press, 1999.

[56] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34:318–325, 1985.

[57] K. Mehlhorn. Sorting presorted files. In *4th GI-Conference on Theoretical Computer Science*, volume 67 of *Lecture Notes in Computer Science*, pages 199–212. Springer, 1979.

[58] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer, 1984.

[59] K. Mehlhorn and A. Tsakalidis. An amortized analysis of insertions into AVL-trees. *SIAM Journal of Computing*, 15(1):22–33, 1986.

[60] A. Moffat, G. Eddy and O. Petersson. Splaysort: Fast, versatile, practical. *Software, Practice and Experience*, 126(7):781–797, 1996.

[61] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *16th International Conference on Very Large Data Bases (VLDB 1990)*, pages 392–405. Morgan Kaufmann, 1990.

[62] O. Nurmi, E. Soisalon-Soininen and D. Wood. Concurrency control in database structures with relaxed balance. In *6th ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS 1987)*, pages 170–176. ACM Press, 1987.

[63] K. Oksanen. *Memory Reference Locality in Binary Search Trees.* Master's thesis, Helsinki University of Technology, 1995.

[64] K. Oksanen and L. Malmi. Memory reference locality and periodic relocation in main memory search trees. In *5th Hellenic Conference of Informatics*. Greek Computer Society, 1995.

[65] O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, 1995.

[66] K. Pollari-Malmi. *Batch Updates and Concurrency Control in B-trees.* Ph.D. thesis, Helsinki University of Technology, 2002.

[67] K. Pollari-Malmi, J. Ruuth and E. Soisalon-Soininen. Concurrency control for B-trees with differential indices. In *International Database Engineering and Applications Symposium (IDEAS 2000)*, pages 287–295. IEEE Computer Society, 2000.

[68] K. Pollari-Malmi, E. Soisalon-Soininen and T. Ylönen. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):975–984, 1996.

[69] N. Rahman, R. Cole and R. Raman. Optimised predecessor data structures for internal memory. In *5th Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2001.

[70] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *25th International Conference on Very Large Data Bases (VLDB 1999)*, pages 78–89. Morgan Kaufmann, 1999.

[71] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486. ACM Press, 2000.

[72] R. Saikkonen. *Group Insertion in AVL Trees.* Master's thesis, Helsinki University of Technology, 2004.

[73] R. Saikkonen and E. Soisalon-Soininen. Cache-sensitive memory layout for binary trees. In *5th IFIP International Conference on Theoretical Computer Science (IFIP TCS 2008)*, volume 273 of *IFIP International Federation for Information Processing*, pages 241–255. Springer, 2008.

[74] R. Saikkonen and E. Soisalon-Soininen. Bulk-insertion sort: Towards composite measures of presortedness. In *8th International Symposium on Experimental Algorithms (SEA 2009)*, volume 5526

of *Lecture Notes in Computer Science*, pages 269–280. Springer, 2009.

[75] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[76] E. Soisalon-Soininen and P. Widmayer. Concurrency and recovery in full-text indexing. In *String Processing and Information Retrieval Symposium (SPIRE 1999)*, pages 192–198. IEEE Computer Society, 1999.

[77] E. Soisalon-Soininen and P. Widmayer. Amortized complexity of bulk updates in AVL-trees. In *8th Scandinavian Workshop on Algorithm Theory (SWAT 2002)*, volume 2368 of *Lecture Notes in Computer Science*, pages 439–448. Springer, 2002.

[78] E. Soisalon-Soininen and P. Widmayer. Single and bulk updates in stratified trees: An amortized and worst-case analysis. In *Computer Science in Perspective: Essays Dedicated to Thomas Ottmann*, volume 2598 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2003.

[79] J. Srivastava and C. V. Ramamoorthy. Efficient algorithms for maintenance of large database indexes. In *4th International Conference on Data Engineering (ICDE 1988)*, pages 402–408. IEEE Computer Society, 1988.

[80] R. E. Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Information Processing Letters*, 16:253–257, 1983.

[81] R. E. Tarjan. Amortized computational complexity. *SIAM Journal of Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[82] A. K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67:173–194, 1985.

[83] A. K. Tsakalidis. Rebalancing operations for deletions in AVL-trees. *RAIRO Informatique Théorique et Applications*, 19(4):323–329, 1985.

[84] S. Virtanen. *Group Update and Relaxed Balance in Search Trees.* Master's thesis, Helsinki University of Technology, 2000.