Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Cache-Oblivious Data Structures for Massive Data Sets

A Dissertation Presented

by

Haodong Hu

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2007

Stony Brook University
The Graduate School

Haodong Hu

We, the dissertation committee for the above candidate for the degree of Doctor of Philosophy, hereby recommend acceptance of this dissertation.

Michael A. Bender, Dissertation Advisor
Associate Professor, Computer Science Department

Joseph S.B. Mitchell, Chairperson of Defense
Professor, Computer Science Department and
Applied Mathematics and Statistics Department

Martin Farach-Colton
Professor, Computer Science Department
at Rutgers University

Xianfeng David Gu
Assistant Professor, Computer Science Department

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Cache-Oblivious Data Structures for Massive Data Sets

by

Haodong Hu

Doctor of Philosophy

in

Computer Science

Stony Brook University

2007

The ***cache-oblivious model*** [39, 50] is one of the most successful models of a memory hierarchy. The cache-oblivious model allows programmers to reason about a simple two-level memory model without knowing any memory parameters, but to prove results about a multilevel memory model. Thus, algorithms and structures based on the cache-oblivious model have the advantage of platform independence and simultaneously optimal on all levels of a memory hierarchy. The ***disk-access model (DAM)*** [4], another successful memory model, assumes a two-level memory model with the full knowledge of memory parameters. Like the DAM model, the performance of the cache-oblivious model is measured by ***memory*** (or ***block***) transfers between two adjacent memory levels with block size B.

In this dissertation, we build highly efficient, optimized cache-oblivious structures in support of cache-oblivious B-trees and other dictionaries. We improve two common fundamental cache-oblivious structures: a static cache-oblivious search tree in ***van Emde Boas layout (vEB)*** [59, 60] and ***packed-memory array (PMA)*** [16] for dynamically maintaining sorted elements in memory or on disk. Specifically, the vEB supports search asymptotically optimally in $O(1+\log_B N)$ memory transfers. The PMA supports the operations insert/delete in an array of size N in $O(1+(\log^2 N)/B)$ amortized memory transfers and range query of L consecutive elements optimally in $O(1+L/B)$ memory transfers.

The vEB and PMA are used as basic building blocks in many cache-oblivious B-trees and dictionaries; see e.g., [1, 13–19, 21, 24, 27, 28, 30, 51, 52]. One of the ***dynamic cache-oblivious B-trees***, proposed by Bender, Duan, Iacono, and Wu [18], combines both the vEB as the top search tree and the PMA as the bottom dynamic structure. Specifically, this dynamic B-tree supports search in $O(1+\log_B N)$ memory transfers, deletion and insertion in $O(1+\log_B N+(\log^2 N)/B)$ amortized memory transfers, and scans of L consecutive elements optimally in $O(1+\log_B N+L/B)$ memory transfers. Thus, any improvements to the vEB layout and the packed-memory array immediately translate to improvements to cache-oblivious B-trees and other dictionaries.

In this dissertation, we present the following results:

We prove tight bounds on the cost of cache-oblivious searching and propose a ***generalized van Emde Boas layout*** to optimize the searching cost in the cache-oblivious structures. we show that there is no cache-oblivious search structure can guarantee that a search performs fewer than $\lg e \log_B N$ memory transfers, i.e., $\lg e \log_B N$ is the lower bound. The upper bound is achieved by the generalized vEB layout, whose expected memory transfers between any two levels of the memory hierarchy arbitrarily close to $[\,\lg e + O(\lg\lg B/\lg B)\,]\log_B N + O(1)$. That is, we achieve the factor, which optimally approaches $\lg e \approx 1.443$ as the block size B increases. The work appears in Chapter 2 and is published in [12].

We give the first ***adaptive packed-memory array (APMA)***, theoretically and practically optimizing PMA's performance on most common input patterns. Like the traditional PMA, any pattern of updates costs only $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ amortized memory transfers per update. However, the adaptive PMA adjusts to the input pattern and therefore performs better on many common input distributions achieving only $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ amortized memory transfers. We analyze sequential inserts, where the insertions are to the front of the APMA, hammer inserts, where the insertions "hammer" on one part of the APMA, random inserts, where the insertions are after random elements in the APMA, and bulk inserts, where for constant $\alpha \in [0,1]$, N^α elements are inserted after random elements in the APMA. This work appears in Chapter 3 and is published in [25].

We develop the ***partially deamortized packed-memory array (PDPMA)*** to reduce the worst-case cost in the PMA. As the traditional PMA, the PDPMA has the same update cost of $O(1+(\log^2 N)/B)$ amortized memory transfers. However, for a single update, the traditional PMA has the worst-case cost of $O(N)$ element moves and $O(1+N/B)$ memory transfers. It is not feasible for industrial application because one insertion might trigger the rebalance of the whole database. Therefore, our partially deamortized PMA is designed for the purpose of cost reduction in the worst case and achieves the worst-case performace of one update in $O(\sqrt{N}\log N)$ element moves and $O(1+(\sqrt{N}\log N)/B)$ memory transfers. This work appears in Chapter 4.

We present the first ***atomic-key B-tree*** to support atomic keys of different sizes in the B-tree with the theoretical guarantee. There exist many practical B-trees in support of variable-length keys. However, none of them have theoretical guarantee and their practical performance degrades when keys are long or vary in length. Our atomic-key B-tree is efficient in this respect. Specifically, we first give an algorithm for building a static atomic-key B-tree. On a dictionary of n keys having average size $\hat{k}$, the expected cost to search for a random key is $O(\lceil \hat{k}/B \rceil \log_{1+\lceil B/\hat{k} \rceil} N)$ memory transfers, under the assumption that all keys are searched with uniform probability. The cost to build this tree is $O(N)$ operations and $O(N+N\hat{k}/B)$ memory transfers. We then show how to build a dynamic atomic-key B-tree with a better performance of $O(N/f+N\hat{k}/B)$ memory transfers, where $f=\max\{2, \lceil B/\hat{k} \rceil\}$. In this dynamic structure, the expect cost to search for random keys stays the same. The cost to insert an arbitrary key κ is the cost to search for κ plus a tree-update cost of $O(\lceil \hat{k}/B \rceil \log_{1+\lceil B/\hat{k} \rceil} N)$ amortized memory transfers. This work appears in Chapter 5.

In summary, we contribute to the theory of the searching cost in the cache-oblivious model by presenting the generalized vEB layout. We develop two dynamic cache-oblivious structures: the adaptive PMA and the partially deamortized PMA to overcome the traditional PMA's deficiencies and make it practical. We also propose the atomic-key B-tree in support of atomic keys with variable length, whose structure is as close as possible to the traditional B-tree while having performance guarantees.

For my parents.

Contents

List of Algorithms **ix**

List of Figures **x**

Acknowledgements **xiv**

1 Introduction **1**
- 1.1 Overview 1
- 1.2 Results 6

2 Cache-Oblivious Searching Cost **11**
- 2.1 Lower Bound for Cache-Oblivious Searching 12
- 2.2 Upper Bound for van Emde Boas Layout 19
- 2.3 Upper Bound for the Generalized van Emde Boas Layout 23
- 2.4 Conclusion 52

3 Adaptive Packed-Memory Array **54**
- 3.1 Structures and Algorithms for Adaptive PMA 55
- 3.2 Analysis of Sequential and Hammer Insertions 65
- 3.3 Analysis for Random and Bulk Insertions 88
- 3.4 Experimental Results 96
- 3.5 Conclusion 101

4 Partially Deamortized Packed-Memory Array **102**
- 4.1 One-Phase Rebalance in PMA 103
- 4.2 Description of Partially Deamortized PMA 108

4.3 Conclusion 114

5 Atomic-key B-tree **115**

5.1 Static Structure 116

5.2 Dynamic Strcuture 126

5.3 Dynamic Structure Using Indirection 133

5.4 Optimal Static Structure by Dynamic Programming 142

5.5 Conclusion 144

Bibliography **145**

List of Algorithms

1 Predictor.insert(x) 60
2 Rebalance.uneven(u_ℓ) 63
3 Rebalance.leftward.interval(leftbound) 105
4 Rebalance.rightward.interval(rightbound) 105
5 rightbound.window(S_0) 107

List of Figures

1 The predictor. Each cell contains a marker element x, the leaf node in the APMA where x resides, and the count number $I(x)$. 61

2 In the simple case, the shaded region is rebalanced just after Phase 1 of node u_ℓ, which starts from Density$(u_{\ell-2}) = 0$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). 66

3 In the simple case, the shaded region is rebalanced just after Phase 2 of node u_ℓ, which starts from Density$(u_{\ell-2}) = \tau_{\ell-2} - \tau_{\ell-1}$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). 67

4 Phase 1 of node u_ℓ starts from Density$(u_{\ell-2}) = 0$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced. . . . 69

5 Phase 2 of node u_ℓ starts from Density$(u_{\ell-2}) = \rho_{\ell-1}$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced. . . 69

6 Phase 3 of node u_ℓ starts from Density$(u_{\ell-2}) = 2\tau_{\ell-2} - \tau_{\ell-1} - \rho_{\ell-1}$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced. 70

7 Subphase 1 starts from Density$(u_{\ell-3}) = \rho_{\ell-2}$ (left) and ends at Density$(u_{\ell-3}) = \tau_{\ell-3}$ (right). The shaded region is rebalanced. . . . 72

8 Subphase 2 starts from Density$(u_{\ell-3}) \geq \rho_{\ell-2}$ (left) and ends at Density$(u_{\ell-3}) = \tau_{\ell-3}$ (right). The shaded region is rebalanced. . . . 73

9 Phase 1 of u_i starts from Density$(u_{i-1}) = \rho_i$ (left) and ends at Density$(u_{i-1}) = \tau_{i-1}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of the phase is shaded. 77

10 Phase 2 of u_i starts from Density$(u_{i-1}) = 2\rho_{i+1}$ (left) and ends at Density$(u_{i-1}) = \tau_{i-1}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of the phase is shaded. 78
11 Phase 3 of u_i starts from Density$(u_{i-1}) = \tau_{i-1} - \rho_{i-1}$ (left) and ends at Density$(u_{i-1}) = \tau_{i-1}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of the phase is shaded. 80
12 The tail-insert stage of u_{i-2} starts from Density$(u_{i-4}) \geq \rho_{i+1}$ (left) and ends at Density$(u_{i-4}) = \tau_{i-4}$ (right). The marker element x is indicated by a black dot. At the end of the tail-insert stage of u_{i-2}, node u_{i-1} is rebalanced. 81
13 Subphase 1 of Phase 3 starts from Density$(u_{i-2}) = 2\tau_{i-1} - 2\rho_{i-1} - 4\rho_{i+1}$ (left) and ends at Density$(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of Subphase 1 is shaded. 83
14 Subphase 2 of Phase 3 starts from Density$(u_{i-2}) = 4\rho_{i+1} + \tau_{i-2} - \tau_{i-1}$ (left) and ends at Density$(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. At the end of Subphase 2, node u_i, the parent of u_{i-1}, is rebalanced. 84
15 Subphase 1 of Phase 2 starts from Density$(u_{i-2}) = \rho_{i-1}$ (left) and ends at Density$(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of Subphase 1 is shaded. 85
16 Subphase 2 of Phase 2 starts from Density$(u_{i-2}) = \tau_{i-2} - \rho_{i-1}$ (left) and ends at Density$(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of Subphase 2 is shaded. 86
17 Subphase 3 of Phase 2 starts from Density$(u_{i-2}) = 4\rho_{i+1} + \tau_{i-2} - \tau_{i-1}$ (left) and ends at Density$(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. At the end of Subphase 3, node u_i is rebalanced. 86
18 An illustration showing the tree divided at height $\lceil \alpha \log N \rceil$. 89

19 Phase 1 of node u_ℓ starts from Density$(u_{\ell-2}) = \tau_{\ell-1}$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced. . . 91
20 Phase 2 of node u_ℓ starts from Density$(u_{\ell-2}) = \tau_\ell$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced. . . . 91
21 The densities of node u_ℓ's descendants at the beginning of Phase 2 of node u_ℓ. 92
22 Sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted. 97
23 Sequential inserts: the running time to insert up to 1.4 million elements. 97
24 Random inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted. 98
25 Random inserts: the running time to insert up to 1.4 million elements. 98
26 Bulk inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted. . . . 99
27 Bulk inserts: the running time to insert up to 1.4 million elements. . 99
28 Multiple sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted. 100
29 Multiple sequential inserts: the running time to insert up to 1.4 million elements. 100
30 Half random, half sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted. 100
31 Half random, half sequential inserts: the running time to insert up to 1.4 million elements. 100

32 The pattern of one-phase rebalance. 104
33 Single leftward intervals. 104
34 PDPMA model. 109

35 The greedy algorithm for the root node of a static tree layout. 117
36 The greedy algorithm for the root node of a dynamic tree layout. . . 126
37 The first set S_1 includes C_0 and part of C_1. 130

38 The set S_i includes the right part of C_{i-1} and the left part of C_i. 130
39 The bottom layer structure, including type-I groups and type-II groups. 135
40 The top layer structure: a greedy tree layout based on the top bottom layer. 138
41 The top tree layout has the leaf nodes trimmed. 140
42 The optimal structure by the dynamic programming. 143

Acknowledgements

Foremost, I am deeply grateful to my supervisor, professor Michael A. Bender, who guides me through research and shares with me a lot of his expertise. Under his advise, I have gained the research insight and became a successful researcher.

I express my gratitude to the professors who served on my committees: professor Martin Farach-Colton from Rutgers University and professors Jie Gao, Xianfeng David Gu, Joseph Mitchell, and Steve Skiena at Stony Brook University. Their thoughtful advices are great help for completing my dissertation.

I would like to thank the coauthors in my published papers: professors Gerth Stølting Brodal at University of Aarhus, Rolf Fagerberg at University of Southern Denmark, John Iacono at Polytechnic University, Alejandro López-Ortiz at University of Waterloo, Ron Y. Pinter and Firas Swidan at Israel Institute of Technology, Steve Skiena at Stony Brook University, and my friends Simai He at Chinese University of HongKong and Dongdong Ge at Stanford University. It was both fun and productive to work with them.

I appreciate faculty and Staff in the Computer Science Department at Stony Brook University. Their kindness and help are unforgettable things during my time at Stony Brook.

I thank all friends including Ziyang Duan, Yonatan Fogel, Xin Li, Marc Tchiboukdjian, Yue Wang, and professor Ker-I Ko, with whom I discussed and gained interest in research.

Finally, I wish to thank my parents. They have always supported and encouraged me to do my best in all matters of life. To them I dedicate this thesis.

Chapter 1

Introduction

1.1 Overview

Hierarchical Memory Models. Traditionally, algorithms were designed to run efficiently in a *random access model (RAM)* of computation, which assumes a flat memory with uniform access times. However, as hierarchical memory systems become steeper and more complicated, algorithms are increasingly designed assuming more accurate memory models; see e.g., [2–7, 9, 42, 53–55, 61–63]. Two of the most successful memory models are the ***disk-access model (DAM)*** and the ***cache-oblivious model***.

The DAM model, developed by Aggarwal and Vitter [4], is a two-level memory model, in which the memory hierarchy consists of an internal memory of size M and an arbitrarily large external memory partitioned into blocks of size B. Algorithms are designed in the DAM model with full knowledge of the values of B and M. Because memory transfers are relatively slow, the performance metric is the number of memory transfers. The main disadvantage of a two-level memory model is that the programmer must focus efforts on a particular level of a given hierarchy, resulting in programs that are not portable and suited for use on a modern multilevel hierarchy.

The cache-oblivious model, developed by Frigo, Leiserson, Prokop, and Ramachandran [39, 50], allows programmers to reason about a two-level memory hierarchy but to prove results about an unknown multilevel memory hierarchy. As

in the DAM model, the objective is to minimize the number of memory transfers between two levels. The main idea of the cache-oblivious model is that by avoiding any memory-specific parametrization (such as the block sizes) the cache-oblivious algorithm has an asymptotically optimal number of memory transfers between all levels of an unknown, multilevel memory hierarchy.

The theory of cache-oblivious algorithms is based on the ideal-cache model of Frigo, Leiserson, Prokop and Ramachandran [39, 50], which assumes both optimal page replacement strategy and fully associative cache. While this model may superficially seem unrealistic, Frigo et al. show that it can be simulated essentially by any memory system with a small constant-factor overhead. Thus, if we run a cache-oblivious algorithm on a multilevel memory hierarchy, we can use the ideal-cache model to analyze the number of memory transfers between each pair of adjacent levels.

Optimal cache-oblivious algorithms have memory performance (i.e., number of memory transfers) that is within a constant factor (independent of B and M) of the memory performance of the optimal DAM algorithm, which knows B and M. There exist surprisingly many (asymptotically) optimal cache-oblivious algorithms; see e.g., [1, 8, 13–16, 18, 26–28, 30, 33, 39, 47, 48, 50, 51, 57].

I/O-Efficient Searching. A fundamental problem in computer science is how to search efficiently among N comparison-based totally-ordered elements on disk. The classic I/O-efficient search tree in the DAM model is ***B-tree*** [10, 32]. The basic idea of B-tree is to maintain a balanced tree of N elements with the node fan-out B, which is designed to fit into one memory block. The B-tree has height $\log_B N$, and a search optimally has $O(1) + \log_B N$ memory transfers (block cost). However, B-tree is designed with full knowledge of the block size B and therefore is only optimized for a two-level memory model. Although theoretically B-tree can be extended to fit a multilevel memory model, the resulting structure becomes much more complex than the original B-tree. Furthermore, as the number k of levels in the memory hierarchy grows, the constant factor of the search cost in an optimal k-DAM structure turns out to be bigger.

A static cache-oblivious search tree, proposed by Prokop [50], also performs

searches in $\Theta(\log_B N)$ memory transfers. It is built as follows: Embed a complete binary tree with N nodes in memory, conceptually splitting the tree at half its height, thus obtaining $\Theta(\sqrt{N})$ subtrees each with $\Theta(\sqrt{N})$ nodes. Lay out each of these trees contiguously, storing each recursively in memory. This type of recursive layout is called a ***van Emde Boas layout (vEB)*** because it is reminiscent of the recursive structure of the van Emde Boas tree [59, 60]. However, the constant factor of memory performance in the vEB search tree is much bigger than that of the B-tree in the DAM model. It would be interesting to narrow the gap between the vEB search tree and the B-tree in a multilevel memory hierarchy.

The static cache-oblivious search tree is a basic building block of essentially all cache-oblivious search structures, including the (dynamic) cache-oblivious B-tree of Bender, Demaine, and Farach-Colton [16], its simplifications and improvements [18, 30, 51], and other cache-oblivious search structures [1, 14, 15, 15, 27, 28]. Thus, any improvements to the static cache-oblivious search structure immediately translate to improvements to these dynamic structures.

Ordered Sparse Array. A classical problem in data structures and databases is how to maintain a dynamic set of N elements in sorted order in a $\Theta(N)$-sized array, which has been known for over two decades and studied under different names, including sparse arrays [43, 44], sequential file maintenance [64–66], and list labeling [34–37]. The problem is also closely related to the order-maintenance problem [13, 34, 36, 58].

The I/O-efficient and cache-oblivious version of the sparse array is called the ***packed memory array (PMA)*** [16, 17], which maintains N elements in sorted order in a $\Theta(N)$-sized array. The idea is to intersperse $\Theta(N)$ empty spaces or gaps among the elements so that only a small number of elements need to be shifted around on an insert or delete. This data structure effectively simulates a library bookshelf, where gaps on the shelves mean that books are easily added and removed. It supports the operations insert/delete in $O(1+(\log^2 N)/B)$ amortized memory transfers and scans of L consecutive elements in $\Theta(1+L/B)$ memory transfers.

The packed-memory array is an efficient and promising data structure, but it also has weaknesses. The main weakness is that the PMA performs relatively poorly on some common insertion patterns such as sequential inserts. For sequential

inserts, the PMA performs near its worst in terms of the number of elements moved per insert. Moreover, sequential inserts are common, and B-trees in databases are frequently optimized for this insertion pattern. It would be better if the PMA could perform near its best, not worst, in this case.

In contrast, one of the PMA's strengths is its performance on common insertion patterns such as random inserts. For random inserts, the PMA performs extremely well with only $O(\log N)$ element moves per insert and only $O(1+(\log N)/B)$ memory transfers. This performance surpasses the guarantees for arbitrary inserts.

The PMA has been used in cache-oblivious B-trees [16–19, 21, 30], concurrent cache-oblivious B-trees [24], cache-oblivious string B-tree [21], and scanning structures [13]. A sparse array in the same spirit as the PMA was independently proposed and used in the locality-preserving B-tree of [52], although the asymptotic space bounds are superlinear and therefore inferior to the linear space bounds of the earlier sparse-array data structures [43, 64–66] and the PMA [16, 17].

Balanced Search Tree. One of the most fundamental data structures for maintaining data on disk is a balanced search tree, which keeps data in order and supports operations such as search, insert, delete and range query. The classic external-memory search structure is B-tree [10, 32], which supports a search optimally in $O(1)+\log_B N$ memory transfers and the operations insert/delete asymptotically optimally in $O(\log_B N)$ memory transfers in the DAM model. Common variants, such as B^+-tree and B^*-tree [32, 45] are more implementable and have the same performance.

B-trees are balanced search trees where all nodes (except possibly the root) have fanout $\Theta(B)$ and the leaves are all at the same depth. Insertions and deletions are supported with a simple balancing scheme. If a block is too full, it is split into multiple blocks. If the block is too empty, then either the block borrows keys from a neighboring block, or else the block is merged with a neighboring block.

One of main deficiencies of the B-tree is that it cannot take full advantage of disk prefetching. The nodes in the B-tree may scattered through a disk in any order. Thus, each fetch of a B-tree node requires a random disk seek. Random block accesses perform two orders of magnitude more slowly than sequential block accesses. In this respect, B-tree performs inefficiently, especially for range queries.

In contrast, the ***dynamic cache-oblivious B-tree*** supports nearly optimal range queries by combining the above packed-memory array structure. Specifically, a range query in the cache-oblivious B-tree involves a search of a leaf block followed by a scan within an array. The first dynamic cache-oblivious B-tree achieves the asymptotically optimal searching cost of $O(\log_B N)$. This B-tree, designed by M. A. Bender, E. Demaine, and M. Farach-Colton [16], appears too complex to get good practical performance. Subsequently, Rahman, Cole, and Roman [51] proposed and implemented another cache-oblivious B-tree based on exponential trees, which supports insertion and deletion in $O(\log_B N + \log\log N)$ memory transfers. Another two related simplifications are obtained by Bender, Duan, Iacono, and Wu [18] and simultaneously by Brodal, Fagerberg, and Jacob [30]. Both of these achieves insert/delete performance in $O(\log_B N + (\log^2 N)/B)$ amortized memory transfers and scans of L consecutive elements optimally in $O(1 + L/B)$ memory transfers.

The cache-oblivious B-tree [18] combines two cache-oblivious structures in a fairly simple way: a static search tree stored in a van Emde Boas layout and the packed-memory array. Specifically, the structure includes a static complete binary tree with $\Theta(N)$ leaves, stored according to the van Emde Boas layout, and a packed-memory structure representing the elements. The structure maintains a fixed one-to-one correspondence between the cells in the packed-memory structure and the leaves in the tree. Some of these cells/leaves are occupied by elements, while others are blank. Therefore, operations are executed by a binary search through the top index tree, and insert, delete and scan in the bottom packed-memory array while updating the top index tree. However, this cache-oblivious B-tree also inherits deficiencies from both the vEB layout and the PMA.

Search Tree with Variable-Length Keys. The B-tree, as described in an algorithms textbook, is a dynamic dictionary designed to store unit-sized keys. For unit-sized keys and memory blocks of size B, the B-tree supports searches and updates at a cost of $O(\log_B N)$ memory transfers.

Industrial-strength B-trees, unlike textbook B-trees, support keys of different sizes. In many applications, such as file systems and databases, dictionaries are implemented using B-trees, even though the keys may have different sizes. For example, Berkeley DB [56] allows individual keys to be as large as 4 GB.

The B-tree operations still work correctly even when keys have different sizes. That is, neither splitting, merging, nor searching require keys to have identical sizes. However, they have no nontrivial performance guarantees.

There already exist dynamic dictionaries designed to store different-size keys, the most famous of which is the ***string B-tree*** [38]. (See Refs. [22, 29] for cache-oblivious string dictionaries.) In the string B-tree, the keys are chopped up and distributed among different nodes of the data structure. Searches and updates of a key κ run in $O(|\kappa|/B+\log_B N)$ memory transfers. Thus, the additional cost to access a key κ is just the additive cost, $\lceil|\kappa|/B\rceil$, to read key κ plus the cost to search in a B-tree, which is optimal.

The string B-tree is different from the B-tree because, as mentioned above, the keys are chopped up. B-trees cannot attain the efficiency of the string B-tree. However, despite their performance limitations and lack of performance guarantees, implementers often prefer to base applications such as file systems and databases, on B-trees.

We call a B-tree that supports different-size keys an ***atomic-key B-tree***. The keys are atomic in the sense that the keys are stored and manipulated in their entirety.

1.2 Results

The B-tree has been the dominant external-memory dictionary data structure for the last three decades, but it has several weaknesses degrading its performance. First, the search cost in a B-tree is only optimized for a two-level memory model. The B-tree's performance degrades in a multilevel memory model in which data locality is required at many levels of granularity. The B-tree cannot take advantage of disk prefetching. Finally, the B-tree's performance guarantees only apply when keys have unit or fixed length.

In this dissertation we address the above issues. We generalize the van Emde Boas layout so that as the number k of levels in the memory hierarchy grows, the search-performance of our cache-oblivious structure relative to an optimal k-DAM structure tends to zero.

The dynamic cache-oblivious B-tree solves some of the problems of B-trees

listed above. It performs efficient range queries and maintains data locality at all granularities. Our solutions, such as the generalized van Emde Boas layout, solve these problem optimizing the search constants.

We also present two improved versions of the packed-memory array: the adaptive PMA and the partially deamortized PMA, and therefore automatically improve the dynamic part in the cache-oblivious B-tree. We develop the first atomic-key B-tree having performance guarantees, which is as close as possible to the traditional B-tree while supporting atomic keys with variable lengths.

In Chapter 2, we first give an analysis of the static cache-oblivious search tree in vEB layout, proving that searches perform at most $2\left(1+3/\sqrt{B}\right)\log_B N + O(1)$ expected memory transfers; the expectation is taken only over the random placement of the data structure in memory. This analysis is tight to within a $1+o(1)$ factor.

We then present a class of ***generalized van Emde Boas layouts*** that optimizes performance through the use of uneven splits on the height of the tree. For any constant $\varepsilon > 0$, we optimize the layout achieving a performance of $[\lg e + \varepsilon + O(\lg\lg B/\lg B)]\log_B N + O(1)$ expected memory transfers. As before, the expectation is taken over the random placement of the data structure in memory. Our new search structure serves to disprove the common belief that even splits yield the best results in the worst case. We suggest the contrary: uneven splits can yield better worst-case performance.

Finally, we demonstrate that it is harder to search in the cache-oblivious model than in the DAM model. Previously the only lower bound for searching in the cache-oblivious model was the $\log_B N$ lower bound from the DAM model. We prove a lower bound of $\lg e \log_B N$ memory transfers for searching in the average case in the cache-oblivious model. Thus, for large B, our upper bound is within a factor of $1+o(1)$ of the optimal cache-oblivious layout.

In Chapter 3, we propose an ***adaptive packed-memory array*** (abbreviated ***adaptive PMA*** or ***APMA***), which adapts to common insertion patterns. We first show that the APMA has the "rebalance property", which ensures that any pattern of insertions cost only $O(1+(\log^2 N)/B)$ amortized memory transfers and $O(\log^2 N)$ amortized element moves. Because the elements are kept in sorted order in the APMA, as with the PMA, scans of L elements cost $O(1+L/B)$ memory transfers.

Thus, the adaptive PMA guarantees a performance at least as good as that of the traditional PMA. We next analyze the performance of the APMA under some common insertion patterns.

- We show that for ***sequential inserts***, where all the inserts are to the front of the array, the APMA makes only $O(\log N)$ amortized element moves and $O((\log N/B)+1)$ amortized memory transfers.
- We generalize this analysis to ***hammer inserts***, where the inserts hammer on any single element in the array.
- We then turn to ***random inserts***, where each insert occurs after a randomly chosen element in the array. We establish that the insertion cost is again only $O(\log N)$ amortized element moves and $O((\log N/B)+1)$ amortized memory transfers.
- We generalize all these previous results by analyzing the case of ***bulk inserts***. In the bulk-insert insertion pattern, we pick a random element in the array and perform N^{α} inserts after it for $\alpha \in [0,1]$. We show that for all values of $\alpha \in [0,1]$, the APMA also only performs $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ amortized memory transfers.
- We next perform simulations and experiments, measuring the performance of the APMA on these insertion patterns. For sequential insertions of roughly 1.4 million elements, the APMA has over four times fewer element moves per insertion than the traditional PMA and running times that are nearly seven times faster. For bulk insertions of 1.4 million elements, where $f(N) = N^{0.6}$, the APMA has over two times fewer element moves per insertion than the traditional PMA and running times that are over three times faster.

In Chapter 4, we propose the ***partially deamortized packed-memory array (PDPMA)*** for the purpose of decreasing the worst rebalance cost of one insertion. The partially deamortized PMA guarantees that the insert/delete cost per update is at most $O(\sqrt{N}\log N)$ element moves and $O(1+(\sqrt{N}\log N)/B)$ memory transfers while keeping the same update cost of $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ amortized memory transfers as the traditional PMA.

- We first give a better rebalance algorithm, called *one-phase rebalance*. Unlike the rebalance in the traditional PMA, which includes one scan of the array to

compress the elements and another scan to evenly space the elements out, we implement a one-phase rebalance in one scan of the array. In this way, we not only perform rebalance efficiently, but also decompose a big rebalance into small scans of leaves independently.

- We then propose the structure of the partially deamortized PMA and analyze its performance. The idea of this structure is to decompose the rebalance of size bigger than $\Omega(\sqrt{N}\log N)$ into smaller scans of size $O(\sqrt{N}\log N)$ by using our one-phase rebalance. After each smaller scan, a new element is inserted. In this way, we deamortize the insert cost to $O(\sqrt{N}\log N)$ in the worst case.

In Chapter 5, we develop the first ***atomic-key B-tree*** that has performance guarantees. The objective is to design a data structure as close as possible to the traditional B-tree. The performance of such structure depends on the average length $\hat{k}$ of the keys. In particular, the traditional B-trees have the performance $O(\lceil \hat{k}/B \rceil \log_{1+\lceil B/\hat{k} \rceil} N)$ when all keys have the same size $\hat{k}$. We come up the same bound when the keys have the same size and extend the bound even if the keys have different sizes.

- We first give an algorithm for building a *static* atomic-key B-tree. On a dictionary of n keys having average size $\hat{k}$, the expected cost to search for a random key is $O(\lceil \hat{k}/B \rceil \log_{1+\lceil B/\hat{k} \rceil} N)$ memory transfers, under the assumption that all keys are searched with uniform probability. The cost to build this tree is $O(N)$ operations and $O(N + N\hat{k}/B)$ memory transfers.

  To understand why this bound achieves our objective of storing different-size keys with the efficiency as same-size keys, we should plug in several values for the average key-size $\hat{k}$. If $\hat{k} = O(1)$, then the expected search cost is $O(\log_{B+1} N)$, the performance for a B-tree storing unit-size keys. On the other hand, if $\hat{k} = O(B)$, then the expected search cost is $O(\log_2 N)$. This is reasonable because the keys are so big that the branching factor is just constant, but sufficiently small that the access time for a given node is just $O(1)$. If the average key size is $\Omega(B)$, then again the branching factor is constant, but now the expected node access cost is $O(\lceil \hat{k}/B \rceil)$, which is $\Omega(1)$.

  In principle, it is nonoblivious that these bounds could be achievable because

different regions of the key space can have varying average key sizes.

- We then show how to build a *dynamic* atomic-key B-tree. The expect cost to search for random keys stays the same. The cost to insert an arbitrary key κ is the cost to search for κ plus a tree-update cost of $O(\lceil \hat{k}/B \rceil \log_{1+\lceil B/\hat{k} \rceil} N)$ amortized memory transfers. Thus, the update cost is dominated by the expected search cost.

  As an intermediate step in this construction, we first present an atomic-key B-tree, in which the cost to insert a key κ is the cost to search for κ plus an update cost of $O((\lceil B/\hat{k} \rceil + \lceil \hat{k}/B \rceil) \log_{1+\lceil B/\hat{k} \rceil} N)$ amortized memory transfers.

  We achieve our better bounds and then applying the technique of "indirection" on this first dynamic atomic-key B-tree. The objective is to divide the keys into groups, choose a representative from each group, and then build a dynamic atomic-key B-tree just on these representative elements. Unfortunately, this strategy does not work because the average key size of the representatives can be much larger than the average key size of all keys. Instead, we use a somewhat more and efficient use of indirection to avoid this potential problem.

- Finally, we give a dynamic-programming algorithm for constructing a static, atomic-key search tree having the minimum expected search cost. The algorithm takes as input the keys $\kappa_1, \ldots, \kappa_n$, their sizes, and their search probabilities $p_1, \ldots, p_n$. The algorithm uses $O(BN^3)$ operations.

Roadmap. The rest of this dissertation is organized as follows. In Chapter 2, we prove tight bounds on the cost of cache-oblivious searching and propose a generalized vEB layout. In Chapter 3, we give the first adaptive packed-memory array, which automatically adapts to common insertion patterns. We also show that our experiment results match the asymptotic bounds from our theoretical results. In Chapter 4, we design the partially deamortized packed-memory array. In Chapter 5, we first present a static atomic-key B-tree, and then we improve it to build a dynamic atomic-key B-tree.

Chapter 2

Cache-Oblivious Searching Cost[1]

In this chapter, we focus on a fundamental problem of searching: Given a set of N ordered elements, design a data structure which does searching operation efficiently. Here, we measure "efficiency" by memory transfers (block cost) because memory transfers are relatively slow.

Previous Work. A simple information-theoretic argument shows $\log_B N + O(1)$ is a lower bound in searching for an element among N elements as follows (See [33]):

Lemma 1 *Starting from an initially empty cache, at least* $\log_B N + O(1)$ *memory transfers is required to search for a desired element, in the average case.*

Proof. A general query element encodes $\lg(2N+1) + O(1) = \lg N + O(1)$ bits of information, because it can be any of the N elements or in any of the $N+1$ positions between the elements. The additive $O(1)$ comes from Kolmogorov complexity (See [49]). Each block read reveals where the query element fits among those B elements, which is at most $\lg(2B+1) = \lg B + O(1)$ bits of information. Thus, the number of block reads is at least $(\lg N + O(1))/(\lg B + O(1)) = \log_B N + O(1)$. □

In the disk access model (DAM), with knowledge of the block size B, the classic search B-tree with fan-out B, which is designed to fit into one memory block, optimally achieves searching performance $\log_B N + O(1)$ memory transfers. Therefore, the B-tree is optimal for searching in a two-level memory model by Lemma 1.

[1] An earlier version appears in [12].

However, we know modern machines have multilevel memory hierarchies, which can be modeled by extending the DAM model to k levels. As k grows, the search costs of the optimal k-level DAM search structure will increase for sure.

In this chapter, we present results in our published FOCS paper [12]. We first obtain the lower bound of searching in cache-oblivious structures by showing that as the number k of levels in the memory hierarchy grows, an optimal k-DAM structure has the search cost tending to $\lg e \log_B N$. Next, we propose a generalized van Emde Boas layout which is a constant approximation where the constant $\lg e$ is better than that in the original vEB layout. Therefore, for a multilevel memory hierarchy, a simple cache-oblivious structure almost replicates the performance of an optimal parameterized k-level DAM structure.

2.1 Lower Bound for Cache-Oblivious Searching

In this section, we prove lower bounds for the cost of cache-oblivious comparison-based searching. The problem we consider is the average cost of successful searches among N distinct elements, where the average is over a uniform distribution of the search key y on the N input elements. For lower bounds, average case complexity is stronger than worst case complexity, so our bounds also apply to the worst case cost. We note that our bounds hold even if the block sizes are known to the algorithm, and that they hold for any memory layout of data, including any specific placement of a single data structure.

Formally, our model is as follows. Given a set S of N elements $x_1 < \cdots < x_N$ from a totally ordered universe, a *search structure* for S is an array M containing elements from S, possibly with several copies of each. A *search algorithm* for M is a binary decision tree where each internal node is labeled with either $y < M[i]$ or $y \leq M[i]$ for some array index i, and each leaf is labeled with a number $1 \leq j \leq N$. A search on a key y proceeds in a top-down fashion in the tree, and at each internal node advances to the left child if the comparison given by the label is true, otherwise it advances to the right. A binary decision tree is a correct search algorithm if for any $x_i \in S$, the path taken by a search on key $y = x_i$ ends in a leaf labeled i. Any such tree must have at least N leaves, and by pruning paths not taken by any search for $x_1, \ldots, x_N$, we may assume that it has exactly N leaves.

To add I/Os to the model, we divide the array M into contiguous *blocks* of size B. An internal node of a search algorithm is said to *access* the block containing the array index i in the label of the node. We define the I/O cost of a search to be the number of distinct blocks of M accessed on the path taken by the search.

The main idea of our proof is to analyze the I/O cost of a given search algorithm with respect to several block sizes simultaneously. We first describe our method for the case of two block sizes. This will lead to a lower bound of $1.207 \log_B N$ block transfers. We then generalize this proof to a larger number k of block sizes, and prove that in the limit as k grows, this gives a lower bound of $\lg e \log_B N \approx 1.443 \log_B N$ block transfers.

Throughout this section, we assume that block sizes are powers of two and that blocks start at memory addresses divisible by the block size. This reflects the situation on actual machines, and entails no loss of generality, as any cache-oblivious algorithm at least should work for this case. The assumption implies that for two block sizes $B_1 < B_2$, a block of size B_1 is contained in exactly one block of size B_2.

Lemma 2 ([46, Section 2.3.4.5]) *For a binary tree with N leaves, the average depth of a leaf is at least* $\lg N$.

Lemma 3 *If a search algorithm on a search structure for block sizes B_1 and B_2, where $B_2 = {B_1}^c$ and $1 < c \leq 2$, guarantees that the average number of block reads is at most $\delta \log_{B_1} N$ and $\delta \log_{B_2} N$, respectively, then*

$$\delta \geq \frac{1}{2/c + c - 2 + 3/(c \lg B_1)} \ .$$

Proof. Let T denote the binary decision tree constituting the search algorithm. Our goal is to transform T into a new binary decision tree T' by transforming each node that accesses a new size B_1 block in T into a binary decision tree of small height, and discarding all other nodes in T. A lower bound on the average depth of leaves in T' then translates into a lower bound on the average number of blocks accesses in T.

To count the number of I/Os of each type (size B_1 blocks and size B_2 blocks) for each path in T, we mark some of the internal nodes by tokens τ_1 and τ_2. A node

v is marked iff none of its ancestors accesses the size B_1 block accessed by v, i.e. if v is the first access to the block. The node v may also be the first access to the size B_2 block accessed by v. In this case, v is marked by τ_2, else it is marked by τ_1. Note that the word “first” above corresponds to viewing each path in the tree as a time line—this view will be implicit in the rest of the proof.

For any root-to-leaf path, let b_i denote the number of distinct size B_i blocks accessed and let a_i denote the number of τ_i tokens on the path, for $i = 1, 2$. By the assumption stated above Lemma 2, a first access to a size B_2 block implies a first access to a size B_1 block, so we have $b_2 = a_2$ and $b_1 = a_1 + a_2$.

We transform T into a new binary decision tree T' in a top-down fashion. The basic step in the transformation is to substitute a marked node v with a specific binary decision tree T_v resolving the relation between the search key y and a carefully chosen subset S_v of the elements. More precisely, in each step of the transformation, the subtree rooted at v is first removed, then the tree T_v is inserted at v's former position, and finally a copy of one of the two subtrees rooted at the children of v is inserted at each leaf of T_v. The top-down transformation then continues downwards at the leafs of T_v. When the transformation reaches a leaf, it is left unchanged. The resulting tree can contain several copies of each leaf of T.

We now describe the tree T_v inserted, and first consider the case of a node v marked τ_2. We let the subset S_v consist of the at most B_1 distinct elements in the block of size B_1 accessed by v, plus every $\frac{B_2}{2B_1}$-th element in sorted order among the at most B_2 distinct elements in the block of size B_2 accessed by v. The size of S_v is at most $B_1 + B_2/(B_2/(2B_1)) = 3B_1$.

The tree T_v is a binary decision tree of minimal height resolving the relation of the search key y to all keys in S_v. If we have $S_v = \{z_1, z_2, \ldots, z_t\}$, with elements listed in sorted order and $t \leq 3B_1$, this amounts to resolving which of the at most $6B_1 + 1$ intervals

$$(-\infty; z_1)\ ,\ [z_1; z_1]\ ,\ (z_1; z_2)\ , \ldots ,\ [z_t; z_t]\ ,\ (z_t; \infty)$$

that y belongs to (we resolve for equality because we chose to allow both $<$ and $\leq$ comparisons in the definition of comparison trees, and want to handle both types of nodes in the transformation). The tree T_v has height at most $\lceil \lg(6B_1 + 1) \rceil$, since a perfectly balanced binary search tree on S_v, with one added layer to resolve equality

questions, will do. As B_1 is a power of two, $\lg(8B_1)$ is an integer and hence an upper bound on the height.

For the case of a node v marked τ_1, note that v in T has exactly one ancestor u marked τ_2 that accesses the same size B_2 block β as v does. When the tree T_u was substituted for u, the inclusion in S_u of the $2B_1$ evenly sampled elements from β ensures that below any leaf of T_u, at most $\frac{B_2}{2B_1} - 1$ of the elements in β can still have an unknown relation to the search key. The tree T_v is a binary decision tree of minimal height resolving these relations. Such a tree has at most $2\frac{B_2}{2B_1} - 1 = \frac{B_2}{B_1} - 1$ leaves and hence height at most $\lg \frac{B_2}{B_1}$, as B_1 and B_2 are powers of two.

Since in both cases T_v resolves the relation between the search key y and all sampled elements, the relation between the search key and the element accessed at v is known at each leaf of T_v, and we can choose either the left or right child of v to continue the transformation with.

When we in the top-down transformation meet an unmarked internal node v (i.e. a node where the size B_1 block accessed at the node has been accessed before), we can similarly discard v together with either the left or right subtree, since we already have resolved the relation between the search key y and the element accessed at v. This follows from the choice of trees inserted at marked nodes: when we access a size B_2 block β_2 for the first time at some node u, we resolve the relation between the search key y and all elements in the size B_1 block β_1 accessed at u (due to the inclusion of all of β_1 in S_u), and when we first time access a key in β_2 outside β_1, we resolve all remaining relations between y and elements in β_2.

The tree T' resulting from this top-down transformation is a binary decision tree. By construction, each search in T' ends in a leaf having the same label as the leaf that the same search in T ends in (this is an invariant during the transformation), so T' is a correct search algorithm if T is.

By the height stated above for the inserted T_v trees, it follows that if a search for a key y in T corresponds to a path containing a_1 and a_2 tokens of type τ_1 and τ_2, respectively, then the search in T' corresponds to a path with length bounded by the following expression.

$$\begin{aligned} a_2 \lg(8B_1) + a_1 \lg \frac{B_2}{B_1} &= b_2 \lg(8B_1) + (b_1 - b_2) \lg \frac{B_2}{B_1} \\ &= b_2 \left[\lg(8B_1) - \lg \frac{B_2}{B_1} \right] + b_1 \lg \frac{B_2}{B_1} \end{aligned}$$

The coefficients of b_2 and b_1 are positive by the assumption $B_1 < B_2 \le {B_1}^2$, so upper bounds on b_1 and b_2 imply an upper bound on the expression above. By assumption, the average values over all search paths of b_1 and b_2 are bounded by $\delta \log_{B_1} N$ and $\delta \log_{B_2} N = (\delta \log_{B_1} N)/c$, respectively.

If we prune the tree for paths not taken by any search for the keys $x_1, \ldots, x_N$, the lengths of root-to-leafs paths can only decrease. The resulting tree has N leaves, and Lemma 2 gives a $\lg N$ lower bound on the average depth of a leaf. Hence, we get

$$\begin{aligned} & \lg N \le \frac{\delta}{c} \log_{B_1} N \left[\lg(8B_1) - \lg \frac{B_2}{B_1} \right] + \delta \log_{B_1} N \lg \frac{B_2}{B_1} \\ = \; & \frac{\delta}{c} \log_{B_1} N [3 + \lg B_1 - (c-1) \lg B_1] + \delta \log_{B_1} N (c-1) \lg B_1 \\ = \; & \delta \lg N [3/(c \lg B_1) + 1/c - (c-1)/c + (c-1)] \\ = \; & \delta \lg N [3/(c \lg B_1) + c + 2/c - 2] \; . \end{aligned}$$

It follows that $\delta \ge 1/[3/(c \lg B_1) + c + 2/c - 2]$. □

Corollary 4 *If a search algorithm on a search structure guarantees, for all block sizes B, that the average number of block reads for a search is at most $\delta \log_B N$, then $\delta \ge 1/(2\sqrt{2} - 2) \approx 1.207$.*

Proof. Letting $c = \sqrt{2}$ in Lemma 3, we get $\delta \ge 1/[2\sqrt{2} - 2 + 3/(\sqrt{2} \lg B_1)]$. The lower bound follows by letting B_1 grow to infinity. □

Lemma 5 *If a search algorithm on a search structure for block sizes $B_1, B_2, \ldots, B_k$, where $B_i = {B_1}^{c_i}$ and $1 = c_1 < c_2 < \cdots < c_k \le 2$, guarantees that the average number of block reads for a search is at most $\delta \log_{B_i} N$ for each block size B_i, then*

$$\delta \ge \frac{1}{\sum_{i=1}^{k-1} \frac{c_{i+1}}{c_i} + \frac{2}{c_k} \left[1 + \frac{\lg(8k)}{2 \lg B_1} \right] - k} \; .$$

Proof. The proof is a generalization of the proof of Lemma 3 for two block sizes, and we here assume familiarity with that proof. The transformation is basically the same, except that we have a token τ_i, $i = 1, \ldots, k$, for each of the k block sizes.

Again, a node v is marked if none of its ancestors access the size B_1 block accessed by v, i.e. if v is the first access to the block. The node v may also be the first access to blocks of larger sizes, and we mark v by τ_i, where B_i is the largest block size for which this is true. Note that v must be the first access to the size B_j block accessed by v for all j with $1 \leq j \leq i$.

For any root-to-leaf path, let b_i denote the number of distinct size B_i blocks accessed and let a_i denote the number of τ_i tokens on the path, for $i = 1, \ldots, k$. We have $b_i = \sum_{j=i}^{k} a_j$. Solving for a_i, we get $a_k = b_k$ and $a_i = b_i - b_{i+1}$, for $i = 1, \ldots, k-1$.

As in the proof of Lemma 3, the transformation proceeds in a top-down fashion, and substitutes marked nodes v by binary decision trees T_v. We now describe the trees T_v for different types of nodes v.

For a node v marked τ_k, the tree T_v resolves the relation between the query key y and a set S_v of size $(2k-1)B_1$, consisting of the B_1 elements in the block of size B_1 accessed at v, plus for $i = 2, \ldots, k$ every $\frac{B_i}{2B_1}$th element in sorted order among the elements in the block of size B_i accessed at v. This tree can be chosen to have height at most $\lceil \lg(2(2k-1)B_1 + 1) \rceil \leq \lg(8kB_1)$.

For a node v marked τ_i, $i < k$, let β_j be the block of size B_j accessed by v, for $1 \leq j \leq k$. For $i+1 \leq j \leq k$, β_j has been accessed before, by the definition of τ_i. We now consider two cases. Case I is that β_{i+1} is the only block of size B_{i+1} that has been accessed inside β_k. By the definition of the tree T_u inserted at the ancestor u of v where β_k was first accessed, at most $B_{i+1}/2B_1 - 1$ of the elements in β_{i+1} can have unknown relations with respect to the search key y. The tree T_v inserted at v resolves these relations. It can be chosen to have height at most $\lg \frac{B_{i+1}}{B_1}$. Case II is that β_{i+1} is not the only block of size B_{i+1} that has been accessed inside β_k. Then consider the smallest j for which β_{j+1} *is* the only block of size B_{j+1} that has been accessed inside β_k. When we first time accessed the second block of size B_j inside β_k at some ancestor u of v, this access was necessarily inside β_{j+1}, and a Case I substitution as described above took place. Hence a tree T_u was inserted which resolved all relations between the search key and elements in β_{j+1}, and the empty tree can be used for T_v, i.e. v and one of its subtrees can simply be discarded.

For an unmarked node v, there is a token τ_i on the ancestor u of v in T where the size B_1 block β_1 accessed by v was first accessed. This gave rise to a tree T_u

in the transformation, and this tree resolved the relations between the search key and all elements in β_1, either directly ($i = k$) or by resolving the relations for all elements in a block containing β_1 ($1 \le i < k$), so v and one of its subtrees can be discarded.

After transformation and final pruning, the length of a root-to-leaf path in the final tree is bounded by the following equation.

$$\begin{aligned} & a_k \lg(8kB_1) + \sum_{i=1}^{k-1} a_i \lg \frac{B_{i+1}}{B_1} = b_k \lg(8kB_1) + \lg B_1 \sum_{i=1}^{k-1} (b_i - b_{i+1})(c_{i+1} - 1) \\ = \;& \lg B_1 \left[b_k \left(1 + \frac{\lg(8k)}{\lg B_1}\right) + b_1(c_2 - 1) \; + \sum_{i=2}^{k-1} b_i(c_{i+1} - c_i) - b_k(c_k - 1) \right] \\ = \;& \lg B_1 \left[\sum_{i=1}^{k-1} b_i(c_{i+1} - c_i) + b_k \left(2 + \frac{\lg(8k)}{\lg B_1} - c_k\right) \right]. \end{aligned}$$

For all i, the average value of b_i over all search paths is by assumption bounded by $\delta \log_{B_i} N = (\delta \log_{B_1} N)/c_i$, and the coefficient of b_i is positive, so we get the following bound on the average number of comparisons on a search path.

$$\begin{aligned} & \delta \log_{B_1} N \lg B_1 \left[\sum_{i=1}^{k-1} \frac{1}{c_i}(c_{i+1} - c_i) \; + \frac{1}{c_k}\left(2 + \frac{\lg(8k)}{\lg B_1} - c_k\right) \right] \\ = \;& \delta \lg N \left[\sum_{i=1}^{k-1} \frac{c_{i+1}}{c_i} + \frac{1}{c_k}\left(2 + \frac{\lg(8k)}{\lg B_1}\right) - k \right]. \end{aligned}$$

By Lemma 2 we have

$$\delta \lg N \left[\sum_{i=1}^{k-1} \frac{c_{i+1}}{c_i} + \frac{1}{c_k}\left(2 + \frac{\lg(8k)}{\lg B_1}\right) - k \right] \ge \lg N ,$$

and the lemma follows. □

Theorem 6 *If a search algorithm on a search structure guarantees, for all block sizes B, that the average number of block reads for a search is at most $\delta \log_B N$, then $\delta \ge \lg e \approx 1.443$.*

Proof. Let k be an integer, and for $i = 1, \ldots, k$ define $B_i = 2^{k+i-1}$. In particular, we have $B_i = {B_1}^{c_i}$ with $c_i = (k+i-1)/k$. Consider the following subexpression of Lemma 5.

$$\begin{aligned}
& \frac{2}{c_k}\left(1+\frac{\lg(8k)}{2\lg B_1}\right)+\sum_{i=1}^{k-1}\frac{c_{i+1}}{c_i}-k \\
= \; & \frac{2k}{2k-1}\left(1+\frac{\lg(8k)}{2k}\right)+\sum_{i=1}^{k-1}\frac{k+i}{k+i-1}-k \\
= \; & \frac{2k}{2k-1}\left(1+\frac{\lg(8k)}{2k}\right)-1+\sum_{i=1}^{k-1}\frac{1}{k+i-1} \\
\leq \; & \frac{2k}{2k-1}\left(1+\frac{\lg(8k)}{2k}\right)-1+\int_{k-1}^{2k-2}\frac{1}{x}\,dx \\
= \; & \frac{2k}{2k-1}\left(1+\frac{\lg(8k)}{2k}\right)-1+\ln 2.
\end{aligned}$$

Letting k grow to infinity Lemma 5 implies $\delta \geq 1/\ln 2 = \lg e$. □

2.2 Upper Bound for van Emde Boas Layout

In this section we give tight analyses of the cost of searching in a binary tree stored with van Emde Boas layout [50]. As mentioned earlier, in the vEB layout, the tree is split evenly by height, except for roundoff. Thus, a tree of height h is split into a top tree of height $\lceil h/2 \rceil$ and bottom tree of height $\lfloor h/2 \rfloor$. Publications [16, 18, 30] show that the number of memory transfers for a search is $4\log_B N$ in the *worst case*; we give a matching configuration showing that this analysis is tight. We then consider the average-case performance over starting positions of the tree in memory, and we show that the expected search cost is $2(1+3/\sqrt{B})\log_B N + O(1)$ memory transfers, which is tight within a $1+o(1)$ factor. We assume that the data structure begins at a random position in memory; if there is not enough space, then the data structure "wraps around" to the first location in memory.

A relatively straightforward analysis of this layout shows that in the worst case the number of memory transfers is no greater than four times that of the optimal *cache-size-aware* layout. More formally,

Theorem 7 *Consider an $(N-1)$-node complete binary search tree that is stored using the Prokop vEB layout. A search in this tree has memory-transfer cost of* $\left(4-\frac{4}{2+\lg B}\right)\log_B N$ *in the worst case.*

Proof. The upper bound has been established before in the literature [16, 18, 30]. For the lower bound we show that this value is achieved asymptotically. Let the block size be $B = \left(2^{2k}-1\right)/3$ for any odd number k and consider a tree T of size $N-1$, where $N = 2^{k2^{m+1}}$ for some constant m. Number the positions within a block from 0 to $B-1$. As we recurse, we eventually obtain subtrees of size $3B = 2^{2k}-1$ and one level down of size 2^k-1. We align the subtree of size $3B$ that contains the root of T so that its first subtree of size 2^k-1 (which also contains the root of T) starts in position $B-1$ of a block. In other words, any root-to-leaf search path in this subtree crosses the block boundary because the root is in the last position of a block. Consider the $\left((2^k+1)/3+1\right)$th subtree of size 2^k-1. The root of this tree starts at position $B-1+(2^k-1)(2^k+1)/3 = 2B-1$, which is also the last position of a block. Thus, any root-to-leaf search path in this subtree crosses the block boundary. Observe that because trees are laid out consecutively, and $3B$ is a multiple of the block size, all other subtrees of size $3B$ start at position $B-1$ inside a block and share the above property (that we can find a root-to-leaf path that has cost 4 inside this size-$3B$ subtree). Notice that a root-to-leaf path accesses 2^m many size-$3B$ subtrees, and if we choose the path according to the above position we know that the cost inside each size $3B$ subtree is 4. More precisely, each size 2^k-1 subtree on this path starts at position $B-1$ in a block. Thus, the total search cost is $4\cdot 2^m$. Because $2^{k2^{m+1}} = N$ and $3B = 2^{2k}-1$, we have

$$4\cdot 2^m = \frac{4\log_B N}{\log_B(3B+1)} = 4\frac{\lg B}{\lg(3B+1)}\log_B N.$$

Furthermore, we bound the parameter $\lg B/\lg(3B+1)$ as follows:

$$\begin{aligned}
\frac{\lg B}{\lg(3B+1)} &< \frac{\lg B}{\lg 3B} \\
&= 1-\frac{\lg 3}{\lg 3+\lg B} \\
&< 1-\frac{1}{2+\lg B}.
\end{aligned}$$

Therefore, the total search cost has $4(1-1/(2+\lg B))\log_B N$ memory transfers in the worst case. □

However, few paths in the tree have this property, which suggests that in practice, the Prokop vEB layout results in a much lower memory-transfer cost assuming random placement in memory.

In Theorem 9, appearing shortly, we formalize this notion. First, however, we give the following useful inequality to simplify the proof.

Claim 8 *Let B be a power of 2, t and t' be positive numbers satisfying $t/2 \le t' \le 2t$, $\sqrt{B}/2 \le t \le \sqrt{B}$, and $tt' \ge B$. Then*

$$2+\frac{t+t'}{B} \le 2\left(1+\frac{3}{\sqrt{B}}\right)\frac{\lg t+\lg t'}{\lg B}.$$

Proof. Because $t^2+(t')^2 \le 5tt'/2$ for all $t/2 \le t' \le 2t$, we have

$$t+t' \le 3\sqrt{\frac{tt'}{2}}. \tag{1}$$

Define $x = tt'$ and define

$$f(x) = 2\left(1+\frac{3}{\sqrt{B}}\right)\frac{\lg x}{\lg B} - 2 - \frac{3}{B}\sqrt{\frac{x}{2}}.$$

We will show that $f(x) \ge 0$ for $B \le x \le 2B$. First, we calculate the second derivative of $f(x)$.

$$f''(x) = -2\left(1+\frac{3}{\sqrt{B}}\right)\frac{1}{x^2\ln B} + \frac{3}{4\sqrt{2}B}\frac{1}{x^{3/2}}.$$

Because $x \le 2B$ (i.e., $x^{1/2} \le \sqrt{2B}$), we obtain

$$f''(x) \le \frac{1}{x^2}\left[\frac{3\sqrt{2B}}{4\sqrt{2}B} - 2\left(1+\frac{3}{\sqrt{B}}\right)\frac{1}{\ln B}\right].$$

By removing the term $-6/(\sqrt{B}\ln B)$, we bound $f''(x)$ as follows:

$$f''(x) \le \frac{1}{x^2}\left(\frac{3}{4\sqrt{B}} - \frac{2}{\ln B}\right) \le 0.$$

Thus, we establish that $f(x)$ is convex in the range $B \leq x \leq 2B$. Because both $f(B)$ and $f(2B)$ are greater than zero, we obtain $f(x) \geq 0$ for $B \leq x \leq 2B$, which is equivalent to

$$2 + \frac{3}{B}\sqrt{\frac{x}{2}} \leq 2\left(1 + \frac{3}{\sqrt{B}}\right)\frac{\lg x}{\lg B}.$$

From (1) and the above inequality, we obtain the follows:

$$2 + \frac{t+t'}{B} \leq 2\left(1 + \frac{3}{\sqrt{B}}\right)\frac{\lg x}{\lg B}.$$

□

Theorem 9 *Consider a path in an* $(N-1)$*-node complete binary search tree of height h that is stored in vEB layout, with the initial page starting at a uniformly random position in a block B. Then the expected memory-transfer cost of the search is at most* $2(1+3/\sqrt{B})\log_B N$.

Proof. Although the recursion proceeds to the base case where trees have height 1, conceptually we stop the recursion at the level of detail where each recursive subtree has at most B nodes. We call those subtrees *critical recursive subtrees*, because they are recursive subtrees in the most ”important” level of detail. Let the number of nodes in a subtree T be $|T|$. Therefore, any critical recursive subtree T has $|T|$ nodes, where $\sqrt{B}/2 \leq |T| \leq B$. Note that because of roundoff, we cannot guarantee that $|T| \geq \sqrt{B}$. In particular, if a tree has $B+1$ nodes and its height h' is odd, then the bottom trees have height $\lfloor h'/2 \rfloor$, and therefore contain roughly $\sqrt{B}/2$ nodes. Then there are exactly $|T|-1$ initial positions for the upper tree that results in T being laid out across a block boundary. Similarly there are $B-|T|+1$ positions in which the block does not cross a block boundary. Hence, the local expected cost of accessing T is

$$\frac{2(|T|-1)}{B} + \frac{B-|T|+1}{B} = 1 + \frac{|T|-1}{B}.$$

Now we need two cases to deal with the roundoff. If $\sqrt{B}/2 \leq |T| \leq \sqrt{B}$ for the critical recursive subtree T, then we consider the next larger level of detail. There exists another critical recursive subtree T' immediately above T on the search path in this level of detail. Notice that $|T||T'| \geq B$. Because otherwise we would consider

the coarser level of detail for our critical recursive subtree. Because we cut in the middle, we know that $2|T'| \geq |T| \geq |T'|/2$. From Claim 8 the expected cost of accessing T and T' is at most

$$1+\frac{|T|-1}{B}+1+\frac{|T'|-1}{B} \leq 2\left(1+\frac{3}{\sqrt{B}}\right)\frac{\lg(|T||T'|)}{\lg B}.$$

For $\sqrt{B} \leq |T| \leq B$ for the critical recursive subtree T, we show that the cost of accessing T is less than $2(1+1/\sqrt{B})\lg|T|/\lg B$. Define $f(x)$ as follows:

$$f(x) = 2\frac{\lg x}{\lg B}\left(1+\frac{1}{\sqrt{B}}\right)-1-\frac{x-1}{B}.$$

By calculating

$$f''(x) = -\frac{2}{x^2 \lg B}\left(1+\frac{1}{\sqrt{B}}\right) \leq 0,$$

we know $f(x)$ is convex. Because both $f(\sqrt{B})$ and $f(B)$ are greater than zero, we obtain $f(x) \geq 0$ for the entire range $\sqrt{B} \leq x \leq B$. Thus, considering $f(|T|)$, we obtain that the expected cost of accessing T is

$$1+\frac{|T|-1}{B} \leq 2(1+\frac{1}{\sqrt{B}})\frac{\lg|T|}{\lg B}.$$

Combining the above arguments, we conclude that although the critical recursive subtrees on a search path may have different sizes, their expected memory-transfer cost is at most

$$\sum_T 2\left(1+\frac{3}{\sqrt{B}}\right)\frac{\lg|T|}{\lg B} = 2\left(1+\frac{3}{\sqrt{B}}\right)\log_B N.$$

This is a factor of $2(1+3/\sqrt{B})$ times the (optimal) performance of a B-tree. □

2.3 Upper Bound for the Generalized van Emde Boas Layout

We now propose and analyze a *generalized van Emde Boas layout* having a better search cost. In the original vEB layout, the top recursive subtree and the bottom

recursive subtrees have the same height (except for roundoff). At first glance this even division would seem to yield the best memory-transfer cost. Surprisingly, we can improve the van Emde Boas layout by selecting different heights for the top and bottom subtrees.

The generalized vEB layout is as follows: Suppose the complete binary tree contains $N-1=2^h-1$ nodes and has height $h=\lg N$. Let a and b be constants such that $0<a<1$ and $b=1-a$. Conceptually we split the tree at the edges below the nodes of depth $\lceil ah \rceil$. This splits the tree into a *top recursive subtree* of height $\lceil ah \rceil$, and $k=2^{\lceil ah \rceil}$ *bottom recursive subtrees* of height $\lfloor bh \rfloor$. Thus, there are roughly N^a bottom recursive subtrees and each bottom recursive subtree contains roughly N^b nodes. We map the nodes of the tree into positions in the array by recursively laying out the subtrees contiguously in memory. The base case is reached when the trees have one node, as in the standard vEB layout.

We find the values of a and b that yield a layout whose memory-transfer cost is arbitrarily close to $[\lg e + O(\lg\lg B/\lg B)]\log_B N + O(1)$ for $a=1/2-\xi$ and large enough N. We focus our analysis on the first level of detail where recursive subtrees have size at most the block size B. In our analysis memory transfers can be classified in two types. There are $\mathcal{V}$ *path-length memory transfers*, which are caused by accessing different recursive subtrees in the level of detail of the analysis, and there are $\mathcal{C}$ *page-boundary memory transfers*, which are caused by a single recursive subtree in this level of detail straddling two consecutive blocks. It turns out that each of these components has the same general recursive expression and differs only in the base cases. The total number of memory transfers is at most $\mathcal{V}+\mathcal{C}$ by linearity of expectation.

The recurrence relation obtained contains rounded-off terms ($\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$) that are cumbersome to analyze. We show that if we ignore the roundoff operators, then the error term is small. We obtain a solution expressed in terms of a power series of the roots of the characteristic polynomial of the recurrence. We show for both $\mathcal{V}$ and $\mathcal{C}$ that the largest root is unique and hence dominates all other roots, resulting in asymptotic expressions in terms of the dominant root.

Using these asymptotic expressions, we obtain the main result, namely a layout whose total cost is arbitrarily close to $[\lg e + O(\lg\lg B/\lg B)]\log_B N + O(1)$ as the split factor $a=1/2-\xi$ approaches $1/2$ and for N large enough. This performance

matches the lower bound from the Section 2.1 up to low-order terms.

Causes of Memory Transfers: Path-Length and Block-Boundary-Crossing Functions

We let $\mathcal{B}(x)$ denote the expected block cost of a search in a tree of height x. To begin, we explain the base case for the recurrence, when the entire tree is a critical recursive subtree. Recall that a *critical recursive subtree* is a recursive subtree of size less than B. If a critical recursive subtree crosses a block boundary, then the block cost is 2; otherwise the block cost is 1. As in the Theorem 9, the expected block cost of accessing a critical recursive subtree T of size $|T| = t-1$ and height $x = \lg t$ is

$$1 + \frac{t-2}{B} = 1 + \frac{2^x - 2}{B}.$$

Thus, the base case is when $|T| < B$, which means that $t \leq B$ and $1 \leq x \leq \lg B$.

We next give the recurrence for the block cost $\mathcal{B}(x)$ of a tree T of height x. By linearity of expectation, the expected block cost is at most that of the top recursive subtree plus the bottom recursive subtree, i.e.,

$$\mathcal{B}(x) \leq \mathcal{B}(\lceil ax \rceil) + \mathcal{B}(\lfloor bx \rfloor),$$

for $x > \lg B$, [2] for $a + b = 1$, $0 < a \leq b < 1$.

We decompose (an upper bound on) the cost of $\mathcal{B}(x)$ into two pieces. Let $\mathcal{V}(x)$ be the number of critical recursive subtrees visited along a root-to-leaf path ($\mathcal{V}$ stands for "vertical"), i.e.,

$$\mathcal{V}(x) = \begin{cases} \mathcal{V}(\lceil ax \rceil) + \mathcal{V}(\lfloor bx \rfloor), & x > \lg B; \\ 1, & 1 \leq x \leq \lg B. \end{cases} \tag{2}$$

Let $\mathcal{C}(x)$ be the expected number of critical recursive subtrees straddling block boundaries along the root-to-leaf path ($\mathcal{C}$ stands for "crossing"), i.e.,

$$\mathcal{C}(x) = \begin{cases} \mathcal{C}(\lceil ax \rceil) + \mathcal{C}(\lfloor bx \rfloor), & x > \lg B; \\ (2^x - 2)/B, & 1 \leq x \leq \lg B. \end{cases} \tag{3}$$

[2] We cannot claim equality, i.e., that $\mathcal{B}(x) = \mathcal{B}(\lceil ax \rceil) + \mathcal{B}(\lfloor bx \rfloor)$, because the leaf node of the top recursive subtree and root node of a bottom recursive subtree can belong to the same block. Thus, an equal sign in the recurrence might double count one memory transfer.

Observe that both $\mathcal{V}(x)$ and $\mathcal{C}(x)$ are monotonically increasing. By linearity of expectation, we obtain

$$\mathcal{B}(x) \le \mathcal{V}(x) + \mathcal{C}(x)$$

for all $x \ge \lg B$.

The recurrences for $\mathcal{V}(x)$ and $\mathcal{C}(x)$ are both of the form

$$\mathcal{F}(x) = \mathcal{F}(\lceil ax \rceil) + \mathcal{F}(\lfloor bx \rfloor).$$

As we will see, it is easier to analyze a recurrence of the form

$$\mathcal{G}(x) = \mathcal{G}(ax) + \mathcal{G}(bx),$$

where the roundoff error is removed. In the next few pages, we show that $\mathcal{F}(x)$ can be approximated by $\mathcal{G}(x)$ as x increases. Afterwards, we show how to calculate $\mathcal{G}(x)$.

Roundoff Error Is Small

We next show that as x increases, the difference between $\mathcal{F}(x)$ and $\mathcal{G}(x)$ can be bounded. To quantify the difference between $\mathcal{F}(x)$ and $\mathcal{G}(x)$ — see Theorem 13 — we use functions $\beta(x)$ and $\delta(x)$ defined recursively below:

Definition 10 *Let $a < \min\{1/2, 1 - 2/\lg B\}$. Define the recursive function $\beta(x)$ and $\delta(x)$ as follows:*

$$\beta(x) = \begin{cases} 0, & x \le \lg B; \\ \beta(ax+1)+1, & x > \lg B. \end{cases}$$

$$\delta(x) = \begin{cases} 1, & x \le \lg B; \\ \delta(ax+1)(1 + 2a^{\beta(x)-2}/\lg B), & x > \lg B. \end{cases}$$

The following lemma gives upper and lower bounds of $\beta(x)$.

Lemma 11 *For all $x > \lg B$, the function $\beta(x)$ satisfies*

$$\frac{2}{a^2 x} \ge \frac{a^{\beta(x)-2}}{\lg B} \ge \frac{1}{2ax}. \tag{4}$$

Proof. For parameter n, define the *nth interval* I_n to be

$$I_n = \left[\frac{\lg B}{2a^{n-1}} + \frac{1}{1-a}, \frac{\lg B - 1 - a - \cdots - a^{n-1}}{a^n}\right].$$

We now prove the following inequality for all $x > \lg B$:

$$\frac{1}{2}\lg B\left(\frac{1}{a}\right)^{\beta(x)-1} \leq x - \frac{1}{1-a} \leq \lg B\left(\frac{1}{a}\right)^{\beta(x)}. \tag{5}$$

We establish (5) in two parts.

1. We first show that the inequality holds for all n and all $x \in I_n$.
2. We then explain that the interval $I_0 \cup I_1 \cup I_2 \cup \cdots$ covers the interval $[\lg B, \infty)$.

We now prove the first part, showing by induction on n that (5) holds for all n and all $x \in I_n$.

Base Case: The base case is when

$$x \in I_0 = \left[\frac{a}{2}\lg B + \frac{1}{1-a}, \lg B\right].$$

Because $a < 1/2$,

$$\frac{1}{1-a} > 0.$$

Therefore, because $x \in I_0$,

$$\frac{a}{2}\lg B \leq x - \frac{1}{1-a} \leq \lg B. \tag{6}$$

Because $x \leq \lg B$ and from Definition 10, $\beta(x) = 0$. Observe that (6) is equivalent to (5) when $\beta(x) = 0$. Therefore, (5) holds in the base case.

Induction step: Assume that (5) holds for the nth interval I_n. We will show that (5) also holds for the $(n+1)$st interval I_{n+1}, i.e., when

$$x \in I_{n+1} = \left[\frac{\lg B}{2a^n} + \frac{1}{1-a}, \frac{\lg B - 1 - a - \cdots - a^n}{a^{n+1}}\right],$$

or equivalently when

$$\frac{\lg B}{2a^n} + \frac{1}{1-a} \leq x \leq \frac{\lg B - 1 - a - \cdots - a^n}{a^{n+1}}. \tag{7}$$

Multiplying by a and adding 1 to both sides of (7), we see that (7) is equivalent to

$$\frac{\lg B}{2a^{n-1}} + \frac{1}{1-a} \leq ax+1 \leq \frac{\lg B - 1 - a - \cdots - a^{n-1}}{a^n},$$

i.e.,

$$ax+1 \in I_n.$$

Thus, by induction (plugging $ax+1$ for x in (5)), we obtain

$$\frac{1}{2}\lg B \left(\frac{1}{a}\right)^{\beta(ax+1)-1} \leq ax+1-\frac{1}{1-a} \leq \lg B \left(\frac{1}{a}\right)^{\beta(ax+1)}.$$

Noticing that $\beta(ax+1) = \beta(x)-1$ by Definition 10 and

$$(ax+1)-\frac{1}{1-a} = a\left(x-\frac{1}{1-a}\right),$$

we establish

$$\frac{1}{2}\lg B \left(\frac{1}{a}\right)^{\beta(x)-2} \leq a\left(x-\frac{1}{1-a}\right) \leq \lg B \left(\frac{1}{a}\right)^{\beta(x)-1},$$

which is equivalent to (5) for $x \in I_{n+1}$.

We now prove the second part, that $\bigcup_{n=0}^{\infty} I_n$ covers the interval $[\lg B, \infty)$. This claim follows when $a < 1-2/\lg B$, which is guaranteed when $B > 16$. The claim follows because intervals overlap, i.e., the right endpoint of the I_n is between the left and right endpoints of the I_{n+1}, that is,

$$\frac{\lg B}{2a^n} + \frac{1}{1-a} \leq \frac{\lg B - 1 - a - \cdots - a^{n-1}}{a^n} \leq \frac{\lg B - 1 - a - \cdots - a^n}{a^{n+1}}.$$

We have now established that (5) holds for all $x > \lg B$.

We next show that (5) is equivalent to the lemma statement, i.e., (4). Taking inverses on both sides of (5), we have

$$2\frac{a^{\beta(x)-1}}{\lg B} \geq \frac{1}{x-\frac{1}{1-a}} \geq \frac{a^{\beta(x)}}{\lg B}$$

i.e.,

$$\frac{1}{a^2x-\frac{a^2}{1-a}} \geq \frac{a^{\beta(x)-2}}{\lg B} \geq \frac{1}{2ax-\frac{2a}{1-a}}.$$

Because $x > \lg B$ and $a < 1 - 2/\lg B$, we have $x > 2/(1-a)$, i.e., $a^2x/2 > a^2/(1-a)$. Therefore, the left side of the above inequality is less than $2/(a^2x)$. The right side is greater than $1/(2ax)$ because $2a/(1-a) > 0$. Thus, we prove the following

$$\frac{2}{a^2 x} \geq \frac{a^{\beta(x)-2}}{\lg B} \geq \frac{1}{2ax}$$

for all $x > \lg B$ as claimed. □

The following lemma gives the properties and the upper bound of $\delta(x)$.

Lemma 12 *The function* $\delta(x)$ *has the following properties:*

(1) If $\beta(x) = \beta(y)$, *then* $\delta(x) = \delta(y)$.

(2) For all $x > \lg B$,

$$(ax+1)\delta(ax+1) \leq ax\delta(x).$$

(3) For all $x > \lg B$,

$$\delta(x) \leq \exp\left[\frac{2}{a(1-a)\lg B}\right],$$

which is

$$1 + O\left(\frac{2}{a(1-a)\lg B}\right) = 1 + O\left(\frac{1}{\lg B}\right).$$

Proof. (1) This claim follows from Definition 10.
(2) This claim follows from Definition 10 of $\delta(x)$ and Lemma 11

$$\frac{a^{\beta(x)-2}}{\lg B} \geq \frac{1}{2ax}.$$

(3) Recall that from Definition 10, we have

$$\frac{\delta(x)}{\delta(ax+1)} = 1 + 2\frac{a^{\beta(x)-2}}{\lg B}$$

for all $x > \lg B$. Furthermore, because $1 + y < e^y$ is true for any $y > 0$, we bound the function $\delta(\cdot)$ as follows

$$\frac{\delta(x)}{\delta(ax+1)} \leq \exp\left[2\frac{a^{\beta(x)-2}}{\lg B}\right]. \tag{8}$$

For simplification, we define P_i be the polynomial $a^i x + a^{i-1} + \cdots + 1$. In the following, we show there exists some big integer n such that $P_{n+1} = a^{n+1}x + a^n + \cdots + 1 < \lg B$. First of all, because $a < 1$, a^n is arbitrary small when n goes to infinity. Thus, if n is big enough, then

$$a^{n+1}x < \frac{\lg B}{2} \tag{9}$$

for fixed number x. Second, for big $B > 16$, we have $1/(1-a) < (\lg B)/2$. Thus,

$$a^n + \cdots + a + 1 < \frac{1}{1-a} < \frac{\lg B}{2} \tag{10}$$

is true for all integer n. Therefore, combining both (9) and (10), we obtain that there exists some big integer n such that

$$P_{n+1} = a^{n+1}x + a^n + \cdots + 1 < \lg B,$$

which means, by Definition 10 of $\delta(x)$, $\delta(P_{n+1}) = \delta(a^{n+1}x + a^n + \cdots + 1) = 1$. Therefore, $\delta(x)$ can be expressed as the multiplication of $n+1$ items, i.e.,

$$\delta(x) = \frac{\delta(x)}{\delta(ax+1)} \frac{\delta(ax+1)}{\delta(a^2x+ax+1)} \cdots \frac{\delta(a^n x + a^{n-1} + \cdots + 1)}{\delta(a^{n+1}x + a^n + \cdots + 1)}$$

Using the term P_i in the above equation, we get the simplified version

$$\delta(x) = \prod_{i=0}^{n} \frac{\delta(P_i)}{\delta(P_{i+1})}. \tag{11}$$

To bound $\delta(x)$, we give the upper bound for $\delta(P_i)/\delta(P_{i+1})$ first. Notice that $P_{i+1} = aP_i + 1$, Replacing x by P_i in (8), we have the upper bound

$$\frac{\delta(P_i)}{\delta(P_{i+1})} \leq \exp\left[2\frac{a^{\beta(P_i)-2}}{\lg B}\right].$$

We claim that $\beta(P_i) = n+1-i$ for all $0 \leq i \leq n+1$. We prove this claim by induction. The base case is for P_{n+1}. From Definition 10 and $P_{n+1} < \lg B$, we have $\beta(P_{n+1}) = 0$. Assume the claim holds for some P_i. We prove the claim holds for P_{i-1}. Because $P_i = aP_{i-1} + 1$, we have $\beta(P_{i-1}) = \beta(P_i) + 1$ from Definition 10 of $\beta(x)$. Therefore, by induction, we obtain $\beta(P_{i-1}) = \beta(P_i) + 1 = n + 1 - i + 1 =$

$n+1-(i-1)$ as claimed. Thus, each of those items $\delta(P_i)/\delta(P_{i+1})$ has the upper bound

$$\exp\left[2\frac{a^{n-i-1}}{\lg B}\right].$$

Therefore, we obtain

$$\delta(x) \leq \exp\left[2\sum_{i=0}^{n}\frac{a^{n-i-1}}{\lg B}\right] = \exp\left[\frac{2}{\lg B}\sum_{i=0}^{n}a^{i-1}\right].$$

Because

$$\sum_{i=0}^{n}a^{i-1} < \sum_{i=0}^{\infty}a^{i-1} = \frac{1}{a(1-a)},$$

we prove that

$$\delta(x) \leq \exp\left[\frac{2}{a(1-a)\lg B}\right],$$

as claimed. □

Theorem 13 (Roundoff Error) *For $0 < a \leq b < 1$ and $a+b=1$, let*

$$\mathcal{F}(x) = \mathcal{F}(\lceil ax \rceil) + \mathcal{F}(\lfloor bx \rfloor) \text{ and } \mathcal{G}(x) = \mathcal{G}(ax) + \mathcal{G}(bx).$$

Then for $B > 8$, all $x > 1$, and constant c, we have

$$\mathcal{F}(x) \leq \mathcal{G}(x\delta(x)) \leq c\left[1+O\left(\frac{1}{\lg B}\right)\right]x + O(1).$$

Proof. First recall that $\mathcal{F}(x)$ and $\mathcal{G}(x)$ are monotonically increasing. Thus, from $\lceil ax \rceil \leq ax+1$ and $\lfloor bx \rfloor \leq bx$, we have

$$\mathcal{F}(x) \leq \mathcal{F}(ax+1) + \mathcal{F}(bx). \tag{12}$$

We prove $\mathcal{F}(x) \leq \mathcal{G}(x\delta(x))$ inductively. The base case is when $1 < x \leq \lg B$, where $\delta(x) = 1$ from Definition 10 and $\mathcal{F}(x) = \mathcal{G}(x)$. Thus, $\mathcal{F}(x) \leq \mathcal{G}(x\delta(x))$ is true when $1 < x \leq \lg B$.

Assuming $\mathcal{F}(x) \leq \mathcal{G}(x\delta(x))$ is true for $1 < x \leq t$, we prove it is true for $1 < x \leq (t-1)/b$. Noticing that $(t-1)/b \leq \min\{t/b, (t-1)/a\}$ (because $b \geq a$), we

have $ax+1 \le t$ and $bx \le t$ for all $1 < x \le (t-1)/b$. Thus, by assumption and (12), we obtain

$$\mathcal{F}(x) \le \mathcal{G}((ax+1)\delta(ax+1)) + \mathcal{G}(bx\delta(bx)), \qquad 1 < x \le (t-1)/b. \tag{13}$$

From Condition (2) in Lemma 12 and $\delta(bx) \le \delta(x)$, we obtain

$$\mathcal{G}((ax+1)\delta(ax+1)) \le \mathcal{G}(ax\delta(x)) \quad \text{and} \quad \mathcal{G}(bx\delta(bx)) \le \mathcal{G}(bx\delta(x)). \tag{14}$$

Pluging (14) into (13), we derive that

$$\mathcal{F}(x) \le \mathcal{G}(ax\delta(x)) + \mathcal{G}(bx\delta(x)) = \mathcal{G}(x\delta(x)), \qquad 1 < x \le (t-1)/b.$$

Therefore, after two inductive steps, it is true for

$$1 < x \le \frac{t-1-b}{b^2},$$

and after n inductive steps, it is true for all

$$1 < x \le \frac{t-1-b-\cdots-b^{n-1}}{b^n} = \frac{t-(1-b^n)/(1-b)}{b^n}.$$

Therefore, as long as $t > 1/(1-b) = 1/a$, we have $\mathcal{F}(x) \le \mathcal{G}(x\delta(x))$ for all $x > 1$. Thus, we need $\lg B > 1/a$, which holds when $B > 8$ and $a > 1/3$.

Furthermore, if $\mathcal{G}(x) \le cx + O(1)$, then by Condition (3) in Lemma 12, we obtain the following:

$$\mathcal{F}(x) \le \mathcal{G}(x\delta(x)) \le cx\delta(x) + O(1) \le c[1+O(1/\lg B)]x + O(1).$$

□

Bounding the Path-Length Function

We now determine the constant in the search cost $O(\log_B N)$, for given values of a and b. To do so, we assume

$$a = \frac{1}{q^k} \quad \text{and} \quad b = \frac{1}{q^m}, \tag{15}$$

for positive real number $q > 1$ and relatively prime integers m and k. Plugging (15) into $a+b=1$, we obtain $1/q^k + 1/q^m = 1$. Define

$$n = k - m\,. \tag{16}$$

Observe that because $k > m$ (since $a < b$), n is positive. We now have the simplified formula

$$q^k = q^n + 1\,. \tag{17}$$

The rationale behind this assumption is that this additional structure helps us in the analysis while still being dense; that is, for any given a and b satisfying $a+b=1$, we can find a' and b' defined as (15) that are arbitrary close to a and b. Because there exits a real number r such that $a = b^r$, we choose rational number k/m, $(k,m) = 1$ as close as desired to r. Let $q = b^{-1/m}$. Then $a' = 1/q^k$ and $b' = 1/q^m$. We call such an (a,b) pair a *twin power pair*.

As before we analyze $\mathcal{V}(x)$ first. We ignore the roundoff based on Theorem 13. Furthermore, we normalize the range for which $\mathcal{V}(x) = 1$ by introducing a function

$$H(x) = \begin{cases} H(ax) + H(bx)\,, & x > 1; \\ 1\,, & 0 < x \leq 1. \end{cases} \tag{18}$$

Note that $\mathcal{V}(x \lg B) \leq H(x\delta(x \lg B))$ by Theorem 13.

First we state a primary lemma of the subsection, which we prove later.

Lemma 14 *Let $(1/q^k, 1/q^m)$ be a twin power pair, and let $n = k - m$. Then for any constant $\varepsilon > 0$ and*

$$c_1 = \left(\sum_{i=1}^{n} q^{-i} + \sum_{i=n+1}^{k} q^{k-i}\right) / \left(kq^{k-1} - nq^{n-1}\right),$$

when $x \geq O(k/\varepsilon)$ we have

$$H(x) \leq (c_1 + \varepsilon) q^k x + O(1).$$

Corollary 15 *For any constant $\varepsilon > 0$, the number $\mathcal{V}(x)$ of recursive subtrees on a root-to-leaf path is bounded by*

$$(c_1 + \varepsilon) q^k \log_B N + O(1),$$

when $N \geq B^{O(k/\varepsilon)}$.

We obtain the main upper-bound result by showing that $c_1 q^k \approx \lg e$ for some twin power pair.

Theorem 16 (Path-Length Cost) *For any constant $\varepsilon > 0$, the number of recursive subtrees on a root-to-leaf path is*

$$(\lg e + \varepsilon)\log_B N + O(1) \approx 1.443 \log_B N + O(1),$$

as the split factor $a = 1/2 - \xi$ approaches $1/2$.

Proof. Choose the twin power pair $a = 1/q^k$ and $b = 1/q^{k-1}$ such that

$$\frac{1}{q^k} + \frac{1}{q^{k-1}} = 1,$$

which is equivalent to

$$q^k = q + 1.$$

The approximate solution for the above equation is

$$q \approx 1 + \frac{\ln 2}{k},$$

for $k \to \infty$. Therefore, we have

$$a = \frac{1}{1+q} \approx \frac{1}{2 + \ln 2/k}. \tag{19}$$

From Lemma 14, for $m = k-1$ (and therefore $n = 1$), we have

$$c_1 = \left(q^{-1} + \sum_{i=2}^{k} q^{k-i}\right) / \left(kq^{k-1} - 1\right) = \frac{q^k - 1}{(q-1)(kq^k - q)} .$$

Thus, for large k, we obtain

$$c_1 q^k = \frac{q^k - 1}{q - 1} \frac{1}{k - \frac{1}{q^{k-1}}} \xrightarrow{k \to \infty} \frac{1}{\ln 2} = \lg e.$$

That is, for a given $\varepsilon/2 > 0$, we can choose a big constant k_ε such that

$$c_1 q^k \leq \lg e + \varepsilon/2, \tag{20}$$

for all $k \geq k_\varepsilon$.

From Corollary 15, for a given $\varepsilon/8 > 0$ and the above constant k_ε, we can choose big constant $N_{\varepsilon,k}$ such that

$$\mathcal{V}(x) \leq (c_1 + \varepsilon/8) q^k \log_B N + O(1), \tag{21}$$

for all $N \geq N_{\varepsilon,k}$. Plugging (20) into (21) and noticing that $q^k = 1/a < 4$, we obtain

$$\mathcal{V}(x) \leq (\lg e + \varepsilon) \log_B N + O(1) \approx 1.443 \log_B N + O(1)$$

as claimed.

Noticing that for big $k \geq k_\varepsilon$, we see that the split factor a approaches $1/2$ by (19). In particular, as long as that

$$\xi \leq \frac{1}{2} - \frac{1}{2 + \ln 2/k_\varepsilon} = \frac{\ln 2}{4k_\varepsilon + \ln 4},$$

it suffices that the split factor $a = 1/2 - \xi$. □

To complete the proof of Lemma 14, we establish some properties of $H(x)$. Since $H(x)$ is monotonically increasing, we can bound the value $H(x)/x$ for $q^i \leq x \leq q^{i+1}$ as follows:

$$\frac{H(q^i)}{q^{i+1}} \leq \frac{H(x)}{x} \leq \frac{H(q^{i+1})}{q^i}. \tag{22}$$

Let H_{min} be the lower bound and H_{max} be the upper bound of $H(q^i)/q^i$, when i is larger than a given integer j. Noticing that the left part in Inequality (22) is $H(q^i)/q^{i+1} \geq H_{min}/q$ and the right part in Inequality (22) is $H(q^{i+1})/q^i \leq qH_{max}$, we obtain

$$\frac{H_{min}}{q} \leq \frac{H(x)}{x} \leq qH_{max},$$

when x is bigger than q^j.

We give the recurrence of $H(\cdot)$. From (18), we have that for $i \geq 0$,

$$H(q^{i+1}) = H(aq^{i+1}) + H(bq^{i+1}). \tag{23}$$

Plugging (15) into (23) and since $n = k - m$, we obtain

$$H(q^{i+1}) = H(q^{i-k+1}) + H(q^{i+n-k+1}). \tag{24}$$

For the sake of notational simplicity, we denote $\alpha_i = H(q^{i-k+1})$. Therefore, (24) is equivalent to

$$\alpha_{i+k} = \alpha_{i+n} + \alpha_i . \quad (25)$$

We define the characteristic polynomial function of Recurrence (25) as $w(x) = x^k - x^n - 1$. Let $r_1, r_2, \ldots, r_k$ be the (possibly complex) roots of $w(x)$. We claim below that these roots are all unique.

The following four lemmas supply basic mathematical knowledge behind the proof of Lemma 14.

Lemma 17 *The k roots of* $w(x) = x^k - x^n - 1$ *are unique, when k and n are relatively prime integers such that* $1 \leq n < k$.

Proof. We prove this lemma by contradiction. If a root r of $w(x)$ is not unique, then $(x-r)^2$ is a factor of $w(x)$, and $x - r$ is a factor of $w'(x) = kx^{n-1}(x^{k-n} - n/k)$. Thus, r is either 0 or a root of $x^{k-n} - n/k$. But 0 is not a root of $w(x)$. Therefore,

$$r^{k-n} = n/k, \quad (26)$$

which means $|r| < 1$ (because $n < k$).

On the other hand, because r is a root of $w(x)$, $w(r) = r^n(r^{k-n} - 1) - 1 = 0$. Plugging (26) into $w(r) = 0$, we obtain $r^n = k/(n-k)$, which means $|r| > 1$ (because $|k| > |k-n|$). This is the contradiction. Therefore, every root of $w(x)$ is unique. □

Because $w'(x) = kx^{k-1} - nx^{n-1} > 0$ when $x > 1$ and q is a root of $w(x)$ greater than 1 (see Equation (17)), there is one unique real root $q > 1$ of $w(x)$. Without loss of generality, let $r_1 = q$.

We now show that if the k roots of the characteristic polynomial function of a series are unique, then the series in question is a linear combination of power series $\{r_j^i\}$ of the roots.

Lemma 18 *Consider a series* $\{\alpha_i\}$ *satisfying* $\alpha_{k+s} = \sum_{i=0}^{k-1} d_i \alpha_{i+s}$ *for complex numbers* d_i *and any integer s, and let* $r_1, r_2, \ldots, r_k$ *be the k unique roots of the characteristic function* $g(x) = x^k - \sum_{i=0}^{k-1} d_i x^i$ *for the series* $\{\alpha_i\}$. *Then there exists complex numbers* $c_1, c_2, \ldots, c_k$ *such that for all i,*

$$\alpha_i = \sum_{j=1}^{k} c_j r_j^i . \quad (27)$$

Proof. First we show that we can find $c_1, c_2, \ldots, c_k$ such that for the base values of α_i, $\alpha_i = \sum_{j=1}^{k} c_j r_j^i$ for all $i = 0, \ldots, k-1$. This can be derived by observing that the determinant of the Vandermonde matrix

$$V = \begin{pmatrix} 1 & r_1 & \cdots & r_1^{k-1} \\ 1 & r_2 & \cdots & r_2^{k-1} \\ \cdots & \cdots & \cdots & \cdots \\ 1 & r_k & \cdots & r_k^{k-1} \end{pmatrix}$$

is nonzero, and that $c_1, c_2, \ldots, c_k$ are the solution of the system of linear equations

$$(\alpha_0, \alpha_1, \ldots, \alpha_{k-1}) = (c_1, c_2, \ldots, c_k)V.$$

Now we show that for all $i \geq 0$,

$$\alpha_i = \sum_{j=1}^{k} c_j r_j^i.$$

Define

$$\beta_i = \sum_{j=1}^{k} c_j r_j^i.$$

We show that $\{\alpha_i\}$ and $\{\beta_i\}$ are the same recursive series. We know that $\beta_i = \alpha_i$ when $0 \leq i \leq k-1$. Because $r_1, r_2, \ldots, r_k$ are the k unique roots of the characteristic function $g(x)$, we know that the power series $\{r_j^i\}$ satisfies the same recursive formula as $\{\alpha_i\}$. Thus $\{\beta_i\}$ satisfies the same recursive formula (for all $s \geq 0$, $b_{k+s} = \sum_{i=0}^{k-1} d_i b_{i+s}$) by linearity. Now observe that the k base values together with the inductive formula uniquely determine the series and hence $\alpha_i = \beta_i$ for all $i \geq 0$. □

Hence we can solve Recurrence (25) by finding c_i that satisfy $\alpha_i = \sum_{j=1}^{k} c_j r_j^i$ for $i = 0, \ldots, k-1$. The base cases of $\{\alpha_i\}_{i=0}^{k-1}$ are determined by the original definition of $\alpha_i = H(q^{i-k+1})$. Because $0 < q^{i-k+1} < 1$ for $i = 0, \ldots, k-1$, we obtain $H(q^{i-k+1}) = \alpha_i = 1$.

Lemma 19 *The dominant root (i.e., the root with the largest absolute value) for $w(x) = x^k - x^n - 1$ is $r_1 = q$. All other roots $r_2, r_3, \ldots, r_k$ have absolute value less than q.*

Proof. We first show that all other roots have magnitude less than q. Suppose that the magnitude of a root r_j (other than r_1) is $|r_j| = d$. We show that $d \leq q$. Since r_j is a root we have

$$1 = |(r_j^{k-n} - 1)\, r_j^n| = |r_j^{k-n} - 1||r_j^n| \geq (|r_j^{k-n}| - 1)\, d^n = d^k - d^n, \tag{28}$$

which means $w(d) = d^k - d^n - 1 \leq 0$. Because $w(q) = 0$ and $w'(x) = kx^{k-1} - nx^{n-1} > 0$ when $x \geq q > 1$, we obtain $w(x) > 0$ for all real $x > q$. Therefore, $d \leq q$, i.e., no root has magnitude strictly greater than q.

Now we prove by contradiction that $d \neq q$. Assume that $d = q$. Then, (28) becomes an equation, since $1 = d^k - d^n$ by (17). Thus,

$$|r_j^m - 1| = |r_j^m| - 1.$$

From the triangle inequality it follows that r_j^m is a real number. Therefore, we have $r_j^m = q^m$. Thus, for some integer $1 \leq s \leq m-1$, we have

$$r_j = qe^{2\pi s\sqrt{-1}/m}.$$

However, because m and n are relatively prime,

$$r_j^n = q^n e^{2\pi sn\sqrt{-1}/m} \neq q^n.$$

Therefore, $r_j^n(r_j^m - 1) \neq q^n(q^m - 1) = 1$, i.e., $w(r_j) \neq 0$. This is contradiction because r_j is a root of $w(x)$. □

In the following lemma, we calculate the coefficient c_1 for the dominant root $r_1 = q$ using the inverse of a Vandermonde matrix.

Lemma 20 *The coefficient c_1 in Lemma 18 is*

$$\left(\sum_{i=1}^{n} q^{-i} + \sum_{i=n+1}^{k} q^{k-i}\right) / (kq^{k-1} - nq^{n-1}).$$

Proof. We first give more notation. Let t and s be positive integers such that $1 \leq t, s \leq k$. We define $S_{t,s}$ as the sum of the products of t different roots not including r_s, that is,

$$S_{t,s} = \sum_{i_1 < i_2 < \ldots < i_t \in \{1,2,\ldots,k\} - \{s\}} r_{i_1} r_{i_2} \ldots r_{i_t}\,. \tag{29}$$

We define

$$S_{0,1} = 1\,, \tag{30}$$

and

$$S_{k,1} = 0\,. \tag{31}$$

We first give and solve the recurrence for $S_{t,1}$. We denote the coefficient of x^{t-1} in $w(x) = x^k - x^n - 1 = \prod_{i=1}^{k}(x - r_i)$ as $[\![x^{t-1}]\!]_{w(x)}$. Thus, we have the well known equation:

$$\sum_{i_1<i_2<\ldots<i_t \in \{1,2,\ldots,k\}} r_{i_1} r_{i_2} \ldots r_{i_t} = (-1)^t [\![x^{k-t}]\!]_{w(x)}. \tag{32}$$

Each product of roots in the summation in (32) either includes r_1 $(= q)$ or it does not, i.e.,

$$\sum_{i_1<i_2<\ldots<i_t \in \{1,2,\ldots,k\}} r_{i_1} r_{i_2} \ldots r_{i_t} = S_{t,1} + qS_{t-1,1}. \tag{33}$$

Thus, from (32) and (33) we obtain the recurrence

$$S_{t,1} + qS_{t-1,1} = (-1)^t [\![x^{k-t}]\!]_{w(x)}. \tag{34}$$

Because coefficients in $w(x)$ are 0 except for $[\![x^k]\!]_{w(x)} = 1$ and $[\![x^n]\!]_{w(x)} = [\![x^0]\!]_{w(x)} = -1$, we divide Recurrence (34) into two parts and solve each separately. Recall from (16) that $m = k - n$. The first part is when $t \in [1, m-1]$ and the second part is when $t \in [m, k-1]$. (Thus, when $t = m$, we need to confirm that the solution in the first part matches that in the second part.)

We solve the first part when $t \in [1, m-1]$. The base case is $t = 1$, that is,

$$S_{1,1} + qS_{0,1} = [\![x^{k-1}]\!]_{w(x)} = 0\,. \tag{35}$$

Observe that by (29) and (33), we have

$$\sum_{1 \le i \le k} r_i = S_{1,1} + qS_{0,1} \quad \text{and} \quad \sum_{2 \le i \le k} r_i = S_{1,1}. \tag{36}$$

Thus, from (36), we confirm that $S_{0,1} = 1$, and therefore from (35), we obtain

$$S_{1,1} = -q. \tag{37}$$

Because from (34),

$$S_{t,1}+qS_{t-1,1}=0 \quad (1\le t\le m-1)\,, \tag{38}$$

we also obtain, from (37) and (38),

$$S_{t,1}=(-q)^t. \tag{39}$$

We now solve the second part when $t\in[m,k-1]$. We start from k, that is,

$$S_{k,1}+qS_{k-1,1}=(-1)^k[\![x^0]\!]_{w(x)}=(-1)^{k-1}\,. \tag{40}$$

Observe that by (29) and (33), we have

$$r_1r_2\ldots r_k=S_{k,1}+qS_{k-1,1} \quad \text{and} \quad r_2r_3\ldots r_k=S_{k-1,1}. \tag{41}$$

From (41), we confirm that $S_{k,1}=0$, and therefore from (40), we obtain $S_{k-1,1}=(-1)^{k-1}/q$. Because by (34),

$$S_{t+1,1}+qS_{t,1}=0 \quad (m\le t\le k-1),$$

we obtain

$$S_{t,1}=(-1)^tq^{t-k}. \tag{42}$$

We now examine the special case where $t=m$ and $[\![x^n]\!]_{w(x)}=-1$, that is,

$$\begin{aligned} S_{m,1}+qS_{m-1,1} &= (-1)^m[\![x^n]\!]_{w(x)} \\ &= (-1)^{m+1}\,. \end{aligned} \tag{43}$$

We solved for all $S_{t,1}$ without using (43). We now confirm that our solution is consistent with (43). Notice that we get the solution in the first part, $S_{m-1,1}=(-q)^{m-1}$, and the solution in the second part, $S_{m,1}=(-1)^mq^{-n}$. In the following, we verify the solutions of $S_{m-1,1}$ and $S_{m,1}$ satisfy (43). Plugging

$$S_{m-1,1}=(-q)^{m-1} \quad \text{and} \quad S_{m,1}=(-1)^mq^{-n}$$

into (43), we obtain

$$\begin{aligned} S_{m,1}+qS_{m-1,1} &= (-1)^mq^{-n}+q(-q)^{m-1} \\ &= (-1)^m\frac{1-q^{n+m}}{q^n} \end{aligned}$$

Because q is a root of $w(x)$, i.e.,

$$q^k = q^{m+n} = q^n + 1\,,$$

we confirm (43).

In summary, for all $1 \leq t \leq k-1$, we have

$$S_{t,1} = \begin{cases} (-q)^t, & \text{if } 1 \leq t \leq m-1; \\ (-1)^t q^{t-k}, & \text{if } m \leq t \leq k-1. \end{cases} \tag{44}$$

We now give even more notation. Define

$$g(x) = \prod_{i=2}^{k} (x - r_i). \tag{45}$$

We have $g(r_1) = g(q) = w'(q)$, because

$$w'(x) = \frac{d}{dx}\left[\prod_{i=1}^{k}(x - r_i)\right] = \sum_{j=1}^{k} \prod_{i=1, i\neq j}^{k} (x - r_i)$$

is a sum of k terms, but $k-1$ of these are 0 when $x = r_1 = q$. Thus, we obtain

$$(-1)^{k-1} g(r_1) = (-1)^{k-1}(k r_1^{k-1} - n r_1^{n-1}) = \prod_{i=2}^{k}(r_i - r_1). \tag{46}$$

Now we are ready to calculate the value of c_1. To do so, we define the Vandermonde matrix V:

$$V = \begin{pmatrix} 1 & r_1 & \cdots & r_1^{k-1} \\ 1 & r_2 & \cdots & r_2^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & r_k & \cdots & r_k^{k-1} \end{pmatrix}.$$

Recall that (27) can be expressed as

$$(c_1, c_2, \ldots, c_k)V = (\alpha_0, \alpha_1, \ldots, \alpha_{k-1})\,.$$

Recall also that

$$\alpha_i = H(q^{i-k+1}) = 1 \quad (0 \leq i \leq k-1)$$

(because $q^{i-k+1} < 1$). Thus,

$$(c_1, \ldots, c_k) = (1, 1, \ldots, 1) V^{-1}, \tag{47}$$

i.e., c_1 can be calculated from the inverse matrix V^{-1}.

In order to calculate V^{-1}, we first present the well known result on how to calculate the determinant $|V|$ of Vandermonde matrix V.

$$|V| = \begin{vmatrix} 1 & r_1 & \cdots & r_1^{k-1} \\ 1 & r_2 & \cdots & r_2^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & r_k & \cdots & r_k^{k-1} \end{vmatrix} = \prod_{1 \le s < t \le k} (r_t - r_s). \tag{48}$$

We now give the inverse of V. Let $A_{i,j}$ be the submatrix of the transpose of the Vandermonde matrix V with the ith column and jth row removed, that is,

$$A_{i,j} = \begin{pmatrix} 1 & 1 & \cdots & 1 & 1 & \cdots & 1 \\ r_1 & r_2 & \cdots & r_{i-1} & r_{i+1} & \cdots & r_k \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ r_1^{j-1} & r_2^{j-1} & \cdots & r_{i-1}^{j-1} & r_{i+1}^{j-1} & \cdots & r_k^{j-1} \\ & & & & & & \\ r_1^{j+1} & r_2^{j+1} & \cdots & r_{i-1}^{j+1} & r_{i+1}^{j+1} & \cdots & r_k^{j+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ r_1^{k-1} & r_2^{k-1} & \cdots & r_{i-1}^{k-1} & r_{i+1}^{k-1} & \cdots & r_k^{k-1} \end{pmatrix}.$$

Thus, V^{-1} can be represented by the determinants of $A_{i,j}$ and V, i.e.,

$$\begin{aligned} V^{-1} &= \frac{1}{|V|} \begin{pmatrix} (-1)^{1+1}|A_{1,1}| & \cdots & (-1)^{k+1}|A_{k,1}| \\ (-1)^{1+2}|A_{1,2}| & \cdots & (-1)^{k+2}|A_{k,2}| \\ \vdots & \ddots & \vdots \\ (-1)^{1+k}|A_{1,k}| & \cdots & (-1)^{k+k}|A_{k,k}| \end{pmatrix} \\ &= \left(\prod_{1 \le s < t \le k} \frac{1}{r_t - r_s} \right) \left\{ (-1)^{i+j} |A_{i,j}| \right\}_{i,j}. \end{aligned}$$

Thus, from (47), c_1 is the sum of the first column of inverse matrix V^{-1}, that is,

$$c_1 = \left(\prod_{1 \le s < t \le k} \frac{1}{r_t - r_s} \right) \left(\sum_{j=1}^{k} (-1)^{1+j} |A_{1,j}| \right). \tag{49}$$

To calculate c_1, we first find $|A_{1,j}|$, which is given by the following claim:

Claim 21

$$|A_{1,j}| = S_{k-j,1} \prod_{2 \le s < t \le k} (r_t - r_s).$$

Proof. When $j = 1$,

$$|A_{1,1}| = \begin{vmatrix} r_2 & \cdots & r_k \\ \vdots & \ddots & \vdots \\ r_2^{k-1} & \cdots & r_k^{k-1} \end{vmatrix}.$$

By moving the common factors $r_2, \ldots, r_k$ out, we obtain

$$|A_{1,1}| = r_2 \cdots r_k \begin{vmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ r_2^{k-2} & \cdots & r_k^{k-2} \end{vmatrix},$$

where the matrix is the transpose of Vandermonde matrix of size $k-1$. Thus, from (29) and (48), we obtain

$$|A_{1,1}| = r_2 \cdots r_k \prod_{2 \le s < t \le k} (r_t - r_s) = S_{k-1,1} \prod_{2 \le s < t \le k} (r_t - r_s).$$

The case when $j \ge 2$ is more complicated than that $j = 1$. In the following, we only consider $j = 2$ because the other cases are analogous.

To solve $|A_{1,2}|$, we first perform matrix operations so that the first column becomes $\begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$. Recall that

$$|A_{1,2}| = \begin{vmatrix} 1 & 1 & \cdots & 1 \\ r_2^2 & r_3^2 & \cdots & r_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ r_2^{k-1} & r_3^{k-1} & \cdots & r_k^{k-1} \end{vmatrix}.$$

Beginning from the second row, we multiply each row by $-r_2$ and add it to the next row.

$$|A_{1,2}| = \begin{vmatrix} 1 & 1 & 1 & \cdots & 1 \\ r_2^2 & r_3^2 & r_4^2 & \cdots & r_k^2 \\ 0 & r_3^2(r_3-r_2) & r_4^2(r_4-r_2) & \cdots & r_k^2(r_k-r_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & r_3^{k-2}(r_3-r_2) & r_4^{k-2}(r_4-r_2) & \cdots & r_k^{k-2}(r_k-r_2) \end{vmatrix}.$$

For the second row, we multiply the first row by $-r_2^2$ and add it to the second row.

$$|A_{1,2}| = \begin{vmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & r_3^2-r_2^2 & r_4^2-r_2^2 & \cdots & r_k^2-r_2^2 \\ 0 & r_3^2(r_3-r_2) & r_4^2(r_4-r_2) & \cdots & r_k^2(r_k-r_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & r_3^{k-2}(r_3-r_2) & r_4^{k-2}(r_4-r_2) & \cdots & r_k^{k-2}(r_k-r_2) \end{vmatrix}.$$

In this way, we reduce the dimension of $|A_{1,2}|$ to $k-2$, i.e.,

$$|A_{1,2}| = \begin{vmatrix} r_3^2-r_2^2 & r_4^2-r_2^2 & \cdots & r_k^2-r_2^2 \\ r_3^2(r_3-r_2) & r_4^2(r_4-r_2) & \cdots & r_k^2(r_k-r_2) \\ \vdots & \vdots & \ddots & \vdots \\ r_3^{k-2}(r_3-r_2) & r_4^{k-2}(r_4-r_2) & \cdots & r_k^{k-2}(r_k-r_2) \end{vmatrix}.$$

By moving out the common factors $r_3-r_2, \ldots, r_k-r_2$ in each column, we obtain:

$$|A_{1,2}| = \prod_{i=3}^{k}(r_i-r_2) \begin{vmatrix} r_3+r_2 & r_4+r_2 & \cdots & r_k+r_2 \\ r_3^2 & r_4^2 & \cdots & r_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ r_3^{k-2} & r_4^{k-2} & \cdots & r_k^{k-2} \end{vmatrix}.$$

Now by splitting the first row, we obtain:

$$|A_{1,2}| = \prod_{i=3}^{k}(r_i-r_2) \left\{ \begin{vmatrix} r_3 & \cdots & r_k \\ r_3^2 & \cdots & r_k^2 \\ \vdots & \ddots & \vdots \\ r_3^{k-2} & \cdots & r_k^{k-2} \end{vmatrix} + \begin{vmatrix} r_2 & \cdots & r_2 \\ r_3^2 & \cdots & r_k^2 \\ \vdots & \ddots & \vdots \\ r_3^{k-2} & \cdots & r_k^{k-2} \end{vmatrix} \right\}. \tag{50}$$

After moving out the common factors $r_3, \ldots, r_k$, the first term in (50) is a Vandermonde matrix of size $k-2$. For the second term in (50), we move out the common factor r_2 in the top row. Thus, using (48) we have

$$
\begin{aligned}
|A_{1,2}| &= \prod_{i=3}^{k}(r_i - r_2) r_3 \cdots r_k \prod_{3 \le s < t \le k} (r_t - r_s) \\
&\quad + \prod_{i=3}^{k}(r_i - r_2) r_2 \begin{vmatrix} 1 & 1 & \cdots & 1 \\ r_3^2 & r_4^2 & \cdots & r_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ r_3^{k-2} & r_4^{k-2} & \cdots & r_k^{k-2} \end{vmatrix} \\
&= r_3 \cdots r_k \prod_{2 \le s < t \le k} (r_t - r_s) + r_2 \prod_{i=3}^{k} (r_i - r_2) \begin{vmatrix} 1 & 1 & \cdots & 1 \\ r_3^2 & r_4^2 & \cdots & r_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ r_3^{k-2} & r_4^{k-2} & \cdots & r_k^{k-2} \end{vmatrix}. \qquad (51)
\end{aligned}
$$

Notice that the determinant in (51) is a form of $A_{1,2}$ of size $k-2$. By the same method, we compute the determinant in (51) as

$$
\begin{aligned}
\begin{vmatrix} 1 & 1 & \cdots & 1 \\ r_3^2 & r_4^2 & \cdots & r_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ r_3^{k-2} & r_4^{k-2} & \cdots & r_k^{k-2} \end{vmatrix} &= r_4 \cdots r_k \prod_{3 \le s < t \le k} (r_t - r_s) \\
&\quad + r_3 \prod_{i=4}^{k} (r_i - r_3) \begin{vmatrix} 1 & \cdots & 1 \\ r_4^2 & \cdots & r_k^2 \\ \vdots & \ddots & \vdots \\ r_4^{k-2} & \cdots & r_k^{k-2} \end{vmatrix}. \qquad (52)
\end{aligned}
$$

Thus, by plugging (52) into (51) we obtain

$$
\begin{aligned}
|A_{1,2}| &= (r_3 \cdots r_k + r_2 r_4 \cdots r_k) \prod_{2 \le s < t \le k} (r_t - r_s) \\
&\quad + r_2 r_3 \prod_{i=3}^{k} (r_i - r_2) \prod_{i=4}^{k} (r_i - r_3) \begin{vmatrix} 1 & \cdots & 1 \\ r_4^2 & \cdots & r_k^2 \\ \vdots & \ddots & \vdots \\ r_4^{k-2} & \cdots & r_k^{k-2} \end{vmatrix}.
\end{aligned}
$$

With one more recursion, we obtain

$$\begin{aligned}|A_{1,2}| &= (r_3\cdots r_k + r_2r_4\cdots r_k + r_2r_3r_5\cdots r_k)\prod_{2\le s<t\le k}(r_t - r_s)\\ &+ r_2r_3r_4\prod_{i=3}^{k}(r_i - r_2)\prod_{i=4}^{k}(r_i - r_3)\prod_{i=5}^{k}(r_i - r_4)\begin{vmatrix} 1 & \cdots & 1\\ r_5^2 & \cdots & r_k^2\\ \vdots & \ddots & \vdots\\ r_5^{k-2} & \cdots & r_k^{k-2}\end{vmatrix}.\end{aligned}$$

Repeating recursive steps and recalling that $S_{k-2,1} = r_3r_4\cdots r_k + r_2r_4\cdots r_k + \cdots + r_2r_3\cdots r_{k-1}$ from the definition of $S_{t,s}$ in (29), we obtain

$$|A_{1,2}| = S_{k-2,1}\prod_{2\le s<t\le k}(r_t - r_s).$$

We thus establish the claim. □

By combining Claim 21 and (49) we obtain

$$c_1 = \left(\prod_{1\le i<j\le k}(r_j - r_i)^{-1}\right)\left(\sum_{i=1}^{k}(-1)^{i+1}S_{k-i,1}\prod_{2\le s<t\le k}(r_t - r_s)\right). \quad (53)$$

Multiplying through and separating two cases of $S_{i,1}$ in (44), we obtain

$$c_1 = \prod_{2\le j\le k}(r_j - r_1)^{-1}\left(\sum_{i=1}^{n} + \sum_{i=n+1}^{k}\right)(-1)^{i+1}S_{k-i,1}. \quad (54)$$

Plugging (44) into (54), we have

$$c_1 = \prod_{2\le j\le k}(r_j - r_1)^{-1}\left[\sum_{i=1}^{n}(-1)^{k+1}q^{-i} + \sum_{i=n+1}^{k}(-1)^{k+1}q^{k-i}\right]. \quad (55)$$

Plugging (46) into (55), we solve for c_1:

$$c_1 = \left(\sum_{i=1}^{n}q^{-i} + \sum_{i=n+1}^{k}q^{k-i}\right)/(kq^{k-1} - nq^{n-1}). \quad (56)$$

Thus, the value of c_1 is as claimed in Lemma 14. □

After establishing the properties of $H(x)$, we give the proof of Lemma 14.

PROOF OF LEMMA 14: To complete the proof we only need to show that

$$H(x) \le (c_1 + \varepsilon) q^k x + O(1),$$

when $x \ge O(k/\varepsilon)$.

Observe that $H(x)$ is monotonically increasing and for each $x > 1$, we have $x \le q^{\lceil \log_q x \rceil} \le qx$. Thus, we bound $H(x)$ as follows:

$$H(x) \le H(q^{\lceil \log_q x \rceil}) = \alpha_{\lceil \log_q x \rceil + k - 1}, \tag{57}$$

where the second equality is the definition of α_i. We denote $\lceil \log_q x \rceil + k - 1$ as i to simplify notation in the rest of the proof. Recall that $\alpha_i = \sum_{j=1}^{k} c_j r_j^i$ and that $r_1 = q$ is the dominant root. Thus, we have

$$\frac{\alpha_i}{q^i} = c_1 + \sum_{j=2}^{k} c_j \left(\frac{r_j}{q} \right)^i. \tag{58}$$

Because r_1 is the dominant root and the other roots have absolute value less than 1, we have

$$\sum_{j=2}^{k} c_j \left(\frac{r_j}{q} \right)^i \le O\left(\frac{k}{q^i} \right).$$

Because $i = \lceil \log_q x \rceil + k - 1$, we have $q^i > x$. Thus, for any $\varepsilon > 0$, we can choose $x \ge O(k/\varepsilon)$ such that the last term in (58) is arbitrary small, that is,

$$\sum_{j=2}^{k} c_j \left(\frac{r_j}{q} \right)^i \le O\left(\frac{k}{x} \right) \le \varepsilon.$$

Therefore, we obtain $\alpha_i = (c_1 + \varepsilon) q^i$. Combining with (57), we have

$$H(x) \le (c_1 + \varepsilon) q^{\lceil \log_q x \rceil + k - 1}. \tag{59}$$

Finally, plugging $q^{\lceil \log_q x \rceil} \le qx$ into (59), we obtain, for $x \ge O(k/\varepsilon)$,

$$H(x) \le (c_1 + \varepsilon) q^k x$$

as claimed. □

Bounding the Block-Boundary Crossing Function

We now give the memory-transfer cost from block-boundary crossings, and we show that it is dominated by the the memory-transfer cost from the path length. We consider the case when $a \geq 1/4$, which includes the best layouts. Using similar reasoning for computing the path-length cost, we obtain the following theorem:

Theorem 22 (Boundary Crossing Cost) *The expected number of block-boundary induced memory transfers $\mathcal{C}(x)$ on a search is at most $O(\lg\lg B/\lg B)\log_B x$ when $1/4 \leq a < 1/2$.*

Proof. The idea to bound $\mathcal{C}(x)$ is the same as that in bounding the path-length cost. That is, we solve the same Recurrence (25) except for the base case α_i ($0 \leq i \leq k-1$), which from (3) is

$$\frac{2^{q^{i-k+1}\lg B}-2}{B}$$

instead of 1.

Thus, we obtain the new value of coefficient c_1' which is similar to (53):

$$c_1' = \left(\prod_{1\leq i<j\leq k}\frac{1}{r_j-r_i}\right)\left(\sum_{i=1}^{k}\frac{2^{q^{i-k}\lg B}-2}{B}(-1)^{i+1}S_{k-i,1}\prod_{2\leq s<t\leq k}(r_s-r_t)\right).$$

Multiplying through and separating the numerator, we have

$$\begin{aligned} c_1' &= \prod_{2\leq j\leq k}\frac{1}{r_j-r_1}\sum_{i=1}^{k}\frac{2^{q^{i-k}\lg B}}{B}(-1)^{i+1}S_{k-i,1} \\ &\quad -\frac{2}{B}\prod_{2\leq j\leq k}\frac{1}{r_j-r_1}\sum_{i=1}^{k}(-1)^{i+1}S_{k-i,1}\,. \end{aligned} \tag{60}$$

Because the second term in (60) is $2c_1/B = O(1/B)$ by (53), we obtain

$$c_1' = \left(\prod_{2\leq j\leq k}\frac{1}{r_j-r_1}\right)\left(\sum_{i=1}^{k}\frac{2^{q^{i-k}\lg B}}{B}(-1)^{i+1}S_{k-i,1}\right) - O\left(\frac{1}{B}\right). \tag{61}$$

In order to bound c_1', we count the number of terms in the summation in (61), i.e., the number of values of i, such that

$$\frac{2^{q^{i-k}\lg B}}{B} > \frac{1}{\lg B}\,.$$

That is, we determine the smallest value of i, such that

$$q^{i-k} > \frac{\lg(B/\lg B)}{\lg B} = 1 - \frac{\lg\lg B}{\lg B}.$$

Thus, we solve that

$$i-k > \ln\left(1 - \frac{\lg\lg B}{\lg B}\right)\frac{\lg e}{\lg q}. \tag{62}$$

We now estimate the previous expression. Recall that $\ln(1-x) > -x$ for $0 < x < 1$. Thus, from (62), we have

$$i-k > -\frac{\lg e \lg\lg B}{\lg q \lg B}.$$

If we denote

$$\nu = k - \frac{\lg e \lg\lg B}{\lg q \lg B}, \tag{63}$$

then we have

$$\frac{2^{q^{i-k}\lg B}}{B} \begin{cases} \leq \dfrac{1}{\lg B}, & \text{when } 1 \leq i \leq \nu; \\ > \dfrac{1}{\lg B}, & \text{when } \nu < i \leq k. \end{cases}$$

Separating the summation in (61) at ν, we obtain

$$\begin{aligned} c_1' \leq & \prod_{2\leq j\leq k} \frac{1}{r_j - r_1} \sum_{i=1}^{\nu} \frac{1}{\lg B}(-1)^{i+1} S_{k-i,1} \\ & + \prod_{2\leq j\leq k} \frac{1}{r_j - r_1} \sum_{i=\nu}^{k} \frac{2^{q^{i-k}\lg B}}{B}(-1)^{i+1} S_{k-i,1} - O\left(\frac{1}{B}\right). \end{aligned} \tag{64}$$

Again, from (53), the first term in (64) is less than $c_1/\lg B = O(1/\lg B)$. Thus, we have

$$c_1' \leq O\left(\frac{1}{\lg B}\right) + \prod_{2\leq j\leq k} \frac{1}{r_j - r_1} \sum_{i=\nu}^{k} \frac{2^{q^{i-k}\lg B}}{B}(-1)^{i+1} S_{k-i,1}. \tag{65}$$

Observe that $2^{q^{i-k}\lg B}/B \leq 1$ for $1 \leq i \leq k$. We separate into the two cases of $S_{i,1}$ in (65) as we do earlier in (54), to obtain

$$c_1' \leq O\left(\frac{1}{\lg B}\right) + (-1)^{k+1} \prod_{2\leq j\leq k} \frac{1}{r_j - r_1} \left(\sum_{\nu<i\leq n} q^{-i} + \sum_{i\geq n+1,\, i>\nu} q^{k-i}\right).$$

Because both q^{-i} and q^{k-i} are less than q^k, we obtain

$$c_1' \le O\left(\frac{1}{\lg B}\right) + (-1)^{k+1} \prod_{2\le j\le k} \frac{1}{r_j - r_1} \sum_{\nu<i\le k} q^k.$$

Plugging (46) and (63) into the above inequality, we have

$$c_1' \le O\left(\frac{1}{\lg B}\right) + \frac{q^k}{kq^{k-1} - nq^{n-1}} \frac{\lg e \lg\lg B}{\lg q \lg B}. \tag{66}$$

We now prove the second term in (66) is $O(\lg\lg B/\lg B)$. Recalling that $q^k = 1/a$ from (15), we have

$$\lg q = -\frac{\lg a}{k} \tag{67}$$

and

$$q^n = q^k - 1 = \frac{1}{a} - 1\,. \tag{68}$$

Taking logs in (68), we obtain

$$n = \frac{\lg(1/a - 1)}{\lg q}\,. \tag{69}$$

Plugging (67) into (69), we obtain

$$n = k\frac{\lg(1/a-1)}{\lg(1/a)}. \tag{70}$$

Notice that the function $f(x) = \lg(x-1)/\lg x$ is increasing for $x > 1$ because $f'(x) > 0$ for $x > 1$. Therefore, by the assumption $a \ge 1/4$ and (70), we have

$$n \le k\frac{\lg 3}{\lg 4} < \frac{4k}{5}. \tag{71}$$

Thus, observing that $q^{k-1} > q^{n-1} > 1$ and the above (71), we obtain

$$kq^{k-1} - nq^{n-1} > kq^{k-1} - \frac{4k}{5}q^{k-1} > k/5. \tag{72}$$

Combining (15), (67) and (72), we have

$$\frac{q^k}{kq^{k-1} - nq^{n-1}} \frac{\lg e}{\lg q} \le -\frac{1}{a}\frac{5}{k}\frac{k\lg e}{\lg a} = \frac{-5\lg e}{a\lg a} \le 10\lg e.$$

Finally, from (66) we obtain

$$c_1' \le O\left(\frac{1}{\lg B}\right) + O\left(\frac{\lg\lg B}{\lg B}\right) = O\left(\frac{\lg\lg B}{\lg B}\right),$$

as claimed. □

Now we present the main Theorem, which we obtain by combining Theorem 16 and 22.

Theorem 23 (Generalized vEB Layout) *The expected cost of a search in the generalized vEB layout is at most* $[\lg e + o(1)] \log_B N + O(\lg\lg B / \lg B) \log_B N + O(1)$.

Applicability of Numerical Results. We have found that numerical simulations provide empirically valuable information (See Lemma 14) on the behavior of a specific choice of parameters for the generalized vEB layout. However an ever present concern is the validity of a —necessarily finite— plot for studying asymptotic behavior. We now give a theorem showing that by using numerical methods in a limited x range we get valid bounds on the functions $\mathcal{B}(x)$, $\mathcal{V}(x)$, $\mathcal{C}(x)$ for *all* possible values of x.

Theorem 24 *Let a and b be constants such that* $0 < a \le b < 1$, $a+b=1$. *Let* $H(x)$ *be a monotonically increasing function whose domain is all the natural numbers, and let* $H(x)$ *satisfy the recursive condition* $H(x) = H(\lceil ax \rceil) + H(\lfloor bx \rfloor)$, *for* $x > \alpha$. *If there exist constants* $c > 0$, $t > 2$ *and* $t > \alpha$, *such that* $H(x) \le cx$ *in the range* $x \in [t-1, t/a]$, *then* $H(x) \le cx$ *for all* $x \ge t-1$.

Proof. First we prove for the purpose of induction that if

$$H(x) \le cx \quad \text{for } t' \ge \frac{t}{a} \text{ and } x \in [t-1, t'],$$

then

$$H(x) \le cx \quad \text{for } x \in \left[t-1, \min\left\{\frac{t'}{b}, \frac{t'-1}{a}\right\}\right].$$

For any

$$y \in \left[t', \min\left\{\frac{t'}{b}, \frac{t'-1}{a}\right\}\right],$$

we know that

$$t-1 \le t \le at' \le ay \le t'-1 \text{ and } t \le bt' \le by \le t',$$

which means

$$t-1 \le \lceil ay \rceil \le t' \text{ and } t-1 \le \lfloor by \rfloor \le t' .$$

Therefore

$$\begin{aligned} H(y) &= H(\lceil ay \rceil) + H(\lfloor by \rfloor) \\ &\le c\lceil ay \rceil + c\lfloor by \rfloor \\ &\le c\lceil (ay+by) \rceil = cy . \end{aligned}$$

We know that

$$t' < \frac{t'-1}{b} \le \min\left\{\frac{t'}{b}, \frac{t'-1}{a}\right\}$$

for all $t' > t/a > 1/a$, and we proved that if $H(x) \le cx$ in range $x \in [t-1, t']$, then $H(x) \le cx$ in range $x \in [t-1, (t'-1)/b]$. A simple induction shows that for all

$$t-1 \le x \le \frac{t'-1-b-\cdots-b^{n-1}}{b^n},$$

we have $H(x) \le cx$. Because

$$t' > \frac{1}{a} = \frac{1}{1-b} = \sum_{i=0}^{\infty} b^i,$$

it is true that $H(x) \le cx$ for all $x \ge t-1$. □

2.4 Conclusion

This chapter gives upper and lower bounds on the cost of cache-oblivious searching; our bounds are tight to within low-order terms. Specifically, we show a lower bound of $\lg e \log_B N$ memory transfers and an upper bound of $[\lg e + \varepsilon + O(\lg\lg B/\lg B)] \log_B N + O(1)$ expected memory transfers in the cache-oblivious model. In contrast, searching uses only $\log_B N + 1$ memory transfers in the DAM model. Interestingly, this $\lg e$ multiplicative slowdown in the cache-oblivious model

compared to the DAM model comes about because the DAM model has only two levels of memory rather then because the memory parameters are unknown in the cache-oblivious model.

We find it intriguing that this constant $\lg e$ plays such a fundamental role in cache-oblivious searching. It would be appealing to find a simpler derivation of $\lg e$ that provides more insight. The current derivation is, we hope, more technical than necessary, but it remains an open question how to simplify.

Chapter 3

Adaptive Packed-Memory Array[1]

In this chapter we study a classic problem in databases and structures, which is how to maintains a dynamic set of N elements in sorted order in a $\Theta(N)$-sized array.

Previous Work. The cache-oblivious version of the above problem is called packed memory array [16, 17]. The PMA maintains N elements in sorted order in a $\Theta(N)$-sized array. It supports operations insert, delete, and scan. Specifically, to insert an element y after a given element x or to delete x costs $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ amortized memory transfers, where B is the number of elements that fit within a memory block. Because the elements are stored physically in sorted order in memory or on disk, the PMA can be used to support extremely efficient range queries. To scan L elements after a given element x costs $\Theta(1+L/B)$ memory transfers. One of the PMA's strengths is its performance on common insertion patterns such as random inserts. For random inserts, the PMA performs extremely well with only $O(\log N)$ element moves per insert and only $O(1+(\log N)/B)$ memory transfers. This performance surpasses the guarantees for arbitrary inserts.

However, the PMA performs relatively poorly on some common insertion patterns such as sequential inserts. For sequential inserts, the PMA performs near its worst in terms of the number of elements moved per insert. The PMA's difficulty with sequential inserts is that the insertions "hammer" on one part of the array,

[1] An earlier version appears in [25].

causing many elements to be shifted around. Although $O(\log^2 N)$ amortized elements moves and $O(1 + (\log^2 N)/B)$ amortized memory transfers is surprisingly good considering the stringent requirements on the data order, it is relatively slow compared with traditional B-tree inserts.

In the rest of chapter we propose an adaptive packed-memory array in our PODS paper [25], which overcomes these deficiencies of the traditional PMA. The adaptive PMA adapts to insertion patterns and is optimized for common insertion patterns such as sequential inserts, random inserts, and bulk inserts. It gives the largest decrease in the cost of sparse arrays/sequential-file maintenance in almost two decades.

3.1 Structures and Algorithms for Adaptive PMA

In this section we introduce the adaptive PMA. We first explain how the adaptive PMA differs from the traditional PMA. We then show that both PMAs have the same amortized bounds, $O(\log^2 N)$ element moves and $O(1 + (\log^2 N)/B)$ memory transfers per insert/delete. Thus, adaptivity comes at no extra asymptotic cost.

Description of Traditional and Adaptive PMAs. We first describe how to insert into both the adaptive and traditional PMAs. Henceforth, ***PMA*** with no preceding adjective refers to either structure. When we insert an element y after an existing element x in the PMA, we look for a neighborhood around element x that has sufficiently low ***density***, that is, we look for a subarray that is not storing too many or too few elements. Once we find a neighborhood of the appropriate density, we ***rebalance*** the neighborhood by spacing out the elements, including y. In the traditional PMA, we rebalance by spacing out the elements evenly. In the adaptive PMA, we may rebalance the elements *unevenly*, based on previous insertions, that is, we leave extra gaps near elements that have recently had inserts after them.

We deal with a PMA that is too full or empty, as with a traditional hash table. Namely, we recopy the elements into a new PMA that is a constant factor larger or smaller. In this chapter, this constant is stated as 2. However, the constant could be larger or smaller (say 1.2) with almost no change in running time. This is because most of the cost from element moves come from rebalances rather than

from recopies.

We now give some terminology. We divide the PMA into $\Theta(N/\log N)$ ***segments***, each of size $\Theta(\log N)$, and we let the number of segments be a power of 2. We call a contiguous group of segments a ***window***. We view the PMA in terms of a tree structure, where the nodes of the tree are windows. The root node is the window containing all segments, and a leaf node is a window containing a single segment. A node in the tree that is a window of 2^i segments has two children, a left child that is the window of the first 2^{i-1} segments and a right child that is the window of the last 2^{i-1} segments.

We let the height of the tree be h, so that $2^h = \Theta(N/\log N)$ and $h = \lg N - \lg\lg N + O(1)$. The nodes at each height ℓ have an ***upper density threshold*** τ_ℓ and a ***lower density threshold*** ρ_ℓ, which together determine the acceptable density of keys within a window of 2^ℓ segments. As the node height *increases*, the upper density thresholds *decrease* and the lower density thresholds *increase*. Thus, for constant minimum and maximum densities $D_{\min}$ and $D_{\max}$, we have

$$D_{\min} = \rho_0 < \cdots < \rho_h < \tau_h < \cdots < \tau_0 = D_{\max}. \tag{73}$$

The density thresholds on windows of intermediate powers of 2 are arithmetically distributed. For example, the maximum density threshold of a segment can be set to 1.0, the maximum density threshold of the entire array to 0.5, the minimum density threshold of the entire array to 0.2, and the minimum density of a segment to 0.1. If the PMA has 32 segments, then the maximum density threshold of a single segment is 1.0, of two segments is 0.9, of four segments is 0.8, of eight segments is 0.7, of 16 segments is 0.6, and of all 32 segments is 0.5.

More formally, upper and lower density thresholds for nodes at height ℓ are defined as follows:

$$\tau_\ell = \tau_h + (\tau_0 - \tau_h)(h-\ell)/h \tag{74}$$

$$\rho_\ell = \rho_h - (\rho_h - \rho_0)(h-\ell)/h. \tag{75}$$

Moreover,

$$2\rho_h < \tau_h, \tag{76}$$

because when we double the size of an array that becomes too dense, the new array must be within the density threshold.[2] Observe that any values of τ_0, τ_h, ρ_0, and ρ_h that satisfy (73)-(76) and enable the array to have size $\Theta(N)$ will work. The important requirement is that

$$\tau_{\ell-1} - \tau_\ell = O(\rho_\ell - \rho_{\ell-1}) = O(1/\log N)\,.$$

We now give more details about how to insert element y after an existing element x. If there is enough space in the leaf (segment) containing x, then we rearrange the elements within the leaf to make room for y. If the leaf is full, then we find the closest ancestor of the leaf whose density is within the permitted thresholds and rebalance. To delete an element x, we remove x from its segment. If the segment falls below its density threshold, then, as before, we find the smallest enclosing window whose density is within threshold and rebalance. As described above, if the *entire* array is above the maximum density threshold (resp., below the minimum density threshold), then we recopy the keys into a PMA of twice (resp., half) the size.

We introduce further notation. Let ***Cap***(u_ℓ) denote the number of array positions in node u_ℓ of height ℓ. Since there are 2^ℓ segments in the node, the capacity is $\Theta(2^\ell \log N)$. Let $\text{Gaps}(u_\ell)$ denote the number of gaps, i.e., unfilled array positions in node u_ℓ. Let $\text{Density}(u_\ell)$ denote the fraction of elements actually stored in node u_ℓ, i.e., $\text{Density}(u_\ell) = 1 - \text{Gaps}(u_\ell)/\text{Cap}(u_\ell)$.

Rebalance. We ***rebalance*** a node u_ℓ of height ℓ if u_ℓ is within threshold, but we detect that a child node $u_{\ell-1}$ is outside of threshold. Any node whose elements are rearranged in the process of a rebalance is ***swept.*** Thus, we ***sweep*** a node u_ℓ of height ℓ when we detect that a child node $u_{\ell-1}$ is outside of threshold, but now u_ℓ need not be within threshold. Note that with this rebalance scheme, this tree can be implicitly rather than explicitly maintained. In this case, a rebalance consists of two scans, one to the left and one to the right of the insertion point until we find a region of the appropriate density.

---

[2]There are straightforward ways to generalize (76) to further reduce space usage. Introducing this generalization here leads to unnecessary complication in presentation.

In a traditional PMA we rebalance evenly, whereas in the adaptive PMA we rebalance unevenly. The idea of the APMA is to store a smaller number of elements in the leaves in which there have been many recent inserts. However, since we must maintain the bound of $O(\log^2 N)$ amortized element moves, we cannot let the density of any child node be too high or too low.

Property 25 *(rebalance property) After a rebalance, if each node u_ℓ (except the root of the rebalancing subtree) has density within u_ℓ's parent's thresholds, then we say that the rebalance satisfies the **rebalance property**. We say that a node u_ℓ is **within balance** or **well balanced** if u_ℓ is within its parent's thresholds.*

The following theorem shows if each rebalance satisfies the rebalance property, then we achieve good update bounds. The proof is essentially that in [16, 17], but the rebalance property applies to a wide set of rebalancing schemes.

Theorem 26 *If the rebalance in a PMA satisfies the rebalance property, then inserts and deletes take $O(\log^2 N)$ amortized element moves and $O(1 + (\log^2 N)/B)$ amortized memory transfers.*

Proof. Let u_ℓ be a node at level ℓ. A rebalance of u_ℓ is triggered by an insert or delete that pushes one descendant node u_i at each height $i = 0, \ldots, \ell - 1$ above its upper threshold τ_i or below its lower threshold ρ_i. (If this were not the case, then we would rebalance a node of a lower height than ℓ.)

Consider one particular such node u_i. Before the sweep of u_i's parent u_{i+1},

$$\text{Density}(u_i) > \tau_i \quad \text{or} \quad \text{Density}(u_i) < \rho_i \,.$$

After the sweep of u_{i+1}, by the rebalance property,

$$\rho_{i+1} \le \text{Density}(u_i) \le \tau_{i+1} \,.$$

Therefore we need at least

$$(\tau_i - \tau_{i+1})\text{Cap}(u_i)$$

inserts or at least

$$(\rho_{i+1} - \rho_i)\text{Cap}(u_i)$$

deletes before the next sweep of node u_{i+1}. Therefore the amortized size of a sweep of node u_{i+1} per insert into child node u_i is at most

$$\max\left\{\frac{\mathrm{Cap}(u_{i+1})}{(\tau_i-\tau_{i+1})\mathrm{Cap}(u_i)},\frac{\mathrm{Cap}(u_{i+1})}{(\rho_{i+1}-\rho_i)\mathrm{Cap}(u_i)}\right\} = \max\left\{\frac{2}{\tau_i-\tau_{i+1}},\frac{2}{\rho_{i+1}-\rho_i}\right\} = O(\log N).$$

When we insert an element into the PMA, we actually insert into $h=\Theta(\log N)$ such nodes u_i, one at each level in the tree. Therefore the total amortized size of a rebalance per insertion into the PMA is $O(\log^2 N)$. Thus, the amortized number of element moves per insert is $O(\log^2 N)$. Because a rebalance is composed of a constant number of sequential scans, the amortized number of memory transfers per insert is $O(1+(\log^2 N)/B)$, as promised. □

Observe that Theorem 26 applies to both insertions and deletions; in contrast, we focus only on insertions in the rest of the chapter, for the sake of simplicity. However, it is likely that, with only minor modifications to the predictor, the same bounds for common insertion distributions can be made to apply to deletion distributions and to distributions combining both operations. There does not seem to be any significant additional difficulties in dealing with deletions.

Prediction. In a ***predictor*** data structure we keep track of a small collection of elements, called ***marker*** elements, that directly precede elements recently inserted into the APMA. The predictor stores a pointer to those leaf nodes of the APMA (i.e., $\Theta(\log N)$-sized segments of the array) that contain marker elements. For each marker element, we count the number of recently inserted elements that directly follow the marker.

We give terminology for prediction. For an element x, let ***insert number*** $I(x)$ denote a count from 0 to $\log N$ estimating the number of inserts after x in the last $O(\log^2 N)$ inserts. The predictor is designed so that

- $I(x)$ is always an underestimate of the number of inserts, and
- $I(x)$ never grows above $\log N$.

Below, we explain why and how these properties are enforced. Furthermore, if element x is not in the predictor, then we define $I(x)=0$.

We now define the insert number $I(u_\ell)$ of a node u_ℓ at level ℓ in the APMA. Specifically, let insert number $I(u_\ell)$ be the sum of the insert numbers of elements in

Algorithm 1 Predictor.insert(x)

```
1:  if ∃ a cell c such that c.element = x then
2:     SWAP(c, c.nextcell)                                  { If c is not the head pointer. }
3:     c.count ← c.count + 1
4:     if c.count > log N then
5:        tailpointer.count ← tailpointer.count − 1
                                                            {When c.count is at the maximum logN...}
6:        c.count ← c.count − 1      { We decrease the tail's count instead of increasing c.count. }
7:     end if
8:  else
9:     if headpointer.nextcell = tailpointer then
10:       tailpointer.count ← tailpointer.count − 1
                                                            { Decrease tail's count when no free space. }
11:    else
12:       headpointer ← headpointer.nextcell
13:       headpointer.element ← x
14:       headpointer.count ← 1
15:       headpointer.leaf ← x.leaf          { In other cases, create a new cell for new element. }
16:    end if
17: end if
18: if tailpointer.count = 0 then
19:    tailpointer ← tailpointer.nextcell    { The tail cell is removed when its count drops to zero.}
20: end if
```

u_ℓ. When rebalancing a node, we reallocate elements unevenly among its descendant leafs according to their insert numbers. The larger the insert number, the fewer elements are allocated.

We now explain how the predictor determines (1) which elements to store as marker elements and (2) what the count numbers are for each element. To do so, we explain how to implement the predictor.

The predictor is a circular linked list, stored in an array. The predictor contains $\beta \log N$ cells, for constant β. Two pointers, a head pointer and a tail pointer, indicate the front and the back of the linked list. Each cell in the circular linked list stores a marker element x. Associated with x are two pieces of data, (1) a pointer to the leaf node in the APMA where x currently resides and (2) the count number $I(x)$ (see Figure 1).

When a new element x is determined to be a marker element, it is inserted into the predictor; x is inserted at the head of the linked list (where the head-pointer points). When an element x is no longer needed as a marker element, it is deleted from the predictor; before x is deleted, x will always be stored at the tail of the linked list (where the tail-pointer points).

When a new element y is inserted into the APMA after an element x, we first check whether x is a marker element (i.e., stored in the predictor). If x is a marker element, we store x and its auxiliary information one cell forward in the APMA (unless x is already at the head of the predictor). Let w be the element displaced by x. We store w (and auxiliary information) in the cell vacated by x. We also increase the element x's count number by 1 unless it is already at the maximum $O(\log N)$. Let z be the element stored in the tail of the predictor. If x is already at the maximum $O(\log N)$, then we decrement z's count number instead of incrementing x's count number. (This decrement is one reason why the count number of x is an underestimate.)

If x is not a marker element, then there are two cases. If there are empty cells in the predictor, then we store x at the head of the predictor. If there are no empty cells in the predictor, then we decrease the count number of z (the element stored in

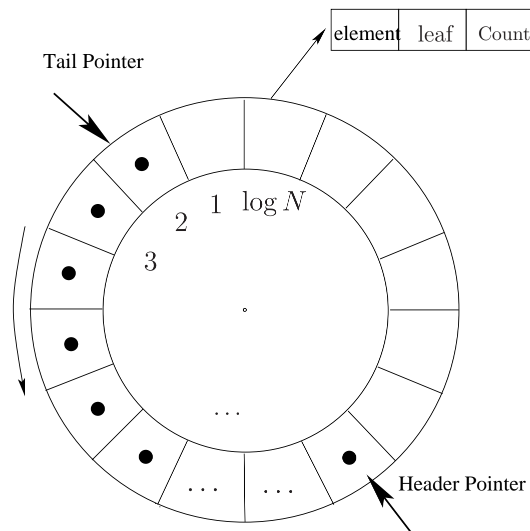

Figure 1: The predictor. Each cell contains a marker element x, the leaf node in the APMA where x resides, and the count number $I(x)$.

the tail) instead of storing x in the predictor. (This lack of space is another reason why the count number of x is an underestimate.)

This decrement may reduce the count number of z to 0. If so, we delete z from the predictor. A new free cell space is now available for future inserts.

The predictor algorithm is engineered to tolerate "random noise" in the insertions. By random noise, we mean that some of the insertions may not follow an insertion distribution (such as head insert, hammer insert, bulk insert, etc). Our guarantees still apply even if as much as a constant fraction of insertions are after random elements in the APMA. To understand why our predictor tolerates random noise, observe that a few arbitrary inserts will not be stored in the predictor unless the tail count drops below zero. If a poor choice of element is, in fact, stored in the predictor, it will soon be swapped to the tail if no new inserts follow.

Uneven Rebalance. Now we present the algorithm for uneven rebalance (See Algorithm 2). Assume that nodes $u_{\ell-1}$ and $v_{\ell-1}$ are left and right children of u_ℓ at level ℓ and that there are m ordered elements $\{x_1, x_2, \ldots, x_m\}$ stored in u_ℓ. The uneven rebalance performs as follows:

- If $I(x_i) = 0$ for all $i \in [1,m]$, then we perform an even rebalance for this node u_ℓ.
- Otherwise, we perform an uneven rebalance. Our uneven rebalance is designed so that, the bigger the insert numbers, the more gaps we leave. Specifically, we minimize the quantity

$$\left| \frac{I(u_{\ell-1})}{\text{Gaps}(u_{\ell-1})} - \frac{I(v_{\ell-1})}{\text{Gaps}(v_{\ell-1})} \right|, \tag{77}$$

subject to the constraint that the rebalance property must be satisfied. When we rebalance, we ***split at an element*** x_i, meaning that we put elements $\{x_1, \ldots, x_i\}$ in $u_{\ell-1}$ and $\{x_{i+1}, \ldots, x_m\}$ in $v_{\ell-1}$. The objective is to find the index i to minimize

$$\left| \frac{\sum_{j=1}^{i} I(x_j)}{\text{Cap}(u_{\ell-1}) - i} - \frac{\sum_{j=i+1}^{m} I(x_j)}{\text{Cap}(v_{\ell-1}) - (m-i)} \right|, \tag{78}$$

subject to the constraints that

$$i \in \Big[\mathrm{Cap}(u_{\ell-1})\rho_\ell, \mathrm{Cap}(u_{\ell-1})\tau_\ell\Big], \tag{79}$$

// density of left child is within parents' threshold

$$i \in \Big[m - \mathrm{Cap}(v_{\ell-1})\tau_\ell, m - \mathrm{Cap}(v_{\ell-1})\rho_\ell\Big]. \tag{80}$$

// density of right child is within parents' threshold

- We recursively allocate elements in $u_{\ell-1}$ and $v_{\ell-1}$'s child nodes and proceed down the tree until we reach the leaves. Once we know the number of elements in each leaf, we rebalance u_ℓ in one scan.

For example, in the insert-at-head case, the insert numbers of right descendants are always 0. Thus, minimizing the simplified objective quantity $|I(u_{\ell-1})/\mathrm{Gaps}(u_{\ell-1})|$ means maximizing $\mathrm{Gaps}(u_{\ell-1})$.

Algorithm 2 Rebalance.uneven(u_ℓ)

1: $u_{\ell-1} \leftarrow u_\ell$'s left child;
2: $v_{\ell-1} \leftarrow u_\ell$'s right child;
3: **if** ($u_{\ell-1}$ is empty) or ($v_{\ell-1}$ is empty) **then**
4: return;
5: **end if**
6: splitnum $\leftarrow \max\{\mathrm{Cap}(u_{\ell-1})\rho_\ell, m - \mathrm{Cap}(v_{\ell-1})\tau_\ell\}$;
7: optvalue $\leftarrow \left| \frac{\sum_{j=1}^{\mathrm{splitnum}} I(x_j)}{\mathrm{Cap}(u_{\ell-1}) - \mathrm{splitnum}} - \frac{\sum_{j=\mathrm{splitnum}+1}^{m} I(x_j)}{\mathrm{Cap}(v_{\ell-1}) - (m - \mathrm{splitnum})} \right|$;
8: **for** $i =$ splitnum to $\min\{\mathrm{Cap}(u_{\ell-1})\tau_\ell, m - \mathrm{Cap}(v_{\ell-1})\rho_\ell\}$ **do**
9: curvalue $\leftarrow \left| \frac{\sum_{j=1}^{i} I(x_j)}{\mathrm{Cap}(u_{\ell-1}) - i} - \frac{\sum_{j=i+1}^{m} I(x_j)}{\mathrm{Cap}(v_{\ell-1}) - (m-i)} \right|$;
10: **if** optvalue $>$ curvalue **then**
11: optvalue $\leftarrow$ curvalue;
12: splitnum $\leftarrow i$;
13: **end if**
14: **end for**
15: $u_{\ell-1} \leftarrow \{x_1, \ldots, x_{\mathrm{splitnum}}\}$;
16: $v_{\ell-1} \leftarrow \{x_{\mathrm{splitnum}+1}, \ldots, x_m\}$;
17: Rebalance.uneven($u_{\ell-1}$);
18: Rebalance.uneven($v_{\ell-1}$);

Now we show how to implement the rebalance so that there is no asymptotic overhead in the bookkeeping for the rebalance. Specifically, the number of element moves in the uneven rebalance is dominated by the size of the rebalancing node, as described in the following theorem:

Theorem 27 *To rebalance a node u_ℓ at level ℓ unevenly requires $O(Cap(u_\ell))$ operations and $O(1+Cap(u_\ell)/B)$ memory transfers.*

Proof. There are three steps to rebalancing a node u_ℓ unevenly. First, we check the predictor to obtain the insert numbers of the elements located in all descendant nodes of u_ℓ. Because the size of the predictor is $O(\log N)$, this step takes $O(\log N)$ operations and $O(1+(\log N)/B)$ memory transfers. Second, we recursively determine the number of elements to be stored in u_ℓ's children, grandchildren, etc., down to descendent leaves. Naively, this procedure uses $O(\ell \text{Cap}(u_\ell))$ operations and $O(1+\ell \text{Cap}(u_\ell)/B)$ memory transfers; below we show how to perform this procedure in $O(\text{Cap}(u_\ell))$ operations and $O(1+\text{Cap}(u_\ell)/B)$ memory transfers. Third, we scan the node u_ℓ putting each element into the correct leaf node. Thus, this last step also takes $O(\text{Cap}(u_\ell))$ operations and $O(1+\text{Cap}(u_\ell)/B)$ memory transfers.

We now show how to implement the second step efficiently. We call the elements in the predictor ***weighted*** elements and the remaining elements ***unweighted***. Recall that only weighted elements have nonzero insert numbers. In the first step, we obtain all information about which elements are weighted. Then, we start the second step, which is recursive. At the first recursive level, we determine which elements are allocated to the left and right children of u_ℓ, i.e., we find the index i minimizing (78). At first glance, it seems necessary to check all indices i in order to get the minimum, which takes $O(\text{Cap}(u_\ell))$ operations, but we can do better. Observe that when the index i is in a sequence of unweighted elements between two weighted elements, the numerator in (78) does not change. Only the denominator changes, and it does so continuously. So in order to minimize (78) at the first recursive level, it is not necessary to check all elements in node u_ℓ. It is enough to check which two contiguous weighted elements the index i is between such that (78) is minimized. Since there are at most $O(\log N)$ weighted elements, the number of operations at each recursive level is at most $O(\log N)$. Furthermore, because there are ℓ recursive levels, the number of operations in the whole recursive step

is at most $O(\ell \log N)$, which is less than $O(\mathrm{Cap}(u_\ell))$. By storing these weighted elements contiguously during the rebalance, we obtain $O(1+\mathrm{Cap}(u_\ell)/B)$ memory transfers. □

3.2 Analysis of Sequential and Hammer Insertions

In this section we first analyze the adaptive PMA for the sequential insert pattern, where inserts occur at the front of the PMA. Then we generalize the result to hammer inserts.

For sequential inserts, we prove the following theorem:

Theorem 28 *For sequential inserts, the APMA has $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ amortized memory transfers.*

We give some notation. In the rest of this section, we assume that u_ℓ is the leftmost node at level ℓ and $v_{\ell-1}$ is the right child of u_ℓ. Recall that leaves have height 0. Suppose that we insert N elements in the front of an array of size cN ($c > 1$). Since we always insert elements at the front, rebalances occur only at the leftmost node u_ℓ ($0 \le \ell \le h$). If we know the number of sweeps of u_ℓ in the process of inserting these N elements, then we also know the total number of moves.

In order to bound the number of sweeps at each level, we need more notation. For $\kappa \le \ell$, let $\mathcal{N}_\kappa(\ell, i)$ be the number of sweeps of the leftmost node u_κ at level κ between the $(i-1)$th sweep and the ith sweep of node u_ℓ. We imagine a virtual parent node u_{h+1} of the root node u_h, where u_{h+1} has size $2cN$. Thus, the time when the root node u_h reaches its upper threshold τ_h, after we insert N elements, is the time when the virtual parent node performs the first rebalance. Thus, $\mathcal{N}_\kappa(h+1,1)$ is the number of sweeps of node u_κ at level κ during the insertion of these N elements ($0 \le \kappa \le h$). Since each sweep of u_κ costs $2^\kappa \log N$ moves, the total number of moves is:

$$\sum_{\kappa=0}^{h} \mathcal{N}_\kappa(h+1,1) 2^\kappa \log N.$$

This quantity is the sum of the sweep costs at each level, until the virtual node needs

its first rebalance. Thus, the amortized number of element moves is

$$\frac{1}{N}\sum_{\kappa=0}^{h}\mathcal{N}_{\kappa}(h+1,1)2^{\kappa}\log N. \tag{81}$$

Sequential Inserts with Only Upper Thresholds. For pedagogical reasons, we now consider the simpler case of a PMA with no lower-bound thresholds and show that Theorem 28 holds in this special case. This lack of lower-bound thresholds makes it significantly easier to achieve the bounds from Theorem 28. By providing this simpler analysis we give insight into the origin of Theorem 28's bounds and why the subsequent analysis is more complicated.

Lemma 29 *For sequential inserts, the APMA with no lower-bound thresholds has $O(\log N)$ amortized element moves and $O(1+\log N/B)$ amortized memory transfers.*

Proof. Recall that $\mathcal{N}_{\kappa}(\ell,1)$ is the number of sweeps of the leftmost child u_κ at level κ until ancestor node u_ℓ performs its first rebalance. Observe that just before u_ℓ performs the first rebalance, $u_{\ell-1}$ reaches its threshold $\tau_{\ell-1}$. We want to find the number of sweeps of u_κ before $u_{\ell-1}$ reaches its upper threshold $\tau_{\ell-1}$.

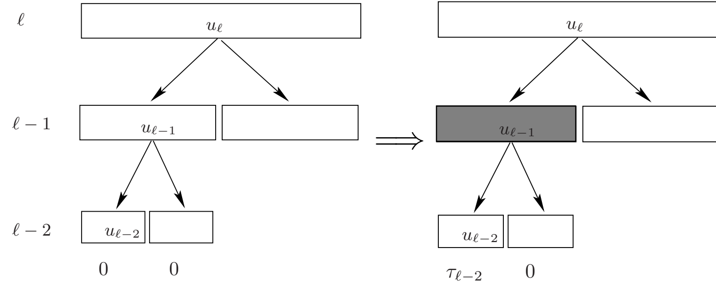

Figure 2: In the simple case, the shaded region is rebalanced just after Phase 1 of node u_ℓ, which starts from Density$(u_{\ell-2})=0$ (left) and ends at Density$(u_{\ell-2})=\tau_{\ell-2}$ (right).

We decompose this process into two phases. Phase 1 ends before the first rebalance of node $u_{\ell-1}$ when we have $\tau_{\ell-2}2^{\ell-2}$ elements in the left child $u_{\ell-2}$ of $u_{\ell-1}$ and 0 elements in the right child $v_{\ell-2}$ of $u_{\ell-1}$ (see Figure 2). According to our uneven-rebalance strategy, since all inserts are to the left child, we allocate

$\tau_{\ell-1}2^{\ell-2}$ elements to $v_{\ell-2}$ and $(\tau_{\ell-2}-\tau_{\ell-1})2^{\ell-2}$ elements to $u_{\ell-2}$ at the end of Phase 1, i.e., we give the maximum allowed number of elements to the right child. Now we consider Phase 2, which takes place between the first rebalance and the second rebalance of $u_{\ell-1}$ (see Figure 3). Since the right child $v_{\ell-2}$ of $u_{\ell-1}$ already has density $\tau_{\ell-1}$, when $u_{\ell-2}$ reaches its threshold $\tau_{\ell-2}$ again, the density of $u_{\ell-1}$ is $(\tau_{\ell-2}+\tau_{\ell-1})/2 > \tau_{\ell-1}$ at the end of Phase 2, which is above its upper threshold.

To summarize, the first time that we rebalance $u_{\ell-1}$ is when we move elements from $u_{\ell-2}$ into $v_{\ell-2}$. This rebalance is triggered because $u_{\ell-2}$ is above its threshold. The next time $u_{\ell-2}$ goes above its threshold $\tau_{\ell-2}$, $u_{\ell-1}$ is also above its threshold $\tau_{\ell-1}$, and we trigger the first rebalance of u_{ℓ}. Thus, there are at most two sweeps of node $u_{\ell-1}$ before it reaches its threshold $\tau_{\ell-1}$. That is

$$\mathcal{N}_{\kappa}(\ell,1) \leq \mathcal{N}_{\kappa}(\ell-1,1)+\mathcal{N}_{\kappa}(\ell-1,2). \tag{82}$$

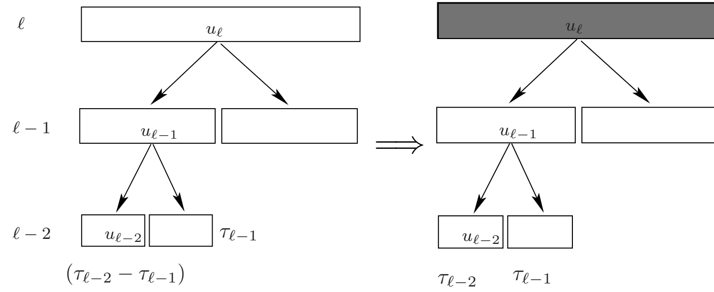

Figure 3: In the simple case, the shaded region is rebalanced just after Phase 2 of node u_ℓ, which starts from Density$(u_{\ell-2}) = \tau_{\ell-2}-\tau_{\ell-1}$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right).

To calculate (81), we first show that $\mathcal{N}_{\kappa}(\ell-1,2) < \mathcal{N}_{\kappa}(\ell-1,1)$. Recall that $\mathcal{N}_{\kappa}(\ell-1,2)$ is the number of sweeps of the leftmost child u_κ at level κ between ancestor node $u_{\ell-1}$'s first sweep (rebalance) and second sweep. The above inequality is true because at the end of both phases $u_{\ell-2}$ reaches its threshold, but the first phase starts with $u_{\ell-2}$ having density 0 (an empty data structure), and the second phase starts with $u_{\ell-2}$ having density $\tau_{\ell-2}-\tau_{\ell-1}$. Thus, by plugging $\mathcal{N}_{\kappa}(\ell-1,2) < \mathcal{N}_{\kappa}(\ell-1,1)$ in (82), we have the recurrence

$$\mathcal{N}_{\kappa}(\ell,1) \leq 2\mathcal{N}_{\kappa}(\ell-1,1).$$

The amortized number of element moves is

$$
\begin{aligned}
\frac{1}{N}\sum_{\kappa=0}^{h}\mathcal{N}_\kappa(h+1,1)2^\kappa\log N &= \sum_{\kappa=0}^{h}\mathcal{N}_\kappa(h+1,1)2^{\kappa-h}\\
&\leq \sum_{\kappa=0}^{h}\left[2\mathcal{N}_\kappa(h,1)\right]2^{\kappa-h}\\
&\leq \sum_{\kappa=0}^{h}\left[2^{h-\kappa+1}\mathcal{N}_\kappa(\kappa,1)\right]2^{\kappa-h}\\
&= \sum_{\kappa=0}^{h}2 = O(\log N).
\end{aligned}
$$

□

Sequential Inserts in APMA with Lower and Upper Thresholds. We now consider the general case of a PMA with both the lower- and upper-bound thresholds and are ready to prove Theorem 28.

PROOF OF THEOREM 28: The proof is a generalization of the proof of Lemma 29; we bound $\mathcal{N}_\kappa(\ell,1)$, the number of sweeps of the leftmost child u_κ at level κ until the ancestor node u_ℓ performs the first rebalance. The difficulty with both the lower- and upper-bound thresholds is that we must decompose the time before the first rebalance of u_ℓ into more than 2 phases, and thus we obtain a more complicated recurrence to solve. We decompose this process into three phases. ***Phase*** i ***of node*** u_ℓ $(1 \leq i \leq 3)$, starts after the $(i-1)$th sweep of $u_{\ell-1}$ and ends at the ith sweep of $u_{\ell-1}$. At the end of the last phase, u_ℓ performs its first rebalance, which is the third sweep of $u_{\ell-1}$. Thus, we have at most three sweeps of node $u_{\ell-1}$ before the first rebalance of u_ℓ:

$$\mathcal{N}_\kappa(\ell,1) \leq \mathcal{N}_\kappa(\ell-1,1)+\mathcal{N}_\kappa(\ell-1,2)+\mathcal{N}_\kappa(\ell-1,3).$$

Now we prove the above claim analyzing the densities in each phase.

I) We consider the densities of child nodes $u_{\ell-2}$ and $v_{\ell-2}$ of $u_{\ell-1}$ at the end of Phase 1. The first rebalance of $u_{\ell-1}$ occurs (see Figure 4) when $u_{\ell-2}$ reaches its upper threshold $\tau_{\ell-2}$. For sequential inserts, we allocate as many free

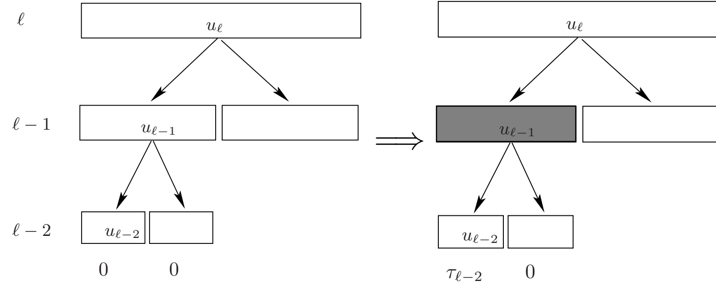

Figure 4: Phase 1 of node u_ℓ starts from Density$(u_{\ell-2}) = 0$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced.

spaces as possible to $u_{\ell-2}$, while ensuring that $u_{\ell-2}$ and $v_{\ell-2}$ have densities between $\rho_{\ell-1}$ and $\tau_{\ell-1}$. Thus, after the first rebalance, which happens after $\tau_{\ell-2}\text{Cap}(u_{\ell-2})$ inserts, we have densities:

$$\begin{aligned} \text{Density}(u_{\ell-2}) &= \rho_{\ell-1}, \\ \text{Density}(v_{\ell-2}) &= \tau_{\ell-2} - \rho_{\ell-1}. \end{aligned}$$

It is immediate that the density setting of $u_{\ell-2}$ is legal; we now explain why the above density setting of $v_{\ell-2}$ is legal, i.e., satisfies the rebalance property. Notice that $\rho_{\ell-1} \leq \tau_{\ell-2} - \rho_{\ell-1} \leq \tau_{\ell-1}$, since $2\rho_{\ell-1} \leq \tau_{\ell-1} < \tau_{\ell-2}$ by (73) and (76) and $\tau_{\ell-2} - \tau_{\ell-1} = O(1/\log N) < \rho_{\ell-1}$ by (73) and (74).

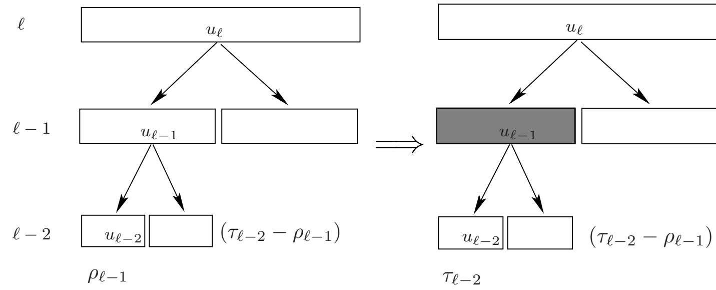

Figure 5: Phase 2 of node u_ℓ starts from Density$(u_{\ell-2}) = \rho_{\ell-1}$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced.

II) We now consider the densities of child nodes $u_{\ell-2}$ and $v_{\ell-2}$ at the end of Phase 2. When $u_{\ell-2}$ reaches its threshold again, Phase 2 of node u_ℓ ends (see Figure 5). After $u_{\ell-1}$ does the second rebalance, which happens after

$(\tau_{\ell-2} - \rho_{\ell-1})\mathrm{Cap}(u_{\ell-2})$ inserts, we have densities:

$$\begin{aligned}\mathrm{Density}(u_{\ell-2}) &= 2\tau_{\ell-2} - \rho_{\ell-1} - \tau_{\ell-1},\\ \mathrm{Density}(v_{\ell-2}) &= \tau_{\ell-1}.\end{aligned}$$

It is immediate that the density setting of $v_{\ell-2}$ is legal; we now show that the density setting of $u_{\ell-2}$ is legal. Notice that $\rho_{\ell-1} < 2\tau_{\ell-2} - \rho_{\ell-1} - \tau_{\ell-1} < \tau_{\ell-1}$, because $2\rho_{\ell-1} < \tau_{\ell-2} < \tau_{\ell-2} + (\tau_{\ell-2} - \tau_{\ell-1})$ by (73) and (76) and $2(\tau_{\ell-2} - \tau_{\ell-1}) = O(1/\log N) < \rho_{\ell-1}$ by (73) and (74).

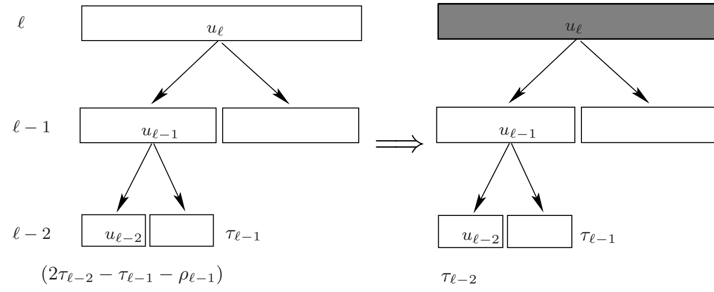

Figure 6: Phase 3 of node u_ℓ starts from $\mathrm{Density}(u_{\ell-2}) = 2\tau_{\ell-2} - \tau_{\ell-1} - \rho_{\ell-1}$ (left) and ends at $\mathrm{Density}(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced.

III) Now we consider the densities of child nodes $u_{\ell-2}$ and $v_{\ell-2}$ at the end of Phase 3. When $u_{\ell-2}$ reaches its threshold a third time, which happens after $(\tau_{\ell-1} - \tau_{\ell-2} + \rho_{\ell-1})\mathrm{Cap}(u_{\ell-2})$ inserts, Phase 3 of node u_ℓ ends (see Figure 6). When $u_{\ell-1}$ does the third sweep, the density of $u_{\ell-1}$ is $(\tau_{\ell-2} + \tau_{\ell-1})/2 > \tau_{\ell-1}$, so $u_{\ell-1}$ is above threshold. Thus, the end of Phase 3 is the first rebalance of u_ℓ.

Thus, there are at most three sweeps of $u_{\ell-1}$ before the first rebalance of u_ℓ, that is,

$$\mathcal{N}_\kappa(\ell,1) \le \mathcal{N}_\kappa(\ell-1,1) + \mathcal{N}_\kappa(\ell-1,2) + \mathcal{N}_\kappa(\ell-1,3). \tag{83}$$

We cannot simply use the bound $\mathcal{N}_\kappa(\ell,1) \le 3\mathcal{N}_\kappa(\ell-1,1)$ for our analysis, since this bound naively leads to $O(N^{\log(3/2)})$ amortized moves, which is far from our goal of $O(\log N)$.

To establish our bound, we prove the following recurrences for Phase 2 and Phase 3:

$$\mathcal{N}_\kappa(\ell-1,2) \leq 2\mathcal{N}_\kappa(\ell-2,2), \tag{84}$$

and

$$\mathcal{N}_\kappa(\ell-1,3) \leq \mathcal{N}_\kappa(\ell-1,2). \tag{85}$$

Solving (83), (84), and (85) will yield the desired bound.

We already showed (83); now we show (84). We proceed by breaking Phase 2 into two subphases. The first subphase begins when Phase 2 begins, i.e., after the first rebalance of $u_{\ell-1}$, and it ends after the next sweep of $u_{\ell-2}$. The second subphase begins when the first subphase ends, and it ends after the next another sweep of $u_{\ell-2}$. We will show that at the end of Subphase 2, $u_{\ell-2}$ is above threshold, meaning that Subphase 2 ends with a sweep of $u_{\ell-1}$, i.e., Phase 2 ends as well.

- At the beginning of Subphase 1, node $u_{\ell-3}$ has density $\rho_{\ell-2}$ by the rebalance property. (Since insertions are at the beginning of the array, we want $u_{\ell-3}$ to be as sparse as possible, and the rebalance property says that after a rebalance $\text{Density}(u_{\ell-3}) \geq \rho_{\ell-2}$.) The sweep of $u_{\ell-2}$ is triggered once the density of $u_{\ell-3}$ reaches $\tau_{\ell-3}$ (see Figure 7). At the end of Subphase 1, after $(\tau_{\ell-3} - \rho_{\ell-2})\text{Cap}(u_{\ell-3})$ inserts, the density of $u_{\ell-3}$ and $v_{\ell-3}$ are:

  $$\begin{aligned} \text{Density}(u_{\ell-3}) &= 2\rho_{\ell-1} - \rho_{\ell-2} + \tau_{\ell-3} - \tau_{\ell-2}, \\ \text{Density}(v_{\ell-3}) &= \tau_{\ell-2}. \end{aligned}$$

  It is immediate that the density of $v_{\ell-3}$ is legal; we show that the density of $u_{\ell-3}$ is legal too. Notice that $\rho_{\ell-2} < 2\rho_{\ell-1} - \rho_{\ell-2} + \tau_{\ell-3} - \tau_{\ell-2} < \tau_{\ell-2}$, because $2\rho_{\ell-2} < 2\rho_{\ell-1}$ and $\tau_{\ell-2} < \tau_{\ell-3}$ by (73) and $2\rho_{\ell-1} < \tau_{\ell-1} < \tau_{\ell-2}$ and $\tau_{\ell-3} - \tau_{\ell-2} = O(1/\log N) < \rho_{\ell-2}$ by (73) and (76).

  We now show that the number of sweeps of u_κ in Subphase 1 is equal to $\mathcal{N}_\kappa(\ell-2,2)$. Observe that Subphase 1 is exactly Phase 2 of the node $u_{\ell-1}$ because they both start with the node $u_{\ell-3}$ having density $\rho_{\ell-2}$ and end with the node $u_{\ell-3}$ having density $\tau_{\ell-3}$. Although in Subphase 1 and Phase 2 of node $u_{\ell-1}$, node $v_{\ell-3}$ has different densities, this difference does not matter because the density of $v_{\ell-3}$ does not affect when Subphase 1 and Phase 2 of node $u_{\ell-1}$ end.

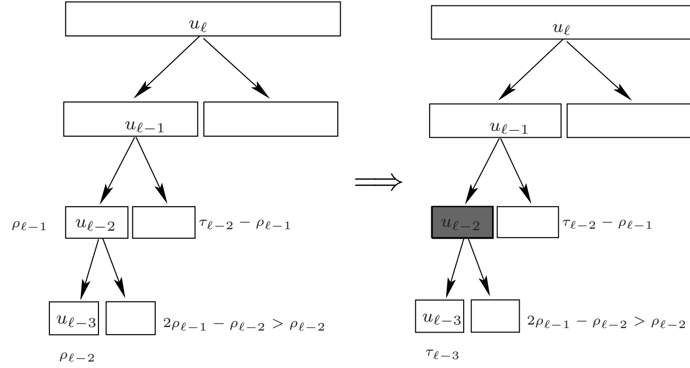

Figure 7: Subphase 1 starts from Density$(u_{\ell-3}) = \rho_{\ell-2}$ (left) and ends at Density$(u_{\ell-3}) = \tau_{\ell-3}$ (right). The shaded region is rebalanced.

- At the beginning of Subphase 2, $u_{\ell-3}$ has density $2\rho_{\ell-1} - \rho_{\ell-2} + \tau_{\ell-3} - \tau_{\ell-2} > \rho_{\ell-2}$, and the subsequent sweep of $u_{\ell-2}$ is triggered once the density of $u_{\ell-3}$ reaches $\tau_{\ell-3}$ again (see Figure 8). Since the density of $v_{\ell-3}$ is $\tau_{\ell-2}$, the density of $u_{\ell-2}$ is $(\tau_{\ell-3} + \tau_{\ell-2})/2 > \tau_{\ell-2}$ at the end of Subphase 2, so $u_{\ell-2}$ is above its upper threshold. Thus, the end of Subphase 2 is the sweep of $u_{\ell-1}$.

  We now prove that the number of sweeps of u_κ in Subphase 2 is less than $\mathcal{N}_\kappa(\ell-2, 2)$, because both Subphase 2 and Phase 2 of node $u_{\ell-1}$ end with node $u_{\ell-3}$ reaching its upper threshold $\tau_{\ell-3}$, but Subphase 2 starts with node $u_{\ell-3}$ having density greater than $\rho_{\ell-2}$ while Phase 2 of node $u_{\ell-1}$ starts with node $u_{\ell-3}$ having density $\rho_{\ell-2}$.

Thus, there are at most two subphases in Phase 2 of node u_ℓ and each subphase has the number of sweeps of node u_κ at most $\mathcal{N}_\kappa(\ell-2, 2)$, which shows (84). Since Recurrence (84) has the base case $\mathcal{N}_\kappa(\kappa, 2) = 1$, we obtain the solution

$$\mathcal{N}_\kappa(\ell-1, 2) \leq 2^{\ell-\kappa-1}. \tag{86}$$

Now we establish the recurrence in (85). Both Phase 2 and Phase 3 end with node $u_{\ell-2}$ reaching its upper threshold $\tau_{\ell-2}$, while Phase 3 starts with the node $u_{\ell-2}$ having density $2\tau_{\ell-2} - \tau_{\ell-1} - \rho_{\ell-1} > \rho_{\ell-1}$. Phase 2 starts with node $u_{\ell-2}$ having density $\rho_{\ell-1}$.

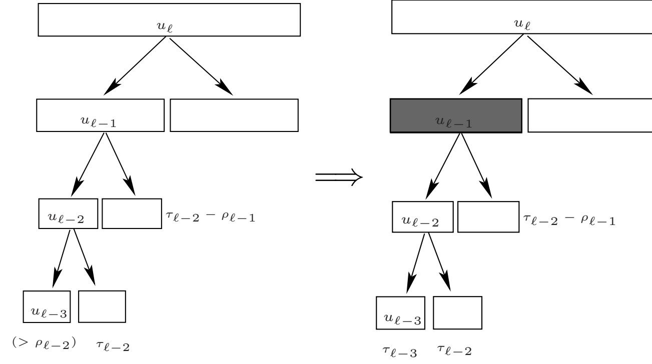

Figure 8: Subphase 2 starts from Density$(u_{\ell-3}) \geq \rho_{\ell-2}$ (left) and ends at Density$(u_{\ell-3}) = \tau_{\ell-3}$ (right). The shaded region is rebalanced.

We now establish the desired bound. Plugging (86) and (85) into (83), we have

$$\begin{aligned} \mathcal{N}_\kappa(\ell,1) &\leq \mathcal{N}_\kappa(\ell-1,1)+\mathcal{N}_\kappa(\ell-1,2)+\mathcal{N}_\kappa(\ell-1,3) \\ &\leq \mathcal{N}_\kappa(\ell-1,1)+2\mathcal{N}_\kappa(\ell-1,2) \\ &\leq \mathcal{N}_\kappa(\ell-1,1)+2\cdot 2^{\ell-\kappa-1} \\ &\leq 2^{\ell-\kappa+1}. \end{aligned} \tag{87}$$

Finally, the amortized number of moves is

$$\frac{1}{N}\sum_{\kappa=0}^{h}\mathcal{N}_\kappa(h+1,1)2^\kappa \log N = \sum_{\kappa=0}^{h}\mathcal{N}_\kappa(h+1,1)2^{\kappa-h}$$
$$\leq \sum_{\kappa=0}^{h}(2^{h-\kappa+2})2^{\kappa-h} = \sum_{\kappa=0}^{h}4 = O(\log N).$$

Observe that after any insert the elements are moved from a contiguous group, and the moves can be performed with a constant number of scans. Therefore the amortized number of memory transfers is $O(1+(\log N)/B)$. □

Hammer Inserts. We now consider the hammer insertion distribution, where we always insert the elements at the same rank. We show that the analysis from sequential insertion distribution (Theorem 28) applies here.

Theorem 30 *When inserted elements have fixed rank (hammer inserts), the APMA has $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ amortized memory transfers.*

Proof. In the hammer-insert case, we always insert new elements after a given element x. Notice that in the rebalancing subtree rooted at u_ℓ, there is a unique path from the leaf node containing the element x to the root node u_ℓ. Let node u_i $(i \le \ell)$ be the ancestor of x at level i, and let v_i be u_i's sibling. An important difference between this proof and the proof of Theorem 28 is that u_{i-1} and sibling v_{i-1} may now be either left or right children of u_i for $i < \ell$.

Recall that, as in the proof of Theorem 28, for level $\kappa \le \ell$, $\mathcal{N}_\kappa(\ell,t)$ is the number of sweeps of the leftmost node u_κ at level κ between the $(t-1)$th sweep and the tth sweep of node u_ℓ.

Intuitively, we want to use a similar argument as in the proof of Theorem 28, to show that $\mathcal{N}_\kappa(\ell,1)$ is bounded as in (87), up to a constant factor, that is, for constant β,

$$\mathcal{N}_\kappa(\ell,1) \le \beta 2^{\ell-\kappa+1}.$$

This approach comes close to working, but requires a much more technical generalization. In particular, as we show, Recurrences (83) and (85) still hold, but there is one value of $i+1$ below which Recurrence (84) might not.

In the following, we explain why there may exist a node u_{i+1} below which Recurrence (84) does not hold. Then we explain that

$$\mathcal{N}_\kappa(i+1,2) = O(2^{i+1-\kappa}),$$

which is the same as the solution of Recurrence (84) up to a constant factor. Finally, we explain why the analysis from Theorem 28 still applies even when there exists such a node u_{i+1}.

We first explain why there may exist a node u_{i+1} for which Recurrence (84) does not hold. To do so, we examine the density of the child u_i after the first sweep of u_{i+1} and demonstrate that $\text{Density}(u_i)$ can be different with sequential inserts and hammer inserts. With sequential inserts, a rebalance tries to put as few elements as possible in u_i and as many elements as possible in v_i without disobeying the upper and lower density thresholds. With hammer inserts, we also want u_i to be as sparse as possible while still maintaining the rebalance property.

But now we have an additional constraint, the ***hammer constraint***, that node x must remain in u_i. What we mean by this additional constraint is the following. Suppose that u_i is a left child, and v_i is a right child. In a rebalance we try to put as few elements as possible in u_i and as many elements as possible in v_i. But if the last element in u_i is x then we cannot reduce the density of u_i any further — the next element to move into v_i is x, but then v_i becomes u_i.

To summarize, there are two cases in which hammer inserts may differ from sequential inserts. The first case is when u_i is a left child and x is the rightmost element in u_i after a sweep of u_{i+1}. The second case is when u_i is a right child and x is the leftmost element in u_i after a sweep of u_{i+1}. In both cases Recurrence (84) may not hold for u_{i+1}. (If x is not in one of these two positions at the end of a rebalance, then the critical constraint is the rebalance property rather than the hammer constraint, as with sequential inserts.)

We now explain that in both cases, the number of sweeps of u_κ between the first sweep and the second sweep of u_{i+1}, $\mathcal{N}_\kappa(i+1,2)$, still has the solution $O(2^{i+1-\kappa})$. When node u_i is a right child and x is the leftmost element in u_i, the bound follows from the analysis in Theorem 28 because the insert pattern of u_i matches the sequential-insert case. The difficult case is when u_i is a left child and x is the rightmost element in u_i after the first sweep of u_{i+1}. We call this the ***tail-insert case***. This case corresponds to a stage beginning after any sweep of u_{i+1} when the element x is the rightmost element in u_i and ending when node u_i reaches its upper threshold, i.e., at the next sweep of u_{i+1}. We call this interval the ***tail-insert stage of*** u_i. Below, we give a bound on the number of sweeps of u_κ in the tail-insert case.

We prove the following claim. The proof is similar to Theorem 28, but significantly more technical.

Claim 31 *Consider the tail-insert stage of u_i: the stage starts after one sweep of u_{i+1} and ends just before the next sweep of u_{i+1}, and x is the rightmost element in u_i at the beginning of the stage. Then the number of sweeps of node u_κ during the stage is $O(2^{i-\kappa})$.*

Proof. We give more details of what happens during the tail-insert stage. During the tail-insert stage, new elements are inserted after x, the rightmost element of u_i at the beginning of the stage. At the end of the stage, node u_i reaches its upper

threshold, which triggers the next sweep of node u_{i+1}. Observe that sweeps occurring during the tail-insert stage do not involve v_i, u_i's sibling. This is because the tail-insert stage ends when u_i reaches its upper threshold, which triggers the sweep of u_{i+1}. We will bound the number of sweeps of u_κ during the tail-insert stage of u_i.

Below, we show that it suffices to prove Claim 31 when the tail-insert stage begins with $\text{Density}(u_i) = \rho_{i+1}$. To do so, we show that the fewer elements there are in u_i at the start of the tail-insert stage, the more sweeps there will be of u_κ (descendant of u_i) during the stage. That is, the number of sweeps of u_κ is maximized when node u_i starts with density ρ_{i+1}, the lowest density possible after a sweep of u_{i+1}.

We now explain why the worst case is when $\text{Density}(u_i) = \rho_{i+1}$. Recall that x is in u_κ, and since all inserts are after x, they are all in u_κ. If node u_i has a low density at the beginning of the stage, then more elements can be inserted after x and into u_κ without triggering a sweep of u_{i+1}, which means that there are more sweeps of u_κ during the stage.

We present additional notation. We define $\mathcal{C}_\kappa(i,t)$ to be the number of sweeps of u_κ between the $(t-1)$th and the tth sweep of u_i since the beginning of the tail-insert stage.[3] We define ***Phase* t *of* u_i** to be the phase starting after the $(t-1)$th sweep of u_i and ending at the tth sweep of u_i since the beginning of the tail-insert stage. Thus, by the above two definitions, the number of sweeps of u_κ in the Phase t of u_i equals $\mathcal{C}_\kappa(i,t)$. To simplify the proof, we constrain the density thresholds τ_0, τ_h, ρ_0, and ρ_h as follows:

$$\tau_0 - \tau_h = \rho_h - \rho_0 \quad \text{and} \quad \tau_0 \le 5\rho_0. \tag{88}$$

For example, setting $\rho_0 = 0.16$, $\rho_h = 0.32$, $\tau_h = 0.64$ and $\tau_0 = 0.8$ satisfies (73)-(76) and (88). Therefore, by (74) and (75), for any $0 \le \ell \le h$ we obtain

$$\tau_\ell + \rho_\ell = \tau_0 + \rho_0 = \tau_h + \rho_h \quad \text{and} \quad \tau_\ell \le 5\rho_\ell. \tag{89}$$

Observe that for any choice of constants ρ_0 and τ_0, there exists a constant β, such that $\tau_0 \le \beta\rho_0$. In this proof, we adopt the constraint that $\tau_0 \le 5\rho_0$ for the sake of

[3] Thus, $\mathcal{C}_\kappa(i,t)$ is defined analogously to $\mathcal{N}_\kappa(i,t)$, except that we begin counting from the beginning of the tail-insert stage rather than from the first insert into the APMA.

relative simplicity; we will explain why the results also carry through if we choose some bigger constant instead.

To establish Claim 31, we decompose the sequence of insertions before the first sweep of u_{i+1} since the beginning of the tail-insert stage into phases of u_i, as defined above. By the similar analysis to that of Theorem 28, we show that there are at most three phases of node u_i before the first sweep of u_{i+1}.

Now we prove that there are at most three phases of u_i in the tail-insert stage of u_i; we do so by analyzing the densities in each phase.

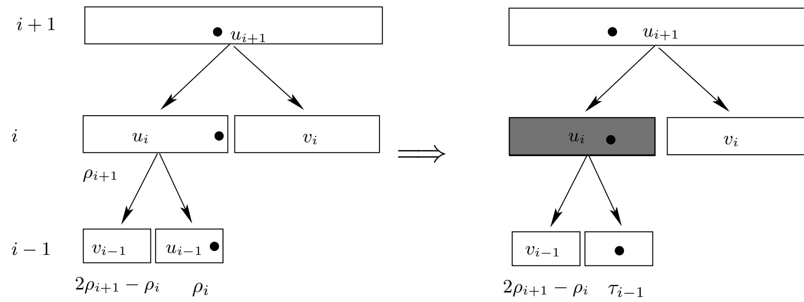

Figure 9: Phase 1 of u_i starts from $\text{Density}(u_{i-1}) = \rho_i$ (left) and ends at $\text{Density}(u_{i-1}) = \tau_{i-1}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of the phase is shaded.

I) Consider the densities of child nodes u_{i-1} and v_{i-1} of node u_i at the end of Phase 1 of u_i. The first sweep of u_i occurs, when u_{i-1} reaches its threshold τ_{i-1} (see Figure 9). After the first sweep of u_i (which is the beginning of Phase 2), we claim that the marker element x is either the leftmost element of the right child of u_i or the rightmost element of the left child of u_i.

We now prove this claim. Notice that the number of elements in u_i before x is $2\rho_{i+1}\text{Cap}(u_{i-1})$ and the number of elements in u_i after x is $(\tau_{i-1} - \rho_i)\text{Cap}(u_{i-1})$. To see why, observe that by assumption the phase begins when $\text{Density}(u_i) = \rho_{i+1}$. Since all inserts are after x, the number of elements before x stays the same. A rebalance of node u_i is triggered when u_{i-1} reaches its threshold, after $(\tau_{i-1} - \rho_i)\text{Cap}(u_{i-1})$ elements have been inserted.

It is legal for u_{i-1} and v_{i-1} to contain $2\rho_{i+1}\text{Cap}(u_{i-1})$ elements and $(\tau_{i-1} - \rho_i)\text{Cap}(u_{i-1})$ elements, and therefore the sweep at level $i-1$ is constrained by

the hammer constraint (not density constraints). Marker element x is always stored in the child having the smaller density (by the hammer constraint). Thus, if there are more elements before x than after x, then x is in the right child of u_i (u_{i-1} is a right child). Otherwise, x is in the left child of u_i (u_{i-1} is a left child).

In the first case, when x is the leftmost element of the right child of u_i, the insert pattern into u_{i-1} in Phase 2 is exactly the head-insert case. Thus, by Theorem 28, the number of sweeps of u_κ in Phase 2 is given by $c_\kappa(i,2) = O(2^{i-\kappa})$.

In the following we consider the second case, when x is the rightmost element of the left child of u_i. Thus, after the first sweep of u_i, by the hammer constraint, we have the the following densities:

$$\begin{aligned}\text{Density}(u_{i-1}) &= 2\rho_{i+1},\\ \text{Density}(v_{i-1}) &= \tau_{i-1} - \rho_i.\end{aligned}$$

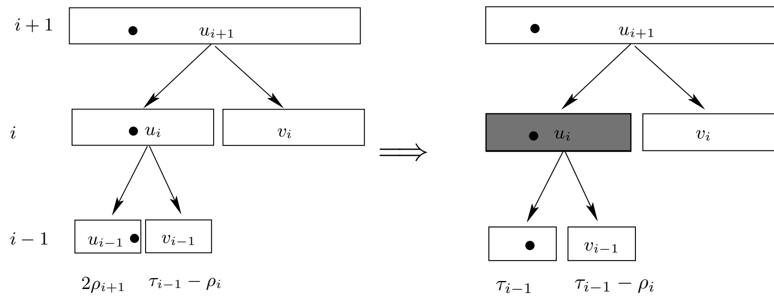

Figure 10: Phase 2 of u_i starts from Density$(u_{i-1}) = 2\rho_{i+1}$ (left) and ends at Density$(u_{i-1}) = \tau_{i-1}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of the phase is shaded.

II) Now (for the above second case) we consider the densities of child nodes u_{i-1} and v_{i-1} of node u_i at the end of Phase 2. The second sweep of u_i occurs when u_{i-1} reaches its upper threshold again (see Figure 10). Recall that at the beginning of the phase, we chose to put the marker element x in the child of u_i having the smaller density, and since we are in the second case, this was

the left child of u_i. Thus, by the hammer constraint,

$$2\rho_{i+1} < \tau_{i-1} - \rho_i\,. \tag{90}$$

When u_{i-1} reaches its threshold, the number of elements after x in u_i is the number of elements in u_{i-1} after x (the new elements inserted in Phase 2) plus the number of elements in v_{i-1}, i.e.,

$$(\tau_{i-1} - 2\rho_{i+1})\mathrm{Cap}(u_{i-1}) + (\tau_{i-1} - \rho_i)\mathrm{Cap}(v_{i-1}). \tag{91}$$

Observe that (91) is greater than $\tau_{i-1}\mathrm{Cap}(v_{i-1})$ by (90). Therefore, the second sweep of u_i is constrained by the rebalance property, not the hammer constraint. In particular, after the second sweep of u_i, node v_{i-1} has density τ_i, the upper threshold of its parent u_i; node u_{i-1} has (the remaining) density $\tau_{i-1} + (\tau_{i-1} - \rho_i) - \tau_i$, which equals $\tau_{i-1} - \rho_{i-1}$ by (89). Thus, after the second sweep, we have the following densities:

$$\begin{aligned}\mathrm{Density}(u_{i-1}) &= \tau_{i-1} - \rho_{i-1},\\ \mathrm{Density}(v_{i-1}) &= \tau_i.\end{aligned}$$

III) We now consider the densities of child nodes u_{i-1} and v_{i-1} of node u_i at the end of Phase 3. (We focus on the above second case in the following, but the first case is now essentially the same.) The third sweep of u_i occurs when u_{i-1} reaches its threshold for a third time (see Figure 11). When u_i does the third sweep, the density of u_i is $(\tau_{i-1} + \tau_i)/2 > \tau_i$, so u_i is above its upper threshold. Thus, the end of Phase 3 is the first sweep of u_{i+1} since the beginning of the tail-insert stage.

We have therefore shown that (for the second case) there are at most three sweeps of u_i before the first sweep of u_{i+1}, that is,

$$C_\kappa(i+1,1) \le C_\kappa(i,1) + C_\kappa(i,2) + C_\kappa(i,3)\,. \tag{92}$$

For the first case, we have the similar recurrence

$$C_\kappa(i+1,1) \le C_\kappa(i,1) + O(2^{i-\kappa}) + C_\kappa(i,3)\,. \tag{93}$$

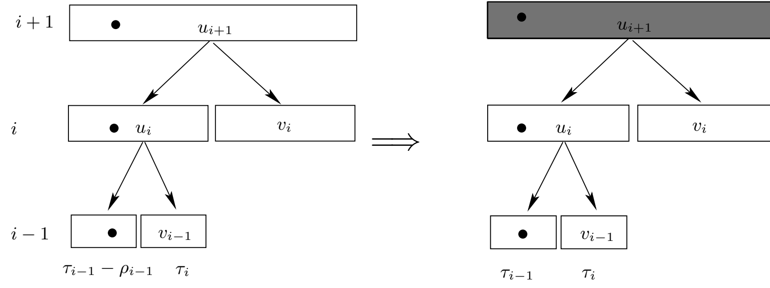

Figure 11: Phase 3 of u_i starts from $\text{Density}(u_{i-1}) = \tau_{i-1} - \rho_{i-1}$ (left) and ends at $\text{Density}(u_{i-1}) = \tau_{i-1}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of the phase is shaded.

As we will show in (94), Recurrence (93) in the first case is actually bounded by Recurrence (92). In the rest of this appendix, we only need focus on (92).

Until now, the proof has been similar to the proof of Theorem 28. However, if we continue to decompose Phase 2, we find that in the worst case there are three subphases. Furthermore, we cannot use the recurrence $\mathcal{C}_\kappa(i,3) \leq \mathcal{C}_\kappa(i,2)$ to prove our bound as in Theorem 28, because the recurrence is true but too weak.

To establish our bound, we instead prove the following recurrences for Phases 2 and 3:

$$\mathcal{C}_\kappa(i,2) \leq \mathcal{C}_\kappa(i-1,1) + \mathcal{C}_\kappa(i-3,1) + O(2^{i-\kappa})\,, \tag{94}$$

and

$$\mathcal{C}_\kappa(i,3) \leq \mathcal{C}_\kappa(i-3,1) + O(2^{i-\kappa})\,. \tag{95}$$

Before we establish Recurrences (94) and (95), we prove the following claim, which describes a subphase in both Phases 2 and 3:

Claim 32 *Consider a tail-insert stage of u_{i-2} starting at* $\text{Density}(u_{i-2}) = 4\rho_{i+1}$ *and ending when node u_{i-2} reaches its upper threshold. The number of sweeps of u_κ during this stage is at most $\mathcal{C}_\kappa(i-3,1)$.*

PROOF OF CLAIM 32: We first give the densities of nodes u_{i-3}, u_{i-4}, v_{i-3}, and v_{i-4} at the beginning of the tail-insert stage of u_{i-2}. We show that the rebalance is constrained by the upper density thresholds of v_{i-3} and v_{i-4}, that is, at the beginning of the tail-insert stage, $\text{Density}(v_{i-3}) = \tau_{i-2}$ and $\text{Density}(v_{i-4}) = \tau_{i-3}$.

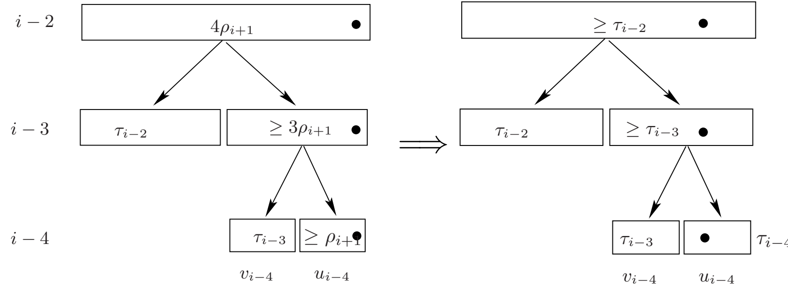

Figure 12: The tail-insert stage of u_{i-2} starts from $\text{Density}(u_{i-4}) \geq \rho_{i+1}$ (left) and ends at $\text{Density}(u_{i-4}) = \tau_{i-4}$ (right). The marker element x is indicated by a black dot. At the end of the tail-insert stage of u_{i-2}, node u_{i-1} is rebalanced.

The tail-insert stage of u_{i-2} begins after a sweep of u_{i-2}, and therefore by the rebalance property

$$\text{Density}(v_{i-3}) \leq \tau_{i-2} \quad \text{and} \quad \text{Density}(v_{i-4}) \leq \tau_{i-3}\,.$$

From (89), we obtain

$$\text{Density}(v_{i-3}) \leq 5\rho_{i-2} \quad \text{and} \quad \text{Density}(v_{i-4}) \leq 5\rho_{i-3}\,.$$

From (73), we obtain

$$\text{Density}(v_{i-3}) \leq 5\rho_{i+1} \quad \text{and} \quad \text{Density}(v_{i-4}) \leq 5\rho_{i+1}\,. \tag{96}$$

Now we bound the densities of u_{i-3} and u_{i-4}. The number of elements in u_{i-3} is the number of elements in u_{i-2} minus the number of elements in v_{i-3} (and similarly for u_{i-4}), that is,

$$\text{Density}(u_{i-3}) = 2\text{Density}(u_{i-2}) - \text{Density}(v_{i-3})\,, \tag{97}$$

$$\text{Density}(u_{i-4}) = 2\text{Density}(u_{i-3}) - \text{Density}(v_{i-4})\,. \tag{98}$$

From (96), we obtain

$$\text{Density}(u_{i-3}) \geq 8\rho_{i+1} - 5\rho_{i+1} = 3\rho_{i+1}\,. \tag{99}$$

Now from (96) and (99),

$$\text{Density}(u_{i-4}) \geq 6\rho_{i+1} - 5\rho_{i+1} = \rho_{i+1}\,. \tag{100}$$

Inequalities (99) and (100) show that at the beginning of the stage, the densities of u_{i-3} and u_{i-4} are above the lower bound thresholds ρ_{i-2} and ρ_{i-3}, respectively, which means that v_{i-3} and v_{i-4} are at their parents' upper thresholds, i.e., $\mathrm{Density}(v_{i-3}) = \tau_{i-2}$ and $\mathrm{Density}(v_{i-4}) = \tau_{i-3}$.

We now explain that when node u_{i-4} reaches its upper threshold, then u_{i-2} also reaches its upper threshold (see Figure 12). This is because when $\mathrm{Density}(u_{i-4}) = \tau_{i-4}$, we already have $\mathrm{Density}(v_{i-4}) = \tau_{i-3}$. Therefore, u_{i-3} is above its upper threshold. We already have $\mathrm{Density}(v_{i-3}) = \tau_{i-2}$, and therefore u_{i-2} is also above its upper threshold.

Therefore, the number of sweeps of u_κ in the tail-insert stage of u_{i-2} is equal to the number of sweeps of u_κ in the tail-insert stage of u_{i-4} (since u_{i-4} is the rightmost grandchild of u_{i-2}; see Figure 12). By the definition of the tail-insert stage, the number of sweeps of u_κ in the tail-insert stage of u_{i-4} (which starts with $\mathrm{Density}(u_{i-4}) \geq \rho_{i+1}$) is less than $\mathcal{C}_\kappa(i-3,1)$ (the number of sweeps of u_κ in the tail-insert stage of u_{i-4} that starts with $\mathrm{Density}(u_{i-4}) = \rho_{i-3}$). □

Now we are ready to prove (95). To do so, we give the densities of the sibling nodes u_{i-2} and v_{i-2} at the beginning of Phase 3. Recall that Phase 3 starts with node u_{i-1} having density $\tau_{i-1} - \rho_{i-1}$, v_{i-1} having density τ_i, and the marker element x residing in u_{i-1}. Since the number of elements before x does not change, node u_{i-1} thus has $2\rho_{i+1}\mathrm{Cap}(u_{i-1})$ elements before x and $(\tau_{i-1} - \rho_{i-1} - 2\rho_{i+1})\mathrm{Cap}(u_{i-1})$ elements (the remaining elements) after x.

We now show that the number of elements after x is smaller than the number of elements before x in node u_{i-1}. Because $\tau_{i-1} \leq 5\rho_{i-1}$ by (89), we obtain

$$\tau_{i-1} - \rho_{i-1} - 2\rho_{i+1} \leq 4\rho_{i-1} - 2\rho_{i+1}.$$

From $\rho_{i-1} < \rho_{i+1}$ by (73), we have

$$\tau_{i-1} - \rho_{i-1} - 2\rho_{i+1} \leq 2\rho_{i+1}\,. \tag{101}$$

Equation (101) says that the number of elements after x is smaller than the number of elements before x. Thus, the marker element x resides in the right child of u_{i-1}, which is u_{i-2}.

We now break Phase 3 of u_i into subphases and bound the number of sweeps of u_κ in the subphases. Subphase t of Phase 3 of u_i is the period between the $(t-1)$th and tth sweeps of u_{i-1}.

Now there are two cases. Case A is that node v_{i-2} has density τ_{i-1}, i.e., this level is constrained by the rebalance property. Then we only have one subphase in Phase 3 of u_i because when u_{i-2} reaches its upper threshold τ_{i-2}, then its parent u_{i-1} has density $(\tau_{i-1}+\tau_{i-2})/2 > \tau_{i-1}$, which means the end of Phase 3.

In the following, we consider Case B when the sweep at level $i-2$ is constrained by the hammer constraint. In Case B, we decompose Phase 3 into two subphases as follows:

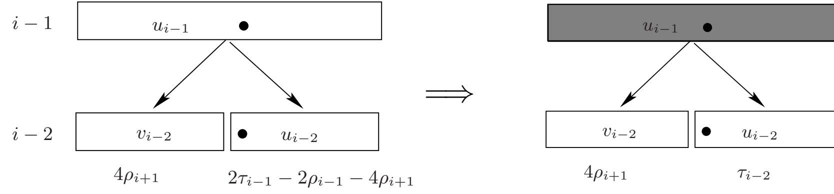

Figure 13: Subphase 1 of Phase 3 starts from Density$(u_{i-2}) = 2\tau_{i-1} - 2\rho_{i-1} - 4\rho_{i+1}$ (left) and ends at Density$(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of Subphase 1 is shaded.

- We consider the densities of u_{i-2} and v_{i-2} at the beginning and end of Subphase 1 (see Figure 13). At the beginning of Subphase 1, because of the hammer constraint, the density of the left child v_{i-2} is $4\rho_{i+1}$ (since the number of elements before x is always $\rho_{i+1}\text{Cap}(u_i)$ – see the beginning of the appendix) and the density of the right child u_{i-2} is $2\tau_{i-1} - 2\rho_{i-1} - 4\rho_{i+1}$ (the remaining elements in node u_{i-1}). At the end of Subphase 1, node u_{i-2} reaches its upper threshold τ_{i-2}.

  Notice that during Subphase 1, the marker element x is the first element in node u_{i-2} and thus within u_{i-2} we have the head-insert case. Therefore, by Theorem 28, there are $O(2^{i-2-\kappa})$ sweeps of u_κ in Subphase 1.
- We now consider the densities of u_{i-2} and v_{i-2} at the beginning and end of Subphase 2 (see Figure 14). The beginning of Subphase 2 is right after the sweep of node u_{i-1}. By the rebalance property, the density of the right child at the beginning of Subphase 2 is τ_{i-1} because before the sweep its density was τ_{i-2} ($> \tau_{i-1}$). After the sweep of node u_{i-1}, the marker element x moves to the left child of u_{i-1}. Therefore, the left child becomes node u_{i-2} and

$\text{Density}(u_{i-2}) = 4\rho_{i+1} + (\tau_{i-2} - \tau_{i-1})$.

Subphase 2 ends when node u_{i-2} reaches its upper threshold τ_{i-2}. Because the density of v_{i-2} is already at parent u_{i-1}'s threshold τ_{i-1}, the end of Subphase 2 is the end of Phase 3.

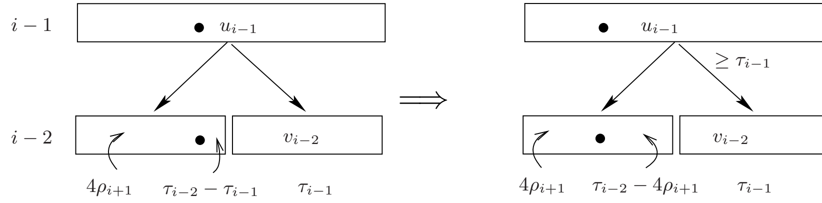

Figure 14: Subphase 2 of Phase 3 starts from $\text{Density}(u_{i-2}) = 4\rho_{i+1} + \tau_{i-2} - \tau_{i-1}$ (left) and ends at $\text{Density}(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. At the end of Subphase 2, node u_i, the parent of u_{i-1}, is rebalanced.

We now prove that the number of sweeps of u_κ in Subphase 2 is less than $\mathcal{C}_\kappa(i-3,1)$, the number of sweeps from Claim 32. Both Subphase 2 and the tail-insert stage of u_{i-2} in Claim 32 end when node u_{i-2} reaches its threshold τ_{i-2}.

However, Subphase 2 starts with more elements after the marker element x than does the tail-insert stage of u_{i-2} and the same number of elements before the marker element x. In particular, Subphase 2 has $4\rho_{i+1}\text{Cap}(u_{i-2})$ elements before and $(\tau_{i-2} - \tau_{i-1})\text{Cap}(u_{i-2})$ elements after x. In contrast, the tail-insert stage of u_{i-2} has $4\rho_{i+1}\text{Cap}(u_{i-2})$ elements before and no elements after x.

Thus, the number of sweeps of u_κ in Subphase 2 is at most the number of sweeps of u_κ in the tail-insert stage of u_{i-2} because fewer elements can be inserted into u_{i-2} before u_{i-2}'s upper threshold is reached.

In summary, there are at most two subphases in Phase 3 and the number of sweeps of u_κ in these two subphases is at most $\mathcal{C}_\kappa(i-3,1)$ plus $O(2^{i-2-\kappa})$, which establishes (95).

We now prove (94). To do so, we decompose Phase 2 of u_i into three subphases, and we analyze the densities of u_{i-2} and v_{i-2} in each subphase.

- We consider the densities of u_{i-2} and v_{i-2} at the beginning and end of Subphase 1 (see Figure 15). At the beginning of Subphase 1, $\text{Density}(u_{i-2}) = \rho_{i-1}$ and $\text{Density}(v_{i-2}) = 4\rho_{i+1} - \rho_{i-1}$ by the rebalance property.

  Here and below we assume that $4\rho_{i+1} - \rho_{i-1} \leq \tau_{i-1}$. The alternative, that $4\rho_{i+1} - \rho_{i-1} > \tau_{i-1}$, is the simple case. Then $\text{Density}(v_{i-2}) = \tau_{i-1}$. As a consequence, there are only two subphases in Phase 2 of u_i, and the recurrence is simpler.

  Subphase 1 ends with the density of u_{i-2} reaching its upper threshold τ_{i-2}. The number of sweeps of u_κ in Subphase 1 is exactly equal to $\mathcal{C}_\kappa(i-1,1)$ because both of them start at $\text{Density}(u_{i-2}) = \rho_{i-1}$ and end with $\text{Density}(u_{i-2}) = \tau_{i-2}$.

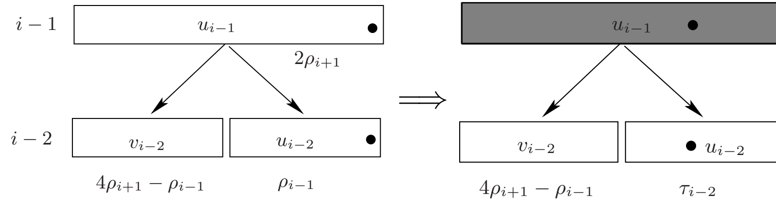

Figure 15: Subphase 1 of Phase 2 starts from $\text{Density}(u_{i-2}) = \rho_{i-1}$ (left) and ends at $\text{Density}(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of Subphase 1 is shaded.

- We next consider the densities of u_{i-2} and v_{i-2} at the beginning and end of Subphase 2 (see Figure 16). The beginning of Subphase 2 is right after the rebalance of u_{i-1}. Notice that there are $4\rho_{i+1}\text{Cap}(u_{i-2})$ elements before the marker element x and $(\tau_{i-2} - \rho_{i-1})\text{Cap}(u_{i-2})$ elements after x. Because $\tau_{i-2} \leq 5\rho_{i-2}$ by (89), we obtain

$$\tau_{i-2} - \rho_{i-1} \leq 5\rho_{i-2} - \rho_{i-1}.$$

  Because $\rho_{i-2} < \rho_{i-1} < \rho_{i+1}$ by (73), we have

$$\tau_{i-2} - \rho_{i-1} < 4\rho_{i+1}\,. \qquad (102)$$

  Equation (102) says that the number of elements after x is less than the number of elements before x in node u_{i-1}. Therefore, the marker element x will be

in the right child of u_{i-1} after the sweep. By the same argument as in Phase 3, we assume the sweep at level $i-2$ is constrained by the hammer constraint. Otherwise, Density$(v_{i-2}) = \tau_{i-1}$, and there are only two subphases in Phase 2.

Thus, we consider the case that v_{i-2} is still below its parent's threshold, i.e., Phase 2 needs a third subphase before it finishes.

We now bound the number of sweeps of u_κ in Subphase 2. Since the marker element x is the leftmost element in u_{i-2}, and thus within u_{i-2} we have the head-insert case. Therefore, by Theorem 28, there are $O(2^{i-2-\kappa})$ sweeps of u_κ in Subphase 2.

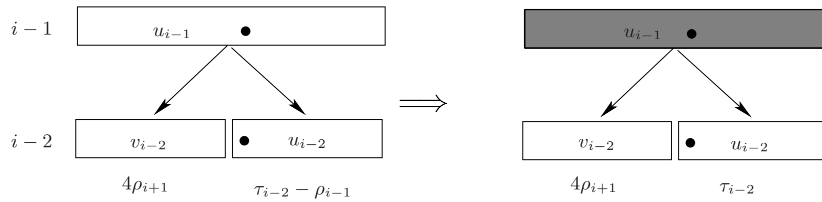

Figure 16: Subphase 2 of Phase 2 starts from Density$(u_{i-2}) = \tau_{i-2} - \rho_{i-1}$ (left) and ends at Density$(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. The region that is rebalanced at the end of Subphase 2 is shaded.

- Finally, we consider the densities of u_{i-2} and v_{i-2} at the beginning and end of Subphase 3 (see Figure 17). Subphase 3 is same as Subphase 2 of Phase 3. By the same argument, the number of sweeps of u_κ in Subphase 3 is $C_\kappa(i-3,1)$.

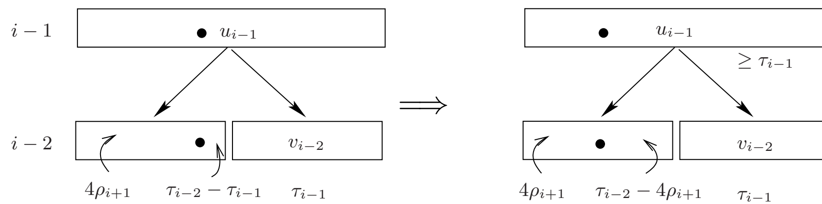

Figure 17: Subphase 3 of Phase 2 starts from Density$(u_{i-2}) = 4\rho_{i+1} + \tau_{i-2} - \tau_{i-1}$ (left) and ends at Density$(u_{i-2}) = \tau_{i-2}$ (right). The marker element x is indicated by a black dot. At the end of Subphase 3, node u_i is rebalanced.

In summary, there are at most three subphases in Phase 2 and the number of sweeps in these three subphases is at most $\mathcal{C}_\kappa(i-1,1)$ plus $O(2^{i-3-\kappa})$ plus $\mathcal{C}_\kappa(i-3,1)$, which establishes (94).

We can now prove our desired bound. Plugging (94) and (95) into (92), we obtain

$$\mathcal{C}_\kappa(i+1,1) \leq \mathcal{C}_\kappa(i,1) + \mathcal{C}_\kappa(i-1,1) + 2\mathcal{C}_\kappa(i-3,1) + O(2^{i-\kappa})\,.$$

We prove our bound by induction. Assume $\mathcal{C}_\kappa(j,1) \leq \beta 2^{j-\kappa}$ for $j \leq i$ and the constant in $O(2^{i-\kappa})$ is α. If we choose β bigger than 4α, then

$$\begin{aligned} \mathcal{C}_\kappa(i+1,1) &\leq \beta 2^{i-\kappa} + \beta 2^{i-1-\kappa} + 2\beta 2^{i-3-\kappa} + \alpha 2^{i-\kappa} \\ &= \frac{7}{4}\beta 2^{i-\kappa} + \alpha 2^{i-\kappa} \\ &\leq \beta 2^{i+1-\kappa}. \end{aligned}$$

Therefore, $\mathcal{C}_\kappa(i+1,1) \leq \beta 2^{i+1-\kappa}$ is true for all $i > 0$, as claimed. □

Finally, we show why, given Claim 31, the analysis from Theorem 28 applies to hammer inserts. Recurrence (84) is true above an intermediate node u_i, that is,

$$\mathcal{N}_\kappa(\ell-1,2) \leq 2^{\ell-i-2}\mathcal{N}_\kappa(i+1,2).$$

Moreover, by Claim 31,

$$\mathcal{N}_\kappa(i+1,2) \leq \beta 2^{i+1-\kappa}$$

for some constant β at node u_i. Therefore,

$$\mathcal{N}_\kappa(\ell-1,2) \leq \beta 2^{\ell-\kappa-1}.$$

Thus, the solution for Recurrence (83) is

$$\mathcal{N}_\kappa(\ell,1) \leq 2^{\ell-\kappa+1}\beta,$$

and the theorem follows. □

3.3 Analysis for Random and Bulk Insertions

In the previous section we analyze the sequential and hammer insertion distributions, where the inserts hammer on one part of the PMA. In this section we first analyze random insertion distribution, where we insert after random elements in the array. Then we generalize all of these distributions and consider the bulk insertion distribution.

The bulk insertion distribution for function N^{α}, $0 \leq \alpha \leq 1$, is defined as follows: pick a random element and insert N^{α} elements after it; then pick another element and repeat. Bulk insert generalizes all distributions seen so far: For $\alpha = 0$, we have random inserts, and for $\alpha = 1$, we have sequential or hammer inserts.

Random Inserts. We now give the performance for the traditional PMA and APMA with random inserts. In the traditional PMA or APMA, each insertion causes only a small number of elements to be moved or triggers a recopying of the entire array.

Theorem 33 ([23,43]) *Consider random insertions into a traditional PMA or APMA, in which each new element is inserted after a random element in the PMA or APMA. Whenever the density of the entire array is below the maximum density threshold, then each insert causes* $O(\log N)$ *element moves and* $O(1 + (\log N)/B)$ *memory transfers with high probability, i.e., probability polynomially small in N. Specifically, each insert causes* $O(\alpha \log N)$ *element moves and* $O(1 + \alpha(\log N)/B)$ *memory transfers with probability at least* $1 - 1/N^{\alpha}$.

Even simpler rebalance schemes perform well under random inserts, as shown in [23, 43]. Publications [23, 43] show that there are $O(\log N)$ moves with high probability for random inserts, even with the following simple rebalance procedure: When we insert an element y after an element x, we simply push the elements to the right or left to make room for y. The maximum number of elements moved is $O(\log N)$ with high probability. Thus, for the traditional PMA, as long as the density thresholds in the leaves is a constant less than 1, we need no big rebalances in the tree.

Bulk Inserts. For bulk inserts, we have the following theorem:

Theorem 34 *For bulk inserts with $f(N) = N^{\alpha}$ ($0 \le \alpha \le 1$), the APMA achieves $O(\log N)$ amortized element moves and $O(1 + (\log N)/B)$ memory transfers.*

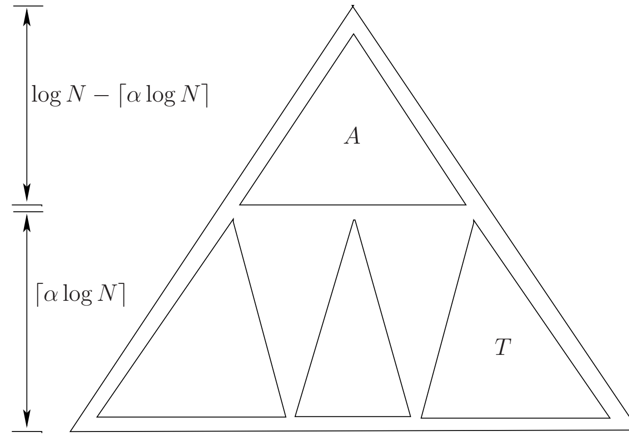

Figure 18: An illustration showing the tree divided at height $\lceil \alpha \log N \rceil$.

The intuition for Theorem 34 is as follows: Conceptually, we divide the virtual tree into a top tree with $\Theta(N/(f(N)\log N))$ leaves, each of which is the root of a bottom tree T with $\Theta(f(N))$ leaves, i.e., $\Theta(f(N)\log N)$ array positions. Thus, we split the virtual tree at height $h' = \lceil \alpha \log N \rceil$. Bulk inserts can be analyzed by looking at the process as a combination of random and hammer inserts: random inserts in the top tree A with big leaf nodes of size $f(N)\log N$ and hammer inserts in a bottom tree T of size $f(N)\log N$. In an insertion, we randomly choose a leaf node of top tree A and do a hammer insert at the bottom subtree of the chosen leaf node of A.

We first show that $f(N) = N^{\alpha}$ ($0 \le \alpha \le 1$) hammer inserts into T costs $O(\log N)$ amortized moves when all the nodes are well balanced. Then, we explain that these $f(N)$ inserts trigger at most one rebalance in the top tree A. Thus, from the point of view of A, there is a big element of size $f(N)$ inserted, and this big insert costs $O(\log N)$ amortized moves in the leaf node.

We prove the following lemma for $f(N) = N^{\alpha}$.

Lemma 35 *Consider inserting $f(N) = N^\alpha$ elements after a fixed element x in subtree T of size $N^\alpha \log N$. Suppose that at the beginning of these insertions, each node in T is well balanced. Then, the amortized number of moves is $O(\log N)$ and the amortized number of memory transfers is $O(1 + (\log N)/B)$.*

Proof. We first show that all sweeps during the insertions of N^α elements occur in subtree T. Because the root node is well balanced, the density of the root is at most $\tau_{h'+1}$. Thus, before root $u_{h'}$ goes outside of its upper threshold, we can insert at least $(\tau_{h'} - \tau_{h'+1})(N^\alpha \log N) = \Theta(N^\alpha)$ elements without triggering sweeps above level h'.

Now we give some assumptions and notation. For simplicity we assume that there are sequential insertions within T. (We know from the proof of Theorem 30 that sequential inserts and hammer inserts have the same analysis except at one level of the recurrence relations.) Now we denote the leftmost node in T at level ℓ as u_ℓ. As in the proof of Theorem 30, we use $\mathcal{N}_\kappa(\ell, i)$ to denote the number of sweeps of node u_κ at level κ between the $(i-1)$th and ith sweep of u_ℓ. Thus, the amortized number of element moves is at most

$$\frac{1}{N^\alpha} \sum_{\kappa=0}^{h'} \mathcal{N}_\kappa(h'+1, 1) 2^\kappa \log N. \tag{103}$$

We bound (103) by considering the worst case when all u_ℓ have density as high as $\tau_{\ell+1}$, $0 \le \ell \le h'$. The time when u_ℓ does its first rebalance is the time when $u_{\ell-1}$ reaches its upper threshold $\tau_{\ell-1}$. This period can be decomposed into two phases, as before:

- Phase 1 of node u_ℓ starts with node $u_{\ell-2}$ having density $\tau_{\ell-1}$ and node $v_{\ell-2}$ having density $\tau_{\ell+1}$. Phase 1 ends with node $u_{\ell-2}$ having density $\tau_{\ell-2}$. Thus, after the first rebalance of $u_{\ell-1}$ (see Figure 19), which occurs after $(\tau_{\ell-2} - \tau_{\ell-1})\mathrm{Cap}(u_{\ell-2})$ inserts, we have densities:

$$\begin{aligned} \mathrm{Density}(u_{\ell-2}) &= \tau_\ell, \\ \mathrm{Density}(v_{\ell-2}) &= \tau_{\ell-1}. \end{aligned}$$

- Phase 2 of node u_ℓ starts with node $u_{\ell-2}$ having density τ_ℓ and ends with node $u_{\ell-2}$ having density $\tau_{\ell-2}$. When node $u_{\ell-1}$ does its second sweep (see

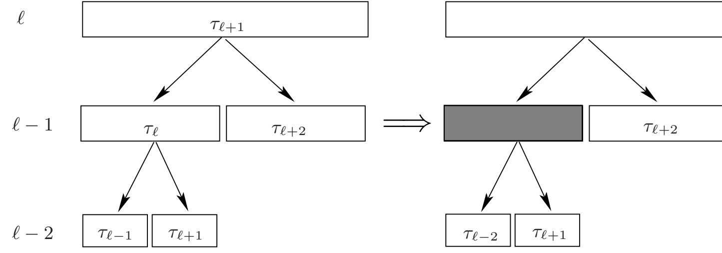

Figure 19: Phase 1 of node u_ℓ starts from Density$(u_{\ell-2}) = \tau_{\ell-1}$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced.

Figure 20), which occurs after $(\tau_{\ell-2} - \tau_\ell)\mathrm{Cap}(u_{\ell-2})$ inserts, the density of node $u_{\ell-1}$ is $(\tau_{\ell-2} + \tau_{\ell-1})/2 > \tau_{\ell-1}$, so node $u_{\ell-1}$ is above its threshold. Thus, the end of Phase 2 is the first rebalance of node u_ℓ.

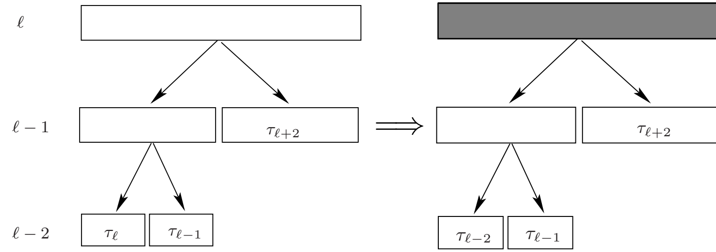

Figure 20: Phase 2 of node u_ℓ starts from Density$(u_{\ell-2}) = \tau_\ell$ (left) and ends at Density$(u_{\ell-2}) = \tau_{\ell-2}$ (right). The shaded region is rebalanced.

Thus, we have recurrence $\mathcal{N}_\kappa(\ell,1) \le \mathcal{N}_\kappa(\ell-1,1) + \mathcal{N}_\kappa(\ell-1,2)$. However, we cannot use the straightforward bound $\mathcal{N}_\kappa(\ell,1) \le 2\mathcal{N}_\kappa(\ell-1,1)$ as we did in Lemma 29. When we try to use this bound, we obtain the solution $\mathcal{N}_\kappa(\ell,1) \le 2^{\ell-\kappa+1}$. Thus, we obtain an amortized number of moves

$$\begin{aligned}\frac{1}{N^\alpha}\sum_{\kappa=0}^{h'} \mathcal{N}_\kappa(h'+1,1)2^\kappa \log N &\le \frac{1}{N^\alpha}\sum_{\kappa=0}^{h'} 2^{h'-\kappa+2}2^\kappa \log N \\ &= O(\log^2 N),\end{aligned}$$

which is greater than our goal of $O(\log N)$. Instead, we need a tighter analysis.

Now we analyze $\mathcal{N}_\kappa(\ell-1,2)$ in more detail. The bound $\mathcal{N}_\kappa(\ell-1,2)$ is the number of sweeps of node u_κ at level κ between the first and second sweeps of $u_{\ell-1}$. After the first rebalance of $u_{\ell-1}$, we have Density$(v_{\ell-2}) = \tau_{\ell-1}$ and Density$(v_{\ell-3}) =$

$\tau_{\ell-2}$ according to our rebalance strategy for the sequential-insert pattern, i.e., both $v_{\ell-2}$ and $v_{\ell-3}$ already have densities as high as their parents' upper thresholds (see Figure 21). The time when $u_{\ell-1}$ does its next sweep is the time when $u_{\ell-2}$ reaches its threshold. Because $v_{\ell-3}$ has density $\tau_{\ell-2}$, this is also the first time when $u_{\ell-2}$ does its next sweep, and because $v_{\ell-4}$ has density $\tau_{\ell-3}$, this is also the first time when $u_{\ell-3}$ does its next sweep, i.e., both $\mathcal{N}_{\ell-2}(\ell-1,2)$ and $\mathcal{N}_{\ell-3}(\ell-1,2)$ are 1. This process continues a number of levels down the tree to be determined below, but not to the leaves.

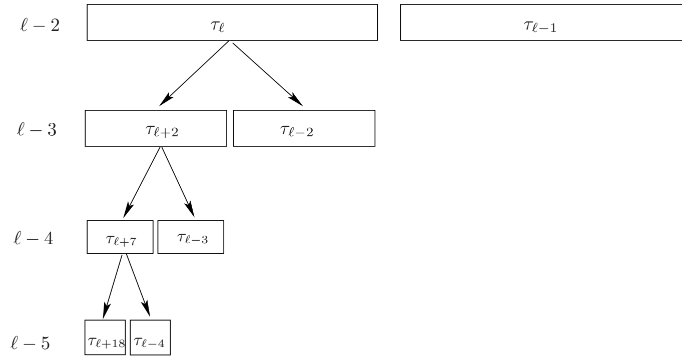

Figure 21: The densities of node u_ℓ's descendants at the beginning of Phase 2 of node u_ℓ.

The process does not continue to the leaves because after the first rebalance of $u_{\ell-1}$, the density of each leftmost child is decreasing from top to bottom. Thus, at some level $\ell-j$, node $u_{\ell-j}$ may be so sparse that there are not enough elements to fill its right child $v_{\ell-j-1}$ to density $\tau_{\ell-j}$. Specifically, we claim that as long as Density$(u_{\ell-j}) \geq \tau_h$, then we can fill $v_{\ell-j-1}$ to density $\tau_{\ell-j}$. Because

$$\begin{aligned}\text{Density}(u_{\ell-j}) &\geq \tau_h \\ &\geq (\tau_h+\rho_h)/2 \\ &= (\tau_{\ell-j}+\rho_{\ell-j})/2\end{aligned}$$

by (74) and (75), we can fill $v_{\ell-j-1}$ to density $\tau_{\ell-j}$ while keeping the density of $u_{\ell-j-1}$ great than $\rho_{\ell-j}$. Thus, there is only one sweep of $u_{\ell-j}$ in Phase 2 of u_ℓ, i.e., $\mathcal{N}_{\ell-j}(\ell-1,2)=1$.

We now calculate the lowest level x such that $\text{Density}(u_x) \geq \tau_h$. First, we give the densities of the nodes above level x after the first rebalance of $u_{\ell-1}$.

Claim 36 *For level $\ell - j > x$,*

$$\text{Density}(u_{\ell-j}) = \tau_{\ell+3\cdot 2^{j-2}-j-1}. \tag{104}$$

The proof of this claim is by induction on j. The base case is $j = 2$. Eq. (104) is satisfied because $\text{Density}(u_{\ell-2}) = \tau_\ell$. Now assume that the claim is true for level $\ell - j$ and all levels above. We show that the claim is also true for level $\ell - j - 1$. Because $\ell - j > x$, $\text{Density}(u_{\ell-j}) \geq (\tau_{\ell-j} + \rho_{\ell-j})/2$. Thus, we can fill $v_{\ell-j-1}$ to density $\tau_{\ell-j}$. Thus, we obtain

$$\begin{aligned}
\text{Density}(u_{\ell-j-1}) &= 2\text{Density}(u_{\ell-j}) - \text{Density}(v_{\ell-j-1}) \\
&= 2\tau_{\ell+3\cdot 2^{j-2}-j-1} - \tau_{\ell-j} \\
&= \tau_{\ell+3\cdot 2^{j-1}-j-2}.
\end{aligned}$$

So (104) is true for level $\ell - j - 1$.

Now we need solve the inequality

$$\tau_{\ell+3\cdot 2^{j-2}-j-1} \geq \tau_h \tag{105}$$

to determine x. Ineq. (105) is equivalent to

$$3 \cdot 2^{j-2} - j - 1 \leq h - \ell.$$

Because $\ell \leq h' = \alpha \log N$ for some fixed constant α, $j = \lg\lg N - O(1)$. That is, the lowest level that x can be is $\ell - \lg\lg N + \lambda_\alpha$, where λ_α is a constant that depends only on α. Thus, we have formula

$$\mathcal{N}_\kappa(\ell - 1, 2) = 1 \tag{106}$$

for $\ell - 1 \geq \kappa \geq \ell - \lg\lg N + \lambda_\alpha$.

For those levels lower than $\ell - \lg\lg N + \lambda_\alpha$, we use simple but straightforward bounds: each sweep of a node costs at most two sweeps of its left child, assuming that each node is within balance. Thus, we have formula

$$\mathcal{N}_\kappa(\ell - 1, 2) \leq 2^{\ell - \lg\lg N + \lambda_\alpha - \kappa}, \tag{107}$$

for $0 \leq \kappa \leq \ell - \lg\lg N + \lambda_\alpha$.

Combining (106) and (107), we obtain

$$\mathcal{N}_\kappa(\ell-1,2) \leq \lceil 2^{\ell-\lg\lg N+\lambda_\alpha-\kappa} \rceil. \tag{108}$$

Now we are ready to bound $\mathcal{N}_\kappa(\ell,1)$ by using (108):

$$\begin{aligned}
\mathcal{N}_\kappa(\ell,1) &\leq \mathcal{N}_\kappa(\ell-1,1) + \mathcal{N}_\kappa(\ell-1,2) \\
&\leq \mathcal{N}_\kappa(\kappa,1) + \sum_{i=\kappa}^{\ell-1} \mathcal{N}_\kappa(i,2) \\
&= 1 + \sum_{i=\kappa}^{\ell-1} \mathcal{N}_\kappa(i,2) \\
&\leq 1 + \sum_{i=\kappa}^{\ell-1} \lceil 2^{i-\lg\lg N+\lambda_\alpha-\kappa} \rceil \\
&\leq \ell - \kappa + 1 + \sum_{i=\kappa+\lg\lg N-\lambda_\alpha}^{\ell-1} 2^{i-\lg\lg N+\lambda_\alpha-\kappa} \\
&\leq \ell - \kappa + 1 + 2^{\ell-\lg\lg N+\lambda_\alpha-\kappa}.
\end{aligned}$$

Finally, we establish that the amortized number of movements for these N^α elements is at most

$$\begin{aligned}
&\frac{1}{N^\alpha} \sum_{\kappa=0}^{h'} \mathcal{N}_\kappa(h',1) 2^\kappa \log N \\
\leq\ & \frac{1}{N^\alpha} \sum_{\kappa=0}^{\lceil \alpha \log N \rceil} (h'-\kappa+1) 2^\kappa \log N + \frac{1}{N^\alpha} \sum_{\kappa=0}^{\lceil \alpha \log N \rceil} 2^{h'-\lg\lg N+\lambda_\alpha-\kappa} 2^\kappa \log N \\
=\ & O(\log N).
\end{aligned}$$

Now we bound the number of memory transfers. Observe that after any insert, the elements moved from a contiguous group, and the moves can be performed with a constant number of scans. Therefore the amortized number of memory transfer is $O(1 + (\log N)/B)$. □

Based on Lemma 35, Theorem 34 is proved as follows.

PROOF OF THEOREM 34: We consider each bottom subtree T. Suppose that an ancestor of the root of T does a rebalance. Then the root of T has density at

most $\tau_{h'+1}$. Thus, we can insert at least $(\tau_{h'} - \tau_{h'+1})\Theta(N^{\alpha}\log N) = \Theta(N^{\alpha})$ elements without triggering sweeps above level h', i.e., inserting N^{α} elements in T triggers at most one rebalance in top subtree A.

Now we consider a ***round*** of N^{α} inserts into some bottom subtree T. We show that there are $O(\log N)$ amortized element moves in the APMA. Recall that we use the predictor to store recent inserts. For the first N^{α} inserts, the predictor only uses one cell. When the next N^{α} inserts start to hammer, the predictor uses the second cell to store new elements. After the count number in the second cell reaches $\log N$, which means there are $\log N$ new elements at the second position, the count number in the first cell begins to decrease. Thus, at most $2\log N$ inserts remove the first cell, meaning that the hammer-insert pattern starts after the first $2\log N$ inserts. Thus, we divide the N^{α} inserts in the round into two parts: the first $2\log N$ ones and the $N^{\alpha} - 2\log N$ subsequent ones. This is one dividing point.

The second dividing point is when some insert triggers a rebalance in the top subtree A. We assume the second dividing point is after the first one. The alternative is similar to the following analysis, although somewhat easier. These two dividing points split the round into three parts. We analyze the cost of the rebalance in the bottom subtree T for these parts as follows:

1. The rebalance cost for the first part, the insertion of the first $2\log N$ elements, is at most $3N^{\alpha}\log N$. To see why, observe that there exists a node u' of size N^{α}, such that these $2\log N$ elements trigger at most one rebalance above u', by an argument similar to that above. This rebalance is within T, and therefore costs at most $N^{\alpha}\log N$. Thus, the total cost is the cost of this rebalance, at most $N^{\alpha}\log N$, plus the cost of the rebalances below u', at most $(2\log N - 1)N^{\alpha}$.
2. The second part is from the $(2\log N)$th element insert to the element insert triggering the rebalance in the top subtree A. The total cost is at most the worst-case cost in Lemma 35, which is $O(N^{\alpha}\log N)$.
3. The third part is from the element insert triggering the rebalance in the top subtree A to the last element insert of these N^{α} elements. From Lemma 35, the cost is less than the cost to insert all N^{α} elements in subtree T whose ancestor did the rebalance, which is $O(N^{\alpha}\log N)$.

Thus, without counting the rebalance cost in the top subtree A, the average cost for each round is $O(N^{\alpha} \log N)/N^{\alpha} = O(\log N)$. If we can show that the average cost in the top subtree A is also $\log N$, then the theorem is proved.

From the view point of top subtree A, the bulk insert is similar to random inserts of "big elements" of size N^{α} in A, because big element triggers at most one rebalance in A and a leaf node of size $N^{\alpha} \log N$ is a black box that has $O(\log N)$ amortized moves. So the bulk insert is: randomly choose a leaf node in A, a black-box operation to insert N^{α} elements in the leaf node, each with $O(\log N)$ moves. If the leaf node reaches its threshold, then a rebalance is triggered at most once in A. Thus, as in Theorem 33, we have $O(\log N)$ element moves in the top subtree A. As before, the memory-transfer bound follows because all rebalances are to contiguous groups of elements. □

3.4 Experimental Results

In this section we describe our simulation and experimental study. We show that our results are consistent with the asymptotic bounds from the previous sections and suggest the constants involved. We also demonstrate that the bookkeeping for the adaptive structure has little computational overhead.

We ran our experiments as follows: For each insert pattern, we began with an empty array and added elements until the array contained roughly 1.4 million elements. We began our measurements once the array had size at least $100,000$. We recorded the amortized number of element moves per insert as well as the running times. We considered the sequential, hammer, random, and bulk insertion distributions from the previous sections. We also added noise to the distributions, combining, for example, the hammer and random distributions, showing that the predictor is resilient to this noise. Each graph plots the intermediate data points in a single run.

We ran our experiments on a Pentium 4 CPU 3.0GHZ, with 2GB of RAM, running Windows XP professional, and a 100G ATA disk drive. Our file contained up to 2^{21} keys, and the total memory used was up to 1.4 GB. We implemented a search into the PMA as a simple binary search. The binary search was appropriate since our experiments were small enough that they did not involve paging to disk.

Consequently, the search time was dominated by the insertion time into the PMA.

The adaptive PMA is ultimately targeted for used in cache-oblivious and locality-preserving B-trees, where the search time becomes relatively more expensive because the data structures do not fit in main memory. In this case the binary search will be too slow because it lacks sufficient data locality. (The number of memory transfers for the PMA insert is $O(1+(\log N)/B)$, which is dominated by the cost of a binary search, $O(\log\lceil N/B\rceil)$, as well as the optimal external-memory search cost, $O(1+\log_B N)$.) Thus, our next round of experiments on larger data sets is to be run with the objective of speeding up inserts in the cache-oblivious B-tree.

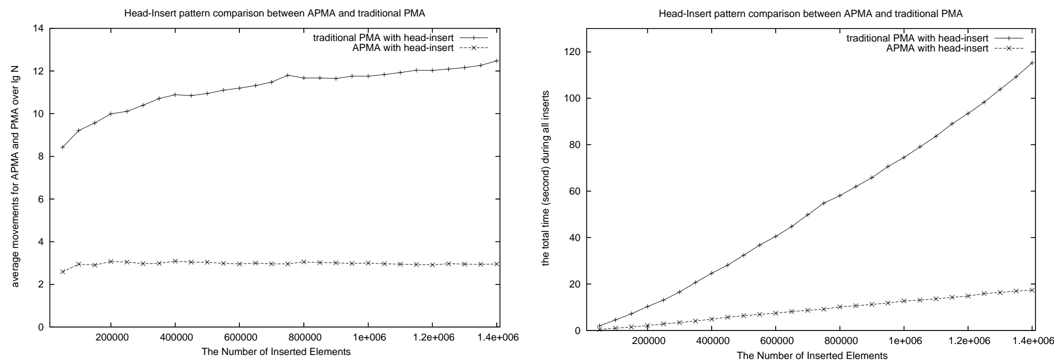

Figure 22: Sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

Figure 23: Sequential inserts: the running time to insert up to 1.4 million elements.

Sequential inserts. We first compared the adaptive and traditional PMAs on sequential insertions. For sequential inserts of roughly 1.4 million elements, the APMA has four times fewer element moves per insertion than the traditional PMA and running times that are nearly seven times faster.

Figure 22 shows the average number of element moves in the PMAs. The x-axis indicates the number of inserted elements up to 1.4 million. The y-axis indicates the number of element moves divided by $\lg N$. For both the adaptive and traditional PMA, we choose the upper and lower density thresholds as follows: $\tau_0 = 0.92$, $\tau_h = 0.7$, $\rho_h = 0.3$, and $\rho_0 = 0.08$. In our experiments, we double when the array gets too full. Thus, before doubling, the array has density over 0.7 and

after, the array has density over 0.35. (By increasing the array size by only a $(1+\varepsilon)$-factor for constant ε, we can make the density of the entire array at least $(1+\varepsilon)\rho_h$ with only a small additive increase in the number of elements moved. Thus, we can have an array whose density is always arbitrary close to 70% full.) The roughly flat line shows the performance of the APMA. These experiments suggest that the constant in front of the $\lg N$ (see Theorem 28) is roughly 2.5 for the density thresholds chosen. Because we are measuring number of element moves, these results are machine independent. Figure 23 gives the running times for our experiment. Observe that the APMA runs almost 7 times faster even though the amortized number of element moves is only 4 times smaller. Hence, the overhead for the adaptive PMA is small. We suspect that this decrease has to do with caching issues; the APMA has a smaller working set than the traditional PMA.

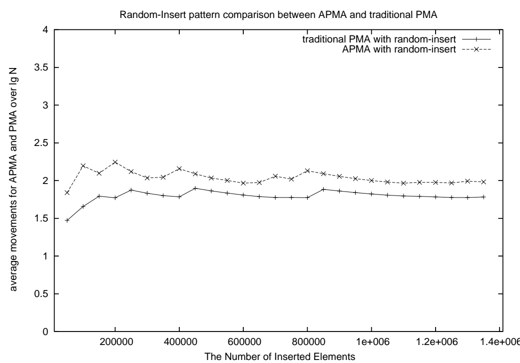

Figure 24: Random inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

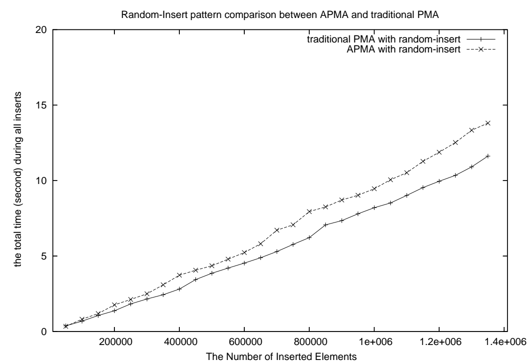

Figure 25: Random inserts: the running time to insert up to 1.4 million elements.

Random inserts. For random insertions the traditional PMA performs slightly better than the APMA because there is seemingly no advantage in uneven rebalances and because the traditional PMA has less overhead. For random insertions of 1.4 million elements with the same density thresholds and axes as in Figures 22 and 23, both the adaptive and traditional PMAs have the same asymptotic performance (see Theorem 33). The traditional PMA's constant seem to be less than 10% smaller. Figures 24 and 25 show that both the amortized number of element moves and the running times are comparable, with the traditional PMA performing

slightly better, as expected. Figure 25 indicates that the bookkeeping overhead for the APMA is small.

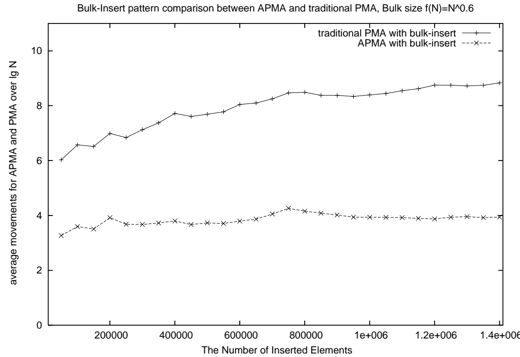

Figure 26: Bulk inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

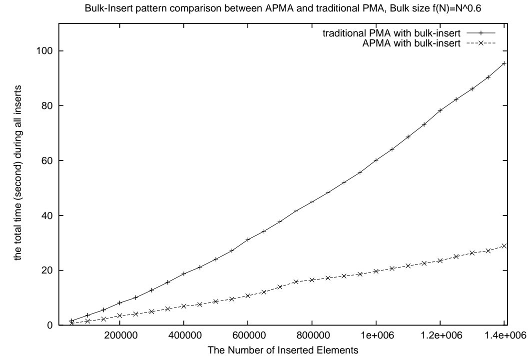

Figure 27: Bulk inserts: the running time to insert up to 1.4 million elements.

Bulk inserts. We next investigated the bulk-insert distribution, comparing both the adaptive and traditional PMAs. For bulk insertions of 1.4 million elements, the APMA has roughly 2.3 times fewer element moves per insertion than the traditional PMA and running times that are over 3.4 times faster. Figure 26 shows the average number of elements moves in the PMAs with the same thresholds as in Figure 22 and bulk parameter $N^{0.6}$. The roughly flat line shows the performance of the APMA. These experiments suggest that the constant in front of the $\lg N$ (see Theorem 34) is roughly 4 for the chosen density thresholds and bulk parameter. Figure 27 shows the running times of the traditional and adaptive PMAs.

Multiple sequential inserts. We next consider a distribution that performs sequential inserts into multiple parts of the array at once. We first choose R random elements and then insert one element at a time after one of these chosen elements. As long as the number of chosen elements R is less than the number of elements stored in the predictor, most predictions are good and the performance of APMA remains $O(\log N)$. Figures 28 and 29 compare the performance of the traditional and adaptive PMAs when we choose 5 fixed elements. The APMA in this case has a performance only slightly worse than that in the sequential-insert case while tradition PMA still performs much worse.

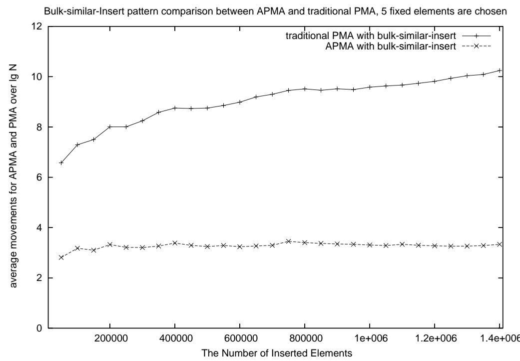

Figure 28: Multiple sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

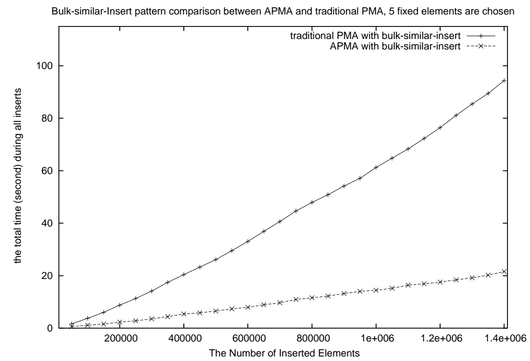

Figure 29: Multiple sequential inserts: the running time to insert up to 1.4 million elements.

Half random and half sequential inserts. Finally, we analyze a distribution that adds noise to sequential inserts. We decide randomly whether to insert a new element at the front of the PMA or after a random element. Thus, roughly half of the inserted elements form random noise. Figures 30 and 31 compare the performance of the traditional PMA and APMA. The roughly flat curve in Figure 30 is the performance of APMA, which is slightly worse than that in random inserts and better than that in sequential inserts, while the performance of traditional PMA is about 3 times worse than that of random inserts.

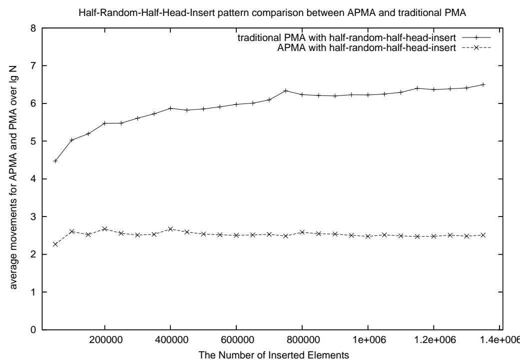

Figure 30: Half random, half sequential inserts: average moves per insert divided by $\lg N$. The array size grows to two million and 1.4 million elements are inserted.

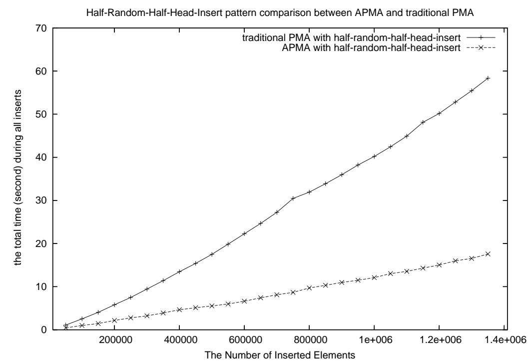

Figure 31: Half random, half sequential inserts: the running time to insert up to 1.4 million elements.

3.5 Conclusion

We introduced an adaptive packed-memory array. The adaptive PMA guarantees a performance at least as good as that of the traditional PMA, while simultaneously adapting to common insertion distributions. Thus, the adaptive PMA always achieves at most $O(\log^2 N)$ amortized element moves and $O(1+(\log^2 N)/B)$ memory transfers per update, but it achieves only $O(\log N)$ amortized element moves and $O(1+(\log N)/B)$ memory transfers for sequential inserts, hammer inserts, random inserts, and bulk inserts. Our simulations and experiments are consistent with these asymptotic bounds. Several open problems remain. For example, can we show some type of working-set property for an adaptive PMA? Perhaps such an investigation will require study into the design of other predictors. The next step in this research is to use the adaptive PMA in a cache-oblivious B-tree and to measure the speedup obtained for updates.

Chapter 4

Partially Deamortized Packed-Memory Array

In this chapter we introduce the ***partially deamortized packed-memory array*** (PDPMA) for the purpose of decreasing the worst-case cost of one insertion. The idea of such a deamortized data structure is to retain the real performance of the traditional PMA in the amortized sense, while achieving better worst-case bounds. In fact, Willard [66] gives an algorithm to deamortize a similar data structure. However, in addition to appearing too complex to implement, the deamortized data structure has poor data locality. Bender, Cole, Demaine, and Farach-Colton [13] give a better worst-case bound, but it is still unlikely to be implementable.

One of the deficiencies in the traditional PMA is that one element insertion might trigger a rebalance of the whole array, which costs $O(N)$ element moves. In contrast, the amortized number of element moves, $O(\log^2 N)$, is not bad. When we do such an insertion in a massive database, one insertion triggering a scan of the whole database is infeasible. Our partially deamortized packed-memory array is a cache-oblivious data structure whose insert/delete cost per update is at most $O(\sqrt{N}\log N)$ element moves and $O(1+(\sqrt{N}\log N)/B)$ memory transfers, while having the same performance of $O(1+(\log^2 N)/B)$ amortized memory transfers as the traditional PMA.

Before presenting the partially deamortized PMA structure, we give an improved rebalance algorithm, *One-phase rebalance*, which plays the same role as the original rebalance algorithm in the traditional PMA, but is a key to design the

partially deamortized PMA.

4.1 One-Phase Rebalance in PMA

In the traditional PMA, we naively rebalance a node (see Chapter 3 for a definition of rebalance) as follows: we compress all elements to the left part of the node without adding empty spaces; we then evenly space out those elements, proceeding from right to left. This rebalance algorithm requires two phases, and each phase needs to scan the whole node. In contrast, the one-phase rebalance performs a rebalance in a single scan. That is, we move elements directly to their final destinations without the intermediate step of compressing elements at one end of the rebalance interval. So our expected one-phase rebalance performs more efficiently than the traditional PMA and plays a key role in the partially deamortized PMA.

We first introduce some terminology about positions of the elements between rebalances. Recall that the PMA is divided into $\Theta(N/\log N)$ segments, each of which has size $\Theta(\log N)$. Because all the elements in a segment can be adjusted at an additional cost of $\Theta(1+(\log N)/B)$ memory transfers, the rebalance algorithm only needs to keep track of which segment an element belongs to, rather than its actual position in the PMA. In the following, we consider which segment an element resides in instead of its actual position. We say that a segment is ***rightward*** (***leftward***) if it includes an element at the beginning of a rebalance, which moves to another segment at *right (left)* side of this segment at the end of rebalance. Recall that we call a contiguous group of segments a ***window***. Similarly, we say that a window is ***rightward*** (***leftward***) if it includes an element at the beginning of rebalance, which moves to another segment at *right (left)* side of this window at the end of rebalance. Like that a segment can be *both* rightward *and* leftward and can be *neither* rightward *nor* leftward, there also exist a both rightward and leftward window, and a neither righward nor leftward window. To simplify notation, we call the both rightward and leftward segment a ***source segment***, and the neither rightward nor leftward segment a ***sink segment***. Specifically, if a window does not include any rightward (leftward) segment, we call it ***leftward*** (***rightward***) interval. Therefore, a subarray of a rebalancing node is composed of rightward intervals and leftward intervals in turn (see Figure 32).

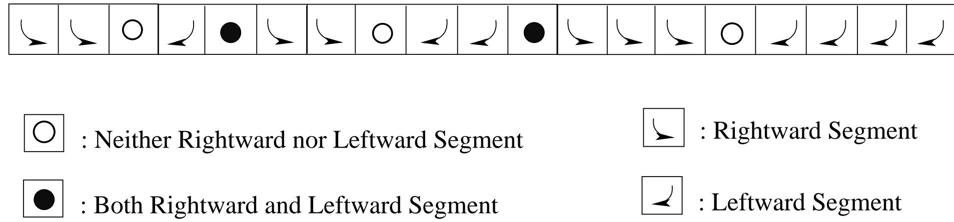

Figure 32: The pattern of one-phase rebalance.

We give further notation. In the rest of this section, we assume that u_ℓ is a node of height ℓ, whose left and right children are $u_{\ell-1}$ and $v_{\ell-1}$.When reblancing a node $u_{h'}$ of height h', for any node u_ℓ in the subtree rooted at $u_{h'}$, we let the number of elements in the node u_ℓ before the rebalance be $\text{Elmt}(u_\ell)$. Noticing that $\text{Elmt}(u_\ell)$ is stored in node u_ℓ, we get it in $O(1)$. Because we do rebalance evenly, the number of elements in a node u_ℓ after rebalance, which we denote $\text{Expt}(u_\ell)$, is computable at the beginning of the rebalance. Basically, for ith segment S_i and jth segment S_j, the number $\text{Expt}(S_i)$ is the same as $\text{Expt}(S_i)$ except for roundoff. Let a window starting from lth segment and ending at rth segment be W_{lr}. Same as the above definitions, $\text{Elmt}(W_{lr})$ and $\text{Expt}(W_{lr})$ are the number of elements in window W_{lr} before and after a rebalance. Notice that number $\text{Elmt}(W_{lr})$ is not available directly. However, we can find it in $O(\log N)$ by tracing along the path from the leaf to the root until the least common ancestor of segments S_l and S_r.

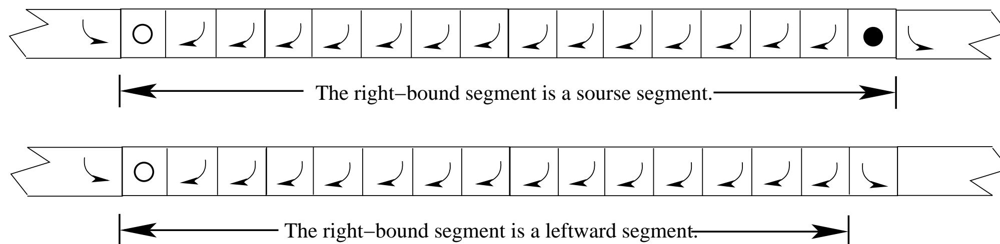

Figure 33: Single leftward intervals.

One-phase Rebalance for an Interval. For the purpose of simplification, we first study the case of rebalancing a single leftward interval, given the left-bound segment S_0 and the right-bound segment S_r. Because the left-neighbor segment of S_0 is a rightward segment, S_0 cannot be either a rightward segment or a source

segment. By the definition of the leftward interval, the segment S_0 is not a leftward segment. Thus, it must be a sink segment. On the other hand, the right-bound segment S_r can be either a leftward segment or a source segment (see Figure 33).

Starting from the left-bound segment S_0, we calculate the number of elements moving to S_0 from its right-neighbor segment S_1. Observe that it is possible that some elements move to S_0 from its left-neighbor segment because S_0 is a sink segment. We cannot calculate it directly by $\text{Expt}(S_0) - \text{Elmt}(S_0)$. Furthermore, because S_r might be a source segment, i.e., some elements may move out of this leftward interval at the place of segment S_r, we do not know exactly who is the rightmost element in this interval.

Algorithm 3 Rebalance.leftward.interval(leftbound)

1: $i \leftarrow \text{leftbound} + 1$.
2: $j \leftarrow$ the last segment in the rebalancing node $u_{h'}$.
3: $W_{ij} \leftarrow$ the window starting from S_i and ending at S_j.
4: Scan and move $\text{Elmt}(W_{ij}) - \text{Expt}(W_{ij})$ elements to S_{i-1}.
5: **while** $\text{Expt}(S_i) > \text{Elmt}(S_i)$ **do**
6:     Scan and move $\text{Expt}(S_i) - \text{Elmt}(S_i)$ elements from S_{i+1} to S_i.
7:     $i \leftarrow i + 1$.
8: **end while**

Algorithm 4 Rebalance.rightward.interval(rightbound)

1: $i \leftarrow$ the first segment in the rebalancing node $u_{h'}$.
2: $j \leftarrow \text{rightbound} - 1$.
3: $W_{ij} \leftarrow$ the window starting from S_i and ending at S_j.
4: Scan and move $\text{Elmt}(W_{ij}) - \text{Expt}(W_{ij})$ elements to S_{j+1}.
5: **while** $\text{Expt}(S_j) > \text{Elmt}(S_j)$ **do**
6:     Scan and move $\text{Expt}(S_j) - \text{Elmt}(S_j)$ elements from S_{j-1} to S_j.
7:     $j \leftarrow j - 1$.
8: **end while**

However, we know the rightmost element in the rebalancing node $u_{h'}$. We consider the window $W_{1\infty}$ starting from S_1 and ending at the last segment in $u_{h'}$. Because no elements will move out of the rightmost segment in $u_{h'}$, the elements that move out of the window $W_{1\infty}$ only exist at the place of segment S_1. Thus, the number of elements that move to S_0 is $\text{Elmt}(W_{1\infty}) - \text{Expt}(W_{1\infty})$. Therefore, we scan

and move $\text{Elmt}(W_{1\infty}) - \text{Expt}(W_{1\infty})$ elements to S_0 from S_1 to finish sweeping the segment S_0.

We now proceed the second segment S_1. Notice that the segment S_1 has its status changed from the original leftward segment to a sink segment now. Furthermore, we know S_1 only receives elements from its right-neighbor segment S_2 because S_0 is a sink segment. The number of elements that the segment S_1 expect from the segment S_2 is $\text{Expt}(S_1) - \text{Elmt}(S_1)$. Thus, we scan and move $\text{Expt}(S_1) - \text{Elmt}(S_1)$ elements from segment S_2 to finish sweeping S_1. Repeat this process until all segments in the leftward interval are swept. Thus, this algorithm rebalances the leftward interval in one scan. Similarly, if we know the right-bound segment of a rightward interval, this algorithm performs the rebalance in one scan of the interval also (see Algorithms 3 and 4).

One-phase Rebalance in General. We now study the general case. As mentioned above, in general, the rebalancing node has the pattern of rightward intervals and leftward intervals in turn (see Figure 32). The problem is how to find those bounds between the rightward and leftward intervals without scanning the whole node. In the following, we present the algorithm about finding those bounds. We start from the first segment S_0. If it is in a leftward interval, then we use Algorithm 3 as a subroutine to sweep this interval until we reach the first segment which is in a rightward interval. Thus, without loss of generosity, we assume the first segment S_0 is in a rightward interval. Notice that we cannot use the above Algorithm 4 as a subroutine because we do not know the right-bound segment of this rightward interval.

Now we present another subroutine (see Algorithm 5), whose input is the left-bound segment S_0 of a rightward interval and whose output is a window starting from S_0 and ending at S', which is the right-bound segment of a rightward interval (see Claim 37). Recall that S_0 is the left bound of a rightward interval, so we have $\text{Elmt}(S_0) > \text{Expt}(s_0)$. Thus, lines 2—8 in Algorithm 5 show that we trace up the tree from S_0 until we find the nearest ancestor u which satisfies either $\text{Elmt}(u) \leq \text{Expt}(u)$ or the rightmost leaf of u is a source segment. There always exists one ancestor because we know that the root node is such an ancestor where $\text{Elmt}(\text{root}) \leq \text{Expt}(\text{root})$. Next, we go down the tree by checking the same condition for child

nodes. That is, if a window W ending at the rightmost leaf of u's left child satisfies $\text{Elmt}(W) \leq \text{Expt}(W)$, we go down to u's left child. Otherwise, we move to u's right child (see lines 5—12 in Algorithm 5).

Algorithm 5 rightbound.window(S_0)

1: $u \leftarrow$ parent of S_0.
2: **while** $\text{Elmt}(u) > \text{Expt}(u)$ **do**
3:     **if** The rightmost leaf of $u \neq$ a source segment **then**
4:         $u \leftarrow$ parent of u.
5:     **else**
6:         BREAK;
7:     **end if**
8: **end while**/* Now $\text{Elmt}(u) \leq \text{Expt}(u)$. */
9: **while** u is not a segment **do**
10:     $W \leftarrow$ a window from S_0 to the rightmost leaf of u's left child.
11:     **if** $\text{Elmt}(W) \leq \text{Expt}(W)$ **then**
12:         $u \leftarrow$ left child of u.
13:     **else**
14:         $u \leftarrow$ right child of u.
15:     **end if**
16: **end while**
17: $S' \leftarrow u$.
18: return a window starting from S_0 and ending at S'.

We give the following claim showing that S' is the right-bound segment of a rightward interval.

Claim 37 *The segment S' in Algorithm 5 is the right bound of a rightward interval and the cost to find S' is $O(\log N)$.*

Proof. There might exist several segments that are the right-bound segment of a right interval. We want to show that the segment S' returned by Algorithm 5 is one of them.

Assume that the tree node u is the nearest ancestor such that the window W_{0r} from S_0 to the rightmost leaf S_r of u satisfies $\text{Elmt}(W_{0r}) \leq \text{Expt}(W_{0r})$ or S_r is a source segment. We first prove that S' must be one of leaf nodes of the tree node u.

There are two cases. In this first case that $\text{Elmt}(W_{0r}) \leq \text{Expt}(W_{0r})$, the rightmost leaf S_r must be either a leftward segment or a sink segment. If S_r is a sink

segment, then it is a right-bound segment of a right interval. If S_r is a leftward segment, then there is at least one segment S', which is a right-bound segment of a right interval, between S_0 and S_r because S_0 is a rightward segment. In the second case that S_r is a source segment, it is similar to the case that S_r is a leftward segment.

We then trace down the tree. The rightmost leaf segment of u'left child must be a rightward segment because if it is one of other three segments, node u is not the nearest ancestor as we find in Algorithm 5. Thus, we know there exists at least one segment S' in the right child of node u. Similarly, when we choose the left child or the right child in Algorithm 5, we always guarantee that there exists at least one segment S' in the corresponding node. Thus, when the node u goes down along the path, it becomes a leaf segment. Therefore, it must be one segment S' which is the right-bound segment of a rightward interval.

The cost to find S' is $O(\log N)$ because we spend time in an around trip between leaf nodes and the root. □

Although the window which subroutine returns might just be a single rightward interval starting from S_0, it is also possible that the window includes other leftward intervals inside. We show that as long as it is not a single interval, Algorithm 5 always can find a bound segment inside because when we trace down the tree, we always choose the child which has at least one segment S' inside. Thus, this subroutine will split a window into two smaller windows. By recursively using Algorithm 5, we finish a rebalance in one scan.

4.2 Description of Partially Deamortized PMA

In this section, we describe the structure of the partially deamortized PMA.

We first review the traditional PMA. Recall that we view the traditional PMA in terms of a tree structure, where the nodes of the tree are windows (i.e., a contiguous group of segments). The tree node at height ℓ have an upper density threshold τ_ℓ and a lower density threshold ρ_ℓ, which together determine the acceptable density of keys within a window. Assume that the height of the tree is $h = \lg N - \lg\lg N + O(1)$. For the root node at height h and leaf nodes at height 0, we select four initial density values of τ_0, τ_h, ρ_0, and ρ_h from $(0,1]$ such that $\rho_0 < \rho_h < \tau_h < \tau_0$. Thus, we define

upper and lower density thresholds for nodes at height ℓ as follows:

$$\begin{aligned}\tau_\ell &= \tau_h + (\tau_0 - \tau_h)(h-\ell)/h \\ \rho_\ell &= \rho_h - (\rho_h - \rho_0)(h-\ell)/h.\end{aligned}$$

Observe that as the node height ℓ increases, the upper density thresholds τ_ℓ decrease and the lower density thresholds ρ_ℓ increase, i.e.,

$$\rho_0 < \rho_1 < \cdots < \rho_h < \tau_h < \cdots < \tau_1 < \tau_0.$$

Another important observation is that

$$\tau_{\ell-1} - \tau_\ell = O(\rho_\ell - \rho_{\ell-1}) = O(1/\log N)\,.$$

In the partially deamortized PMA, we split the virtual tree at height $\hbar = \lceil \lg N/2 \rceil$ (see Figure 34). Hence, the tree is conceptually divided into a top tree A with $\Theta(\sqrt{N}/\lg N)$ leaves, each of which is the root of a bottom tree T_i with $\Theta(\sqrt{N})$ leaves, i.e., $\Theta(\sqrt{N}\lg N)$ array positions. The intuition of splitting tree near half height is that we want to separate big rebalances occurring in the top tree A from small rebalances occurring in the bottom trees.

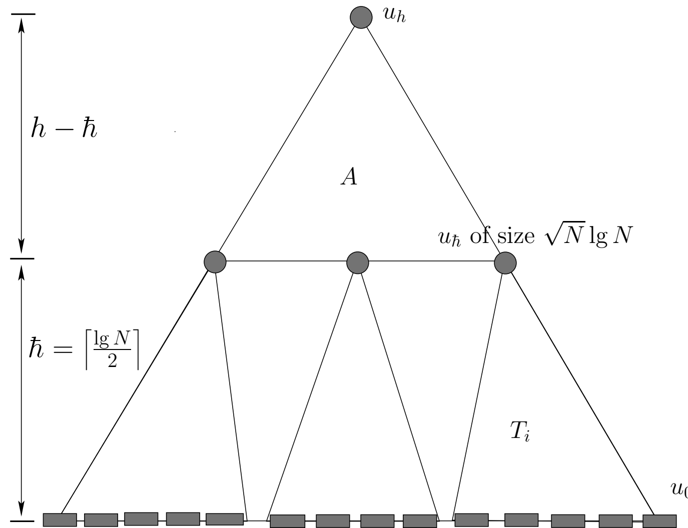

Figure 34: PDPMA model.

Thresholds. We set the thresholds on the top tree A and bottom trees separately. Recall that in the traditional PMA the difference of density thresholds between any two adjacent levels is $(\tau_0 - \tau_h)/h$ and $(\rho_h - \rho_0)/h$. (For simplification, we choose $\rho_h - \rho_0 = \tau_h - \tau_0$.) In the partially deamortized PMA, we set a bigger gap at the node $u_{\hbar}$. To do so, we evenly split the interval $[\rho_0, \rho_h]$ and $[\tau_h, \tau_0]$ into $h+1$ steps instead of the original h steps, and therefore each step has size

$$\Delta = \frac{\tau_0 - \tau_h}{h+1} \quad \left(\text{or } \frac{\rho_h - \rho_0}{h+1}\right).$$

Thus, we have the series of thresholds for nodes at each level.

$$\rho_0 < \cdots < (\rho_{\hbar} < \rho_{\hbar'}) < \rho_{\hbar+1} < \cdots < \rho_h < \tau_h < \cdots < (\tau_{\hbar} < \tau_{\hbar'}) < \tau_{\hbar+1} < \cdots < \tau_0.$$

More formally, we set the density thresholds at each level except for the node $u_{\hbar}$ at height $\hbar$ as follows:

$$\tau_\ell = \begin{cases} \tau_0 - \ell\Delta & 0 \le \ell < \hbar; \\ \tau_h + (h-\ell)\Delta & \hbar < \ell \le h. \end{cases} \tag{109}$$

and

$$\rho_\ell = \begin{cases} \rho_0 + \ell\Delta & 0 \le \ell < \hbar; \\ \rho_h - (h-\ell)\Delta & \hbar < \ell \le h. \end{cases} \tag{110}$$

Observe that the threshold difference between $u_{\hbar-1}$ and $u_{\hbar+1}$ is 3Δ, where we will set the thresholds for node $u_{\hbar}$. Specifically, the node $u_{\hbar}$ at height $\hbar$ has two sets of thresholds. If we view $u_{\hbar}$ as the root of a bottom tree, then its upper and lower thresholds are

$$\tau_{\hbar} = \tau_{\hbar-1} - \Delta \quad \text{and} \quad \rho_{\hbar} = \rho_{\hbar-1} + \Delta.$$

If we view $u_{\hbar}$ as the leaf node of the top tree A, then its upper and lower thresholds are

$$\tau_{\hbar'} = \tau_{\hbar+1} + \Delta \quad \text{and} \quad \rho_{\hbar'} = \rho_{\hbar+1} - \Delta.$$

Insert. Now we give the detail about how to insert a new element y after the existing element x in the partially deamortized PMA. Recall in the traditional PMA, we check whether the leaf node where the element y is inserted is full. If full, we trace up the tree until the nearest ancestor within thresholds and then evenly

rebalance it. In the partially deamortized PMA, we check both the leaf node of a bottom tree and the leaf node of the top tree A, where element y inserts. We try to rebalance early before the leaf node of the top tree A becomes too full.

Specifically, we assume the existing element x is in a leaf node (segment) u_0 of a bottom tree T_i rooted at node $u_{\hbar}$. The node $u_{\hbar}$ is also a leaf node of the top tree A. When we insert an element y after the element x, we check the density of u_0, the leaf node of T_i, and the density of $u_{\hbar}$, the leaf node of A. Thus, there are three cases as follows:

1. The naive case is that both leaf nodes $u_{\hbar}$ and u_0 are within thresholds. Thus, we adjust elements in u_0 to accommodate the new element y with the cost of element moves at most $O(\log N)$.
2. Segment u_0 is out of thresholds while node $u_{\hbar}$ is within thresholds, i.e., $\rho_{\hbar'} \leq \text{Density}(u_{\hbar}) \leq \tau_{\hbar'}$. In this case, we only need to perform rebalance in the bottom tree T_i because the top tree A is still well rebalanced after the new element y is inserted. Notice that we have

   $$\rho_{\hbar} < \rho_{\hbar'} \leq \text{Density}(u_{\hbar}) \leq \tau_{\hbar'} < \tau_{\hbar}.$$

   Thus, the nearest ancestor of u_0 which is within thresholds must be under the bottom tree T_i. So it is consistent with the fact that we do not need rebalance in the top tree A. Thus, the cost of element moves is at most $\Theta(\sqrt{N}\log N)$.
3. The leaf node $u_{\hbar}$ of the top tree A is out of thresholds. Without loss of generosity, we assume that $\text{Density}(u_{\hbar}) > \tau_{\hbar'}$. In this case, no matter whether node u_0 is within or out of thresholds, we are going to perform big rebalance in the top tree A.

Rebalance in the Top Tree. Notice that any rebalance in the top tree A costs at least $\Omega(\sqrt{N}\log N)$ element moves. In order to achieve the goal that every insert has at most $O(\sqrt{N}\log N)$ element moves, we need decompose the rebalance process. The idea is using the one-phase rebalance in the Section 4.1 since it can perform a rebalance by sweeping each leaf node once. Assume there are j leaf nodes in the rebalancing node in the top tree A. Because there are at most $O(\sqrt{N}/\log N)$ leaf nodes in A, j is at most $O(\sqrt{N}/\log N)$. Thus, we can decompose this rebalance into j phases. In each phase, one leaf node is swept and a new element is inserted,

that is, we insert j new elements during those j phases. Thus, each of them costs $\Theta(\sqrt{N}\log N)$ element moves. For those j elements, we use different insert algorithm. When we insert an element, if we detect it is in a phase of a rebalance in the top tree A, we only do rebalance in the bottom tree if necessary and do not check the leaf node in the top tree. Furthermore, we mark those j elements so that they will not be counted in the following phases of the rebalance in the top tree A. That is, we treat those j elements as *shadows* of existing elements until the end of the rebalance in tree A.

We now show that it is consistent that we insert j elements using different insert algorithms. First of all, those j elements will not trigger rebalance above the root node of the bottom tree. Because at the beginning of the first phase, we have the density of the root node $u_{\hbar}$ of each bottom tree at most $\tau_{\hbar'}$. Thus, node $u_{\hbar}$ allows

$$\begin{aligned} \mathrm{Cap}(u_{\hbar})(\tau_{\hbar} - \tau_{\hbar'}) &= (\sqrt{N}\log N)\Delta \\ &= O(\sqrt{N}) \end{aligned}$$

extra insertions before triggering the rebalance in the top tree A. Therefore, those j ($< \sqrt{N}/\log N$) elements will not trigger rebalances above node $u_{\hbar}$ even if all j elements are inserted in the same bottom tree. Secondly, although leaf nodes in the top tree A get extra shadow elements after the rebalance, they are still within thresholds. Notice that each leaf node in a rebalancing node in the top tree A has the density at most $\tau_{\hbar+1}$ after the rebalance, if we do not count the j shadow elements. Thus, even in the worst case of inserting all j shadow elements in one single leaf node, we have the density of that leaf node at most

$$\begin{aligned} \mathrm{Density}(u_{\hbar}) &= \frac{\tau_{\hbar+1}\mathrm{Cap}(u_{\hbar}) + j}{\mathrm{Cap}(u_{\hbar})} \\ &\leq \tau_{\hbar+1} + \frac{\sqrt{N}}{\log N \cdot \sqrt{N}\log N} \\ &= \tau_{\hbar+1} + \frac{1}{\log^2 N}\ , \end{aligned}$$

which is less than the upper threshold $\tau_{\hbar'}$. Therefore, both the top tree and bottom trees are well rebalanced after the j phases, each of which includes one sweep of a leaf node in the top tree A and one insertion of a shadow element in a bottom tree.

We analyze the amortized rebalance cost per insertion or deletion in the following theorem.

Theorem 38 *To insert/delete an element, the partially deamortized PMA achieves at most $O(\sqrt{N}\log N)$ element moves and $O(\sqrt{N}\log N/B)$ memory transfers. In the amortized sense, the partially deamortized PMA has the insert/delete performance in $O(\log^2 N)$ amortized element moves and $O(\log^2 N/B)$ amortized memory transfers per update.*

Proof. We already show that in the worst case, the partially deamortized PMA achieves at most $O(\sqrt{N}\log N)$ element moves and $O(\sqrt{N}\log N/B)$ memory transfers for a single insert.

We now show that in the amortized sense, the performance of the partially deamortized PMA is as good as the traditional PMA. For rebalances occurring in the bottom trees, the cost is the same as that in the traditional PMA. For rebalances occurring in the top tree, there are two differences. One is the upper and lower thresholds, which are slightly lower in the partially deamortized PMA. However, it is essentially the same because both have steps of size $O(1/\log N)$ between any two adjacent levels. The other is that the leaf node in the top tree A might have additional shadow elements up to $j \le O(\sqrt{N}/\log N)$ after the rebalance. We calculate the number of elements inserted in the node u_ℓ ($\ell \ge \hbar$) in the top tree A between two concatenated rebalances. If there are j shadow elements inserted after the first rebalance, the number of element that we can insert before the next rebalance is

$$\tau_\ell \mathrm{Cap}(u_\ell) - (\tau_{\ell+1}\mathrm{Cap}(u_\ell) + j) = \mathrm{Cap}(u_\ell)\Delta - j.$$

Thus, the amortized rebalance cost per insertion at node u_ℓ is

$$\begin{aligned} \frac{\mathrm{Cap}(u_\ell)}{\mathrm{Cap}(u_\ell)\Delta - j} &\le \frac{\mathrm{Cap}(u_\ell)}{\mathrm{Cap}(u_\ell)\Delta - O(\sqrt{N}/\log N)} \\ &= \frac{1}{\Delta - O(\sqrt{N}/(\mathrm{Cap}(u_\ell)\log N))}. \end{aligned} \tag{111}$$

Observe that $\mathrm{Cap}(u_\ell) \ge \mathrm{Cap}(u_\hbar) = O(\sqrt{N}\log N)$ for $\ell \ge \hbar$. Plugging this inequality into (111), we obtain

$$\frac{\mathrm{Cap}(u_\ell)}{\mathrm{Cap}(u_\ell)\Delta - j} \le \frac{1}{\Delta - O(1/\log^2 N)} = O(\log N),$$

because $\Delta = O(1/\log N)$.

In summary, the amortized rebalance cost per insertion at each node either in the top tree or bottom tress is $O(\log N)$. Notice that when we insert or delete an element, we insert or delete within $O(\log N)$ different tree nodes containing this element. Therefore, the total amortized rebalance cost per insertion or deletion is $O(\log^2 N)$ element moves, the same as the cost in the traditional PMA. □

4.3 Conclusion

We design the partially deamortized PMA using one-phase repbalance scheme, which is as good as the traditional PMA in the amortized sense while having the worst-case bound, i.e., at most $O(\sqrt{N}\log N)$ element moves and $O(\sqrt{N}\log N/B)$ memory transfers for a single insert. Therefore, this structure overcomes one of deficiencies in the traditional PMA, that is, one insert might trigger the rebalance of the whole database. However, this structure cannot adopt the uneven rebalances, as presented in Chapter 3. It is interesting open problem how to design a structure combining the advantages of both the partially deamortized PMA and the adaptive PMA. Such a structure would adapt to common insertion patterns and would have a good bound on the maximum cost of a single insert.

Chapter 5

Atomic-key B-tree

In this chapter, we present an ***atomic-key B-tree*** that supports different-size keys. The essential feature of an atomic-key B-tree is that keys are stored and manipulated in their entirety. That is, entire keys are stored in data structure nodes, and entire keys are sent to the comparison function.

As explained earlier in this thesis, the B-tree is a dynamic dictionary storing unit-sized keys. For disk-block-size B, the B-tree supports searches and updates with a cost of $O(\log_B N)$ memory transfers.

Although B-trees still work correctly with different-size keys, they lack (non-trivial) performance guarantees. Roughly speaking, it is better to store small keys in nodes near the top of the tree: when keys are smaller, more can be stored in a node, thus increasing the branching factor and dividing the search space into a larger number of pieces. However, one cannot simply put the smallest keys in the root node: our other objective is to choose keys that are roughly uniformly distributed from the dictionary so that when the search space is divided into pieces, the largest piece is as small as possible. Standard algorithms for building and maintaining B-trees do not take these considerations into account.

In this chapter, we explain how to build an atomic-key B-tree having strong performance guarantees even when keys have different sizes. We achieve the following type of guarantees. Consider two dictionaries of n keys having average size $\hat{k}$, where the keys have different sizes in the first and the same size in the second. Our atomic-key B-tree performs operations in the first dictionary at least as efficiently as the traditional B-tree performs operations in the second. An atomic-key

B-tree cannot attain the efficiency of the string B-tree, but unlike the string B-tree, it retains the structure of the traditional B-tree.

5.1 Static Structure

In this section, we consider the problem of constructing a static tree layout on N different-size, atomic keys. We use greedy algorithm to generate the tree layout, called a static atomic-key B-tree, and guarantee the search performance in this tree layout is as efficient as that in the traditional B-tree when keys have the same size.

We give some notation before presenting the greedy layout. Assume that the average length of these N keys $\{\kappa_i\}$ is $\hat{k}$. For block size B, we define

$$f = \max\left\{3, \left\lfloor \frac{B}{\hat{k}} \right\rfloor\right\}. \tag{112}$$

The idea of our greedy layout is to store small keys near the top of the tree and big keys near the bottom. We keep the tree structure as close as to the traditional B-tree.

We now give the following greedy algorithm to create the root node.

Greedy Algorithm. Divide the N keys into f sets $\{C_i\}_{0 \le i \le f-1}$ and therefore each set contains N/f keys. The first N/f keys go in the first set C_0, the next N/f keys go in the second set C_1, and so on. For each set except for the first and the last sets, we pick the ***representative key*** r_i to be the minimum-length key in each set; we do not need a representative from the first and the last sets. Now we store these $f-2$ representatives $\{r_i\}_{1 \le i \le f-2}$ in the root node of the tree layout as indices. In this way, we create the root node of the static atomic-key B-tree (See Figure 35).

In the following Lemmas 39 and 40, we estimate the size of the root node in the case that the average length $\hat{k} \le B/3$ and the case that $\hat{k} > B/3$. Before that, we introduce additional notation. Let $\hat{c}_i$ be the average length of keys in the ith set C_i and k'_i be the minimum-length key in the set C_i.

Lemma 39 *Suppose that $\hat{k} \le B/3$. Then the root node has size strictly less than B and thus fits within a single memory block.*

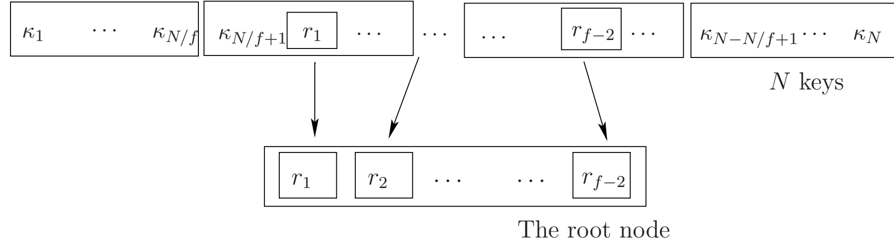

Figure 35: The greedy algorithm for the root node of a static tree layout.

Proof. In the case that $\hat{k} \le B/3$, by (112) we have

$$f = \max\left\{3, \left\lfloor \frac{B}{\hat{k}} \right\rfloor\right\} = \left\lfloor \frac{B}{\hat{k}} \right\rfloor.$$

Because the total length of N keys is the sum of the length of keys in each set C_i, $0 \le i \le f-1$, we have

$$\sum_{i=0}^{f-1} \frac{N}{f} \hat{c}_i = N\hat{k}.$$

Replacing the average key length $\hat{c}_i$ by the smallest key length k'_i for each i, we obtain

$$\sum_{i=0}^{f-1} \frac{N}{f} k'_i \le N\hat{k}.$$

Simplifying the above equation and noticing that $f \le B/\hat{k}$, we have

$$\sum_{i=0}^{f-1} k'_i \le f\hat{k} \le B. \tag{113}$$

Because we store $f-2$ representatives in the root node, the root node has size

$$\sum_{i=1}^{f-2} k'_i,$$

which is less than B by (113). Thus, the root node fits in one memory block. □

Lemma 40 *Suppose that $\hat{k} > B/3$. Then the root contains a single key whose length is at most $3\hat{k}$ and therefore fits in at most $\lceil 3\hat{k}/B \rceil$ memory blocks.*

Proof. In the case that $\hat{k} > B/3$, by (112) we have

$$f = \max\left\{3, \left\lfloor \frac{B}{\hat{k}} \right\rfloor\right\} = 3.$$

Thus, the root node has fanout 2 and contains only a single representative key from the middle set C_1. Noticing that the length of keys in the set C_1 is strictly less than the total length of the N keys, we have

$$N\hat{k} \geq \hat{c}_1 \frac{N}{3},$$

i.e., $\hat{c}_1 \leq 3\hat{k}$. Thus, the root node fits in at most $\lceil 3\hat{k}/B \rceil$ memory blocks. □

We give more notation. Suppose that those $f-2$ representative keys seperate N keys into $f-1$ sets $\{S_i\}_{1\leq i\leq f-1}$ and therefore each set becomes a child of the root node. Assume that the average length of each child set S_i is $\hat{k}_i$.

Greedy Layout. We recursively apply the above greedy algorithm to each child set S_i and thus generate the greedy layout for our static atomic-key B-tree.

We now analyze the search cost in this static atomic-key B-tree in Theorem 42. Assume that the search pattern is uniformly distributed, i.e., every key has the same probability to be searched. We need the following claim to simplify the proof in Theorem 42:

Claim 41 *For all $x_i > 0$ and $x > 0$, we have*

$$\sum_{i=1}^{f-1} t_i \frac{1+x_i}{\ln(2+1/x_i)} \leq \frac{1+x}{\ln(2+1/x)}$$

with the constraints

$$\sum_{i=1}^{f-1} t_i = 1 \quad \text{and} \quad \sum_{i=1}^{f-1} t_i x_i = x.$$

Proof. Let $h(x)$ the function defined for $x > 0$ be

$$h(x) = \frac{1+x}{\ln(2+1/x)}$$

The inequality we want to prove can be rewritten as

$$\sum_{i=1}^{f-1} t_i h(x_i) \leq h\left(\sum_{i=1}^{f-1} t_i x_i\right).$$

The above inequality holds as long as the function $h(x)$ is concave.

To prove the function $h(x)$ is concave, we show that its second derivative is less than zero. We first calculate its first derivative, i.e.,

$$h'(x) = \frac{\ln(2+1/x) + (1+x)/(2x^2+x)}{\ln^2(2+1/x)}.$$

Therefore, its second derivative is

$$h''(x) = \frac{2+2x-(3x+1)\ln(2+1/x)}{(2x^2+x)^2\ln^3(2+1/x)}. \tag{114}$$

Because $x > 0$, we have $\ln(2+1/x) > 0$. Thus, we show that the numerator of (114) is less than zero, i.e.,

$$\ln(2+1/x) > \frac{2+2x}{1+3x}.$$

Let $y = 1/x$. Because x is greater than zero, the range of y is also in $(0,\infty)$. Thus, by replacing $1/x$ by y in the above inequality, we show that for $y > 0$,

$$\ln(2+y) > \frac{2y+2}{y+3} = 2 - \frac{4}{y+3}. \tag{115}$$

We calculate the derivative of the left part in (115):

$$(\ln(2+y))' = \frac{1}{2+y} > 0$$

and the derivative of the right part in (115):

$$\left(2 - \frac{4}{y+3}\right)' = \frac{4}{y^2+6y+9} > 0.$$

Thus, both $\ln(2+y)$ and $2-4/(y+3)$ are monotonically increasing. Furthermore, we show that

$$(\ln(2+y))' \geq (2 - \frac{4}{y+3})'.$$

This is because

$$\frac{1}{2+y} \geq \frac{4}{y^2+6y+9},$$

which is equivalent to $y^2+6y+9 \geq 4y+8$, i.e., $y^2+2y+1 \geq 0$. Therefore, $\ln(2+y)$ increases faster than $2-4/(y+3)$ in $(0,\infty)$. Notice that at the zero, $\ln(2+y)$ has value $\ln 2 \approx 0.69$ greater than the value $2-4/3 \approx 0.67$ of $2-4/(y+3)$ at zero. Thus, we obtain (115) in $(0,\infty)$.

In summary, the second derivative of $h(x)$ is less than zero and $h(x)$ is concave. Therefore, the claim holds. □

Theorem 42 *The greedy layout of a static tree in the DAM model has the expected search cost:*

$$O\left(\left(1+\frac{\hat{k}}{B}\right)\log_{(2+B/\hat{k})} N\right).$$

Proof. In this greedy layout, the root node R consists of $f-2$ (≥ 1) keys. Thus, the root node has $f-1$ children $\{T_i\}_{1\leq i\leq f-1}$, each of which is a subtree on the set S_i.

We prove this theorem by induction on N. Assume that for the subtrees T_i of size $|S_i|$ (less than N elements), the search cost is

$$c\left(1+\frac{\hat{k}_i}{B}\right)\log_{2+B/\hat{k}_i}|S_i|,$$

for some constant c (>0). We show that the search cost applies to the tree of size N also.

The expected search cost in the tree $T=(R,T_1,\ldots,T_{f-1})$ is the number of block transfers to fetch the root node R plus the expected search cost in the corresponding subtree T_i. We first calculate the number of block transfers to fetch the root node R. By Lemmas 39 and 40, we know that in the case that $\hat{k} \leq B/3$, the size of the root node is less than B; in the case that $\hat{k} > B/3$, the size of the root node is at most $\lceil 3\hat{k}\rceil$. In summary, the number of block transfers to fetch the root node is at most $1+3\hat{k}/B$. Thus, to prove the theorem, we show that for the same constant c,

$$1+\frac{3\hat{k}}{B}+\sum_{i=1}^{f-1}\frac{|S_i|}{N}c\left(1+\frac{\hat{k}_i}{B}\right)\log_{2+B/\hat{k}_i}|S_i| \leq c\left(1+\frac{\hat{k}}{B}\right)\log_{2+B/\hat{k}} N \qquad (116)$$

subject to the constraints:

- the tree contains N keys

$$\sum_{i=1}^{f-1} |S_i| = N \tag{117}$$

- the total length of all keys is

$$\sum_{i=1}^{f-1} |S_i|\hat{k}_i = N\hat{k} \tag{118}$$

- and by construction

$$\forall i, 0 < |S_i| < \frac{2N}{f} \tag{119}$$

To simplify, we introduce $x_i = \hat{k}_i/B$, $x = \hat{k}/B$ and $t_i = |S_i|/N$. Thus, Equation (117) becomes

$$\sum_{i=1}^{f-1} t_i = 1, \tag{120}$$

Equation (118) becomes

$$\sum_{i=1}^{f-1} t_i x_i = x, \tag{121}$$

Equation (119) becomes

$$\forall i, 0 < t_i < \frac{2}{f} \tag{122}$$

and Equation (116) becomes

$$1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_i N) \leq c(1+x)\log_{2+1/x} N. \tag{123}$$

To show (123), we simplify its left side first. By (122), we obtain that $t_i N < 2N/f$. Thus, we have

$$1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_i N) \leq 1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(2N/f).$$

Moving $\ln(2N/f)$ out of the summation in the above inequality, we obtain

$$\begin{aligned} & 1+3x+c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) \\ \leq \quad & 1+3x+c\ln(2N/f)\sum_{i=1}^{f-1} t_i\frac{1+x_i}{\ln(2+1/x_i)}. \end{aligned} \tag{124}$$

Plugging the inequality in Claim 41 into (124), we obtain

$$1+3x+c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) \quad \leq \quad 1+3x+c\ln(2N/f)\frac{1+x}{\ln(2+1/x)}.$$

Simplying the above inequality, we obtain

$$\begin{aligned} & 1+3x+c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_iN) \\ \leq \quad & c(1+x)\log_{2+1/x}N+1+3x-c(1+x)\log_{2+1/x}(f/2)\ . \end{aligned} \tag{125}$$

To prove the theorem, we need find the constant c such that the right part in (125) is less than $c(1+x)\log_{2+1/x}N$, that is

$$1+3x-c(1+x)\log_{2+1/x}(f/2)\leq 0.$$

Therefore, we derive that

$$c\geq\frac{1+3x}{1+x}\frac{\ln(2+1/x)}{\ln(f/2)}.$$

Because $(1+3x)/(1+x)=3-2/(1+x)<3$, it is equivalent to find the constant c such that

$$c\geq 3\frac{\ln(2+1/x)}{\ln(f/2)}.$$

To find such constant c, we give the following claim.

Claim 43 *For $x=\hat{k}/B$ and $f=\max\{3,\lfloor B/\hat{k}\rfloor\}$, we have a constant c independent of x and f, such that*

$$c\geq 3\frac{\ln(2+1/x)}{\ln(f/2)}. \tag{126}$$

PROOF OF CLAIM 43: There are two cases.

The first case is when $B/\hat{k} < 3$. In this case, we have $f = 3$ and $1/x > 3$. Thus, we can choose

$$c = \frac{3\ln 5}{\ln(3/2)}$$

such that (126) is true.

The second case is when $B/\hat{k} \geq 3$. In this case, we have $f = \lfloor B/\hat{k} \rfloor = \lfloor 1/x \rfloor$ and $1/x \geq 3$. Therefore, we have

$$\frac{\ln(2+1/x)}{\ln(f/2)} \leq \frac{\ln(3+\lfloor 1/x \rfloor)}{\ln(f/2)} \leq \frac{\ln(2\lfloor 1/x \rfloor)}{\ln(f/2)} = \frac{\ln(2f)}{\ln(f/2)} \leq \frac{\ln 6}{\ln(3/2)}.$$

The first inequality is by $1/x \leq 1 + \lfloor 1/x \rfloor$; The second inequality follows from $\lfloor 1/x \rfloor \geq 3$; the third equation is from $f = \lfloor 1/x \rfloor$ and the last inequality follows by the fact that $\ln(2f)/\ln(f/2)$ is monotonically decreasing and $f \geq 3$. Thus, we can choose $c = 3\ln 6/\ln(3/2)$ such that (126) is true.

In summary, $c = 3\ln 6/\ln(3/2)$ is the constant that (126) is always true in any case. □

In conclusion, by (125) and for the constant c from Claim 43, we prove that

$$1 + 3x + c\sum_{i=1}^{f-1} t_i(1+x_i)\log_{2+1/x_i}(t_i N) \leq c(1+x)\log_{2+1/x} N,$$

which is equivalent to (116). □

To simplify the notation for the search cost, we have the following corollary:

Corollary 44 *The greedy layout of a static tree in the DAM model has the expected search cost:*

$$O\left(\left\lceil \frac{\hat{k}}{B} \right\rceil \log_{1+\lceil B/\hat{k} \rceil} N\right).$$

Proof. By Theorem 42, we have the expected search cost for the greedy layout of a staic tree is

$$O\left((1+\hat{k}/B)\log_{(2+B/\hat{k})} N\right).$$

Because

$$\left\lceil \frac{\hat{k}}{B} \right\rceil \leq 1 + \frac{\hat{k}}{B} \leq 2\left\lceil \frac{\hat{k}}{B} \right\rceil$$

and

$$1+\left\lceil\frac{B}{\hat{k}}\right\rceil \le 2+\frac{B}{\hat{k}} \le 2\left(1+\left\lceil\frac{B}{\hat{k}}\right\rceil\right),$$

we obtain that the search cost is equivalent to

$$O\left(\left\lceil\frac{\hat{k}}{B}\right\rceil \log_{1+\lceil B/\hat{k}\rceil} N\right).$$

□

Building the Tree. We explain how to build the static tree layout efficiently and analyze the building cost for N atomic keys of different sizes.

We first give the cost to read those N keys in the following lemma.

Lemma 45 *For N keys scattered on disk, it takes*

$$O\left(N+\frac{N\hat{k}}{B}\right)$$

memory transfers to read them, where $\hat{k}$ is the average length of N keys.

Proof. Each leaf node in our structure may scatter on disk. For each key κ_i, it costs $\lceil \kappa_i/B \rceil + 1$ block transfers to read. Thus, the total cost of block transfers is at most

$$\sum_{i=1}^{N}\left(\left\lceil\frac{\kappa_i}{B}\right\rceil+1\right) \le 2N+\sum_{i=1}^{N}\frac{\kappa_i}{B} = 2N+\frac{N\hat{k}}{B},$$

as claimed. □

The naive solution is that we build the tree one level by one level, i.e., we generate the root node for all N keys first, then we generate all child nodes of the root node. We continue generating the grandchild nodes at the next level until the leaf nodes. To generate the root node, we need to scan all N keys to pick the right representatives. Noticing that for all keys in the child nodes of the root node (in the second level), they do not share common keys. Therefore, to pick the representatives for all child nodes of the root node takes the cost of scanning all N

keys. In general, to pick the representatives at each level need to scan all N keys once. Because there are roughly $\log_{2+B/\hat{k}} N$ levels, the cost of the naive solution is

$$O\left(\left(N+\frac{N\hat{k}}{B}\right)\log_{(2+B/\hat{k})} N\right).$$

We present the better algorithm whose cost is linear. To do so, we give two algorithms to do the preprocessing job.

- Preprocess the N keys, such that we can get the average length of keys between the ith and jth keys in time $O(1)$. The cost to do this preprocessing is $O(N+N\hat{k}/B)$ and the space is $O(N)$.

  This preprocessing can be done by scans of N keys and store the total length of keys from the 1st key to the ith key into the ith slot of an array of size N.

- Preprocess the N keys, such that we can get the minimal-length key between the ith and jth keys in time $O(1)$. The cost to do this preprocessing is $O(N+N\hat{k}/B)$ and the space is $O(N)$.

  This problem is known as the *Range Minimum Query (RMQ)* [20,31,41]. The idea of this algorithm is to reduce RMQ to the *Least Common Ancestor (LCA)* by constructing a Cartesian trees [40]. Surprisingly, the LCA problem can be reduce back to the special case of RMQ, called RMQ_{-}^{+}. For this special case, we can construct it in linear time by using indirection and answer query in $O(1)$.

Given the above two algorithms, we are ready to build the tree. We start from the root node. By using the first algorithm, we obtain the average key size $\hat{k}$ of all N keys in $O(1)$. Thus, we calculate $f=\max\{3,\lceil B/\hat{k}\rceil\}$ in $O(1)$. Next, We need to pick $f-2$ representative keys from the sets $\{C_i\}_{i\in[1,f-2)}$. For each C_i, we calculate its boundary

$$\left[i\frac{N}{f},(i+1)\frac{N}{f}\right]$$

in $O(1)$. Therefore, the minimal-length key in the above interval is the representative key from the set C_i. By the second algorithm, we obtain it in $O(1)$. Therefore, the total cost to generate the root node is the cost to find the representatives and the

cost to store those representatives into the root node, i.e.,

$$O(f-2)+O\left(\frac{\text{Size(root)}}{B}\right).$$

For each child of the root node, we recursively do the above step until the leaf node. In summary, The total cost to build the tree is the cost to find all representatives and the cost to move each representative to the right tree node. Because the number of representative keys is $O(N)$ and the the cost to move each representative key to the right tree node is $O(N+N\hat{k}/B)$ memory transfers by Lemma 45, we have the following lemma.

Lemma 46 *The cost to build the greedy layout for a static atomic-key B-tree in the DAM model is*

$$O\left(N+\frac{N\hat{k}}{B}\right)$$

memory transfers.

5.2 Dynamic Strcuture

In this section, we consider the problem how to generate a dynamic tree layout for given N keys $\{\kappa_i\}_{i\in[1,N]}$, each of which has different length and is atomic. We present the greedy algorithm of creating the root node. Because the same greedy algorithm applies to all child nodes, we focus on the root node in the rest of this section.

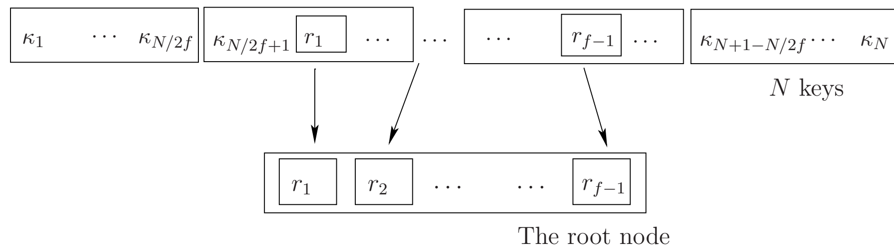

Figure 36: The greedy algorithm for the root node of a dynamic tree layout.

In order to support insert/delete operations in the atomic-key B-tree, we need to modify our greedy layout in Section 5.1. Assume that $\hat{k}$ is the average length of

the N keys and B is the block size. We define

$$f = \max\left\{2, \left\lfloor \frac{B}{\hat{k}} \right\rfloor\right\} \tag{127}$$

and divide the N keys into $f+1$ sets $\{C_i\}_{i\in[0,f]}$, each of which contains N/f keys except for the first set C_0 and the last set C_f. We let the first and last set as half many as the set in the middle, i.e., $|C_0| = |C_f| = N/(2f)$. The reason that we treat the first and last sets differently is that we do not pick the representative keys from them. For the $f-1$ sets in the middle, $\{C_1,\ldots,C_{f-1}\}$, we pick representative keys $\{r_i\}_{i\in[1,f)}$ from each set and store them in the root node as the index (See Figure 36). In Section 5.1, we pick the minimal-length key as a representative key. However, to guarantee the length of the root node, we can choose any key whose length is the order of the average key length $\hat{c}_i$ of a middle set C_i. In this way, we gain flexibility to choose representative keys and therefore we obtain the flexibility to keep the tree balanced.

We give two lemmas to state this problem as follows.

Lemma 47 *Suppose that $\hat{k} \le B/2$. If we choose a representative key r_i from each middle set C_i such that $r_i = O(\hat{c}_i)$, then the root node has size $O(B)$ and thus fits within constant memory blocks.*

Proof. If $\hat{k} \le B/2$, by definition we have

$$f = \max\left\{2, \left\lfloor \frac{B}{\hat{k}} \right\rfloor\right\} = \left\lfloor \frac{B}{\hat{k}} \right\rfloor.$$

Because the total length of the N keys is the sum of the length of each set C_i, $0 \le i \le f$, we obtain

$$N\hat{k} = \sum_{i=1}^{f-1} \frac{N}{f}\hat{c}_i + \frac{N}{2f}\hat{c}_0 + \frac{N}{2f}\hat{c}_f \ge \sum_{i=1}^{f-1} \frac{N}{f}\hat{c}_i.$$

Suppose that we choose a representative r_i from the set C_i such that $r_i \le \beta\hat{c}_i$ for some constant β. Then, the above equality becomes

$$N\hat{k} \ge \sum_{i=1}^{f-1} \frac{r_i}{\beta}\frac{N}{f},$$

which is equivalent to

$$\sum_{i=1}^{f-1} r_i \leq \beta f \hat{k}.$$

Notice that $f = \lfloor B/\hat{k} \rfloor$, i.e., $f\hat{k} \leq B$, we obtain that

$$\sum_{i=1}^{f-1} r_i \leq \beta B,$$

which means that the length of the root node is $O(B)$ and therefore the root node fits in the constant memory blocks. □

Lemma 48 *Suppose that $\hat{k} > B/2$. Then the root contains a single key whose length is $O(\hat{k})$ and therefore fits in $O(\hat{k}/B)$ memory blocks.*

Proof. If $\hat{k} > B/2$, by definition we have

$$f = \max\left\{2, \left\lfloor \frac{B}{\hat{k}} \right\rfloor\right\} = 2.$$

Thus, the root node has fan-out 2 and contains only a single representative from the set C_1. Since the set C_1 of size $N/2$ is a subset of the N keys, the total length of N keys is larger than the total length of the set C_1, i.e., $N\hat{k} \geq \hat{c}_1 N/2$. Suppose that we choose the representative r_1 such that $r_1 \leq \beta \hat{c}_1$. Then, we have

$$N\hat{k} \geq \frac{r_1}{\beta} \frac{N}{2},$$

which is equivalent to $r_1 \leq 2\beta\hat{k}$. Therefore, the root node fits in $O(\hat{k}/B)$ memory blocks. □

Because the representative key is not necessary to be the minimal-length key in the set C_i, we have the flexibility to choose r_i such that it is closer to the key in the middle of C_i. The following lemma gives us the sense how close the representative key can be to the middle key in C_i.

Lemma 49 *The number of keys in the set C_i ($1 \leq i \leq f-1$), whose size is less than $\beta\hat{c}_i$ for some constant β, is at least*

$$\left(1 - \frac{1}{\beta}\right) \frac{N}{f}.$$

Proof. We divide the set C_i into two subsets, C_{i1} and C_{i2}. The set C_{i1} contains keys of size less than $\beta\hat{c}_i$ and the set C_{i2} contains keys of size bigger than $\beta\hat{c}_i$. We denote the number of keys in C_{i1} be $\#_1$ and the number of keys in C_{i2} be $\#_2$.

We estimate the number $\#_2$ in this way. If we do not count the first set C_{i1} and replace the length of each key in the second set C_{i2} by the smaller length $\beta\hat{c}_i$, we have that the total length of keys in C_i is bigger than the number of keys in C_{i2} times the smaller length $\beta\hat{c}_i$, that is,

$$\frac{N}{f}\hat{c}_i \geq \#_2\beta\hat{c}_i.$$

Because $\#_1 = N/f - \#_2$, we obtain

$$\#_1 \geq \frac{N}{f} - \frac{N}{\beta f} = \left(1 - \frac{1}{\beta}\right)\frac{N}{f}.$$

□

Now we give the algorithm to choose the representative key in this dynamic layout. Instead of choosing the minimal-length key in the whole set C_i, we choose the minimal-length key in a smaller subset. Specifically, suppose that the set C_i starts from the $[(i-1/2)N/f]$th key and ends at the $[(i+1/2)N/f]$th key. The smaller subset is an interval

$$\left[\left(i - \frac{1}{2\beta}\right)\frac{N}{f}, \left(i + \frac{1}{2\beta}\right)\frac{N}{f}\right],$$

that is, we choose the minimal-length key from the above interval as the representative key. From Lemma 49, we know that the representative key we choose has the length less than $\beta\hat{c}_i$.

Thus, we guarantee that the representative r_i $(1 \leq i \leq f-1)$ is chosen from the set C_i such that there are at least $(1/2 - 1/(2\beta))N/f$ keys at its left and right sides. Suppose those $f-1$ representatives separate the N keys into f sets $\{S_i\}_{i\in[1,f]}$, each of which belongs to one child of the root node. We have the following lemma:

Lemma 50 *For all child sets S_i, $1 \leq i \leq f$, our greedy layout guarantees that*

$$\left(1 - \frac{1}{\beta}\right)\frac{N}{f} \leq |S_i| \leq \left(1 + \frac{1}{\beta}\right)\frac{N}{f}.$$

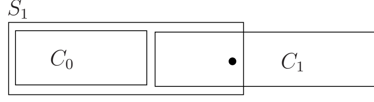

Figure 37: The first set S_1 includes C_0 and part of C_1.

Proof. According to the greedy algorithm, we know that the first set S_1 and the last S_f are different from middle child sets $\{S_i\}_{2\leq i\leq f-1}$.

We first consider the first set S_1, which is the set of keys at the left side of r_1. It contains two parts. One part is the whole set C_0 and the other is the keys at the left side of r_1 in C_1 (See Figure 37). From Lemma 49, we can choose the representative r_1 such that the number of keys at the left side of r_1 is between $(1/2-1/(2\beta))N/f$ and $(1/2+1/(2\beta))N/f$. Thus, we obtain

$$\frac{N}{2f}+\left(\frac{1}{2}-\frac{1}{2\beta}\right)\frac{N}{f}\leq|S_1|\leq\frac{N}{2f}+\left(\frac{1}{2}+\frac{1}{2\beta}\right)\frac{N}{f},$$

that is,

$$\left(1-\frac{1}{2\beta}\right)\frac{N}{f}\leq|S_1|\leq\left(1+\frac{1}{2\beta}\right)\frac{N}{f}. \tag{128}$$

The above result applies to the last set S_f too.

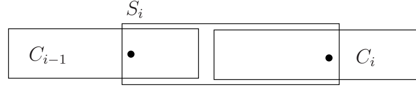

Figure 38: The set S_i includes the right part of C_{i-1} and the left part of C_i.

Now we consider the middle child set S_i ($2\leq i\leq f-1$), which is the set of keys between r_{i-1} and r_i. It contains two parts (See Figure 38). One part is the keys at the right side of r_{i-1} in C_{i-1} and the other is the keys at the left side of r_i in C_i. By the same reason from Lemma 49, we obtain

$$2\left(\frac{1}{2}-\frac{1}{2\beta}\right)\frac{N}{f}\leq|S_i|\leq2\left(\frac{1}{2}+\frac{1}{2\beta}\right)\frac{N}{f},$$

that is,

$$\left(1-\frac{1}{\beta}\right)\frac{N}{f}\leq|S_i|\leq\left(1+\frac{1}{\beta}\right)\frac{N}{f}. \tag{129}$$

Combining both (128) and (129), we obtain that for all child sets S_i, $1 \leq i \leq f$, the results hold. □

In summary, the greedy layout guarantee two properties for the root node:

1) The root node is not too big, that is,

$$\text{Size}(\text{root}) \leq O\left(B+\hat{k}\right),$$

2) Each child set S_i includes keys

$$|S_i| = O(N/f).$$

The greedy algorithm applies to all tree nodes. In this way, we garrantee that each tree node maintains the above two properties. Therefore, we have the following theorem.

Theorem 51 *The greedy layout for dynamic atomic-key B-tree in the DAM model has the expected search cost:*

$$O\left((1+\hat{k}/B)\log_{(2+B/\hat{k})} N\right).$$

Proof. The proof is similar to Theorem 42. □

Building the Tree. Recall that in the previous section, we build the static tree layout in linear time. The algorithm to build the dynamic tree layout is essentially the same, except that we need to find representative keys in a smaller subset. We also use the same two algorithms to do the preprocessing job as in building the static tree. The detail is given as follows.

We start from the root node. By using the first preprocessing algorithm, we obtain the average key-size $\hat{k}$ of all N keys in $O(1)$. Thus, we calculate $f = \max\{2, B/\hat{k}\}$ in $O(1)$. Next, We need to pick $f-1$ representative from the sets $\{C_i\}_{i\in[1,f)}$. For each C_i, we calculate its boundary

$$\left[(i-1/2)\frac{N}{f}, (i+1/2)\frac{N}{f}\right]$$

in $O(1)$. From Lemma 49, we know at least one representative key exists in the interval

$$\left[\left(i-\frac{1}{2\beta}\right)\frac{N}{f},\left(i+\frac{1}{2\beta}\right)\frac{N}{f}\right],$$

for given constant β. Therefore, the minimal-length key in the above interval is one of eligible representative keys from the set C_i. By the second preprocessing algorithm, we obtain it in $O(1)$. Therefore, the total cost to generate the root node is the cost to find the representatives and the cost to store those representatives into the root node, i.e.,

$$O(f)+O\left(\frac{\text{Size(root)}}{B}\right).$$

For each child of the root node, we recursively do the above step until the leaf node. In summary, The total cost to build the tree is the cost to find all representatives and the cost to move each representative to the right tree node. Because the number of representative keys is $O(N)$ and the the cost to move each representative key to the right tree node is $O(N+N\hat{k}/B)$ memory transfers, we prove the following lemma.

Lemma 52 *The cost to build the greedy layout for the dynamic atomic-key B-tree in the DAM model is*

$$O\left(N+\frac{N\hat{k}}{B}\right)$$

memory transfers.

Insertion. We propose our insert algorithm. When we insert an key into the tree, we first search the right place to insert. Then, we check whether the child set S_i (where the new key resides) has the size between $[N/(3f), 5N/(3f)]$ (we choose $\beta = 3/2$). If it is not, we need rebuild this child node according to our greedy algorithm, such that $S_i \in [N/(2f), 3N/(2f)]$ (we choose $\beta = 2$).

In this way, we can insert $\Theta(N/f)$ keys between rebalances. We show that during those inserts, property (1) is always true by giving the following lemma. Assume that the average length of a tree node right after the previous rebuild is $\hat{k}$ and the average length of a tree node just before the next rebuild is $\hat{K}$.

Lemma 53 *For inserts between rebuild of a tree node, they can lower the average key size at most constant factor, i.e.,*

$$\hat{K} = \Omega(\hat{k}).$$

Proof. This is because before the next rebuild, there are at most $\Theta(N/f)$ keys inserted. Noticing that the number of keys under the tree node remains $\Theta(N/f)$, the average key size $\hat{K}$ is at least a constant times the original average key size $\hat{k}$ even if we do not count the length of newly inserted keys. □

Thus, our dynamic tree layout still maintains two properties during inserts. Therefore, the search cost remains the same and the amortized cost per insertion is

$$\frac{O(N + N\hat{k}/B)}{N/f} \log_{(2+B/\hat{k})} N = O\left(\left(\frac{B}{\hat{k}} + \frac{\hat{k}}{B}\right) \log_{(2+B/\hat{k})} N\right).$$

5.3 Dynamic Structure Using Indirection

In the previous section, we built the dynamic atomic-key B-tree in linear time (see Lemma 52). However, its structure loses data locality and therefore the amortized insert cost is not as good as the traditional B-tree in the case that keys have the unit length. Specifically, when keys have the unit length, our dynamic atomic-key B-tree has amortized insert cost of $B \log_B N$ memory transfers while the traditional B-tree has insert cost of $\log_B N$ memory transfers. To preserve the data locality and improve the insert performance of atomic-key B-tree, we use one level of indirection and hence the resulting structure has two layers.

Bottom Layer. The bottom layer includes all N keys, clustered into small groups. In each group, elements are stored in consecutive memory blocks. We cluster those N keys as follows: starting from the first key, we store it in the first group. If the length of the first group is smaller than block size $B/2$, then we store the second key in the first group also. That is, as long as the total length of the first group is smaller than $B/2$, we can store the next key. In another word, the first group ends if

its total length is bigger than $B/2$. We next start putting keys in the second group, and so on.

In general, each group satisfies the following three properties:

1) If all keys are less than $B/2$, then the total length of this group must be in $[B/2, 3B/2]$.
2) If there exists one key bigger than $B/2$, then the total length of other keys in this group must be less than B.
3) At most one key whose length is greater than $B/2$.

Given the above three properties, we present the layout of each group. We store keys in this group as the one-level B-tree. Specifically, we store all keys less than $B/2$ into the root node in order. Because the total length of those keys of length less than $B/2$ is at most $3B/2$, they are stored in at most three contiguous memory blocks. For the key whose length is bigger than $B/2$, it is the only child of the root node. In the case there is no keys of length less than $B/2$, this group only includes a single key of length bigger than $B/2$ and therefore it is stored in the root node of this one-level B-tree.

For the N keys, we split them into groups by the above way. The groups may scatter on disk and the order between groups are saved by address pointers. We count the number of groups in the following lemma.

Lemma 54 *The number of groups for the N keys is at most $2N/f$.*

Proof. Notice that the total length of N keys is $N\hat{k}$ and the total length of each group is bigger than $B/2$. Therefore, the number of groups is at most

$$\left\lceil \frac{N\hat{k}}{B/2} \right\rceil \leq \frac{2N}{\lfloor B/\hat{k} \rfloor} .$$

On the other hand, since each group contains at least one key, the number of groups is at most N. Thus, the number of groups is at most

$$\min\left\{N, \frac{2N}{\lfloor B/\hat{k} \rfloor}\right\},$$

which is equal to $2N/f$ by (127). □

When we insert/delete a key, we first do insert/delete operations in the bottom group. To simplify notations, we classify groups into two types, that is, a group

satisfying the property (1) is called ***type-I group*** and a group satisfying the property (2) or (3) is called ***type-II group*** (See Figure 39). In the following, we give a scheme to do split/merge operations among groups while keeping properties of each group.

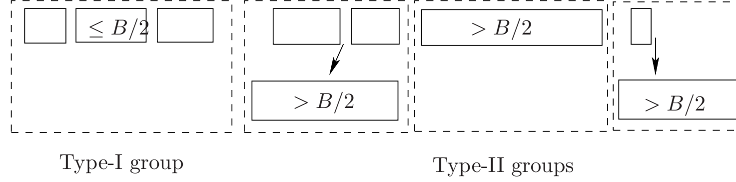

Figure 39: The bottom layer structure, including type-I groups and type-II groups.

We first give some notation. Let the newly inserted key be K_{new}, the unique key of length bigger than $B/2$ in a group be $K_>$ and the set of keys of length less than $B/2$ in a group be $S_\leq$. Assume that the total length of keys in the set $S_\leq$ is $\text{Size}(S_\leq)$.

We first give the scheme how to insert a key in the type-II group.

1. If the inserted key K_{new} has length bigger than $B/2$, then it causes the instant split of this type-II group. Each of the resulting two groups contains either $K_>$ or K_{new}. In this case, the two new groups are both type-II groups.
2. If the inserted key K_{new} has length less than $B/2$, we check the total length of keys which are less than $B/2$ in this group. We have

$$\text{Size}(K_{\text{new}}) + \text{Size}(S_\leq) \leq 3B/2,$$

because $\text{Size}(K_{\text{new}}) \leq B/2$ and $\text{Size}(S_\leq) \leq B$ by the property (2).

Furthermore, if

$$\text{Size}(K_{\text{new}}) + \text{Size}(S_\leq) \leq B,$$

this group is still a type-II group and we do not need split it.

On the other hand, if

$$B < \text{Size}(K_{\text{new}}) + \text{Size}(S_\leq) \leq 3B/2,$$

we separate them into two parts: the set $S'_\leq$ of keys at the left side of the key $K_>$ and the set $S''_\leq$ of keys at the right side of the key $K_>$. Thus, we have

$$B < \text{Size}(S'_\leq) + \text{Size}(S''_\leq) \leq 3B/2, \tag{130}$$

because

$$K_{\text{new}} \bigcup S_{\leq} = S'_{\leq} \bigcup S''_{\leq} .$$

Without loss of generosity, we assume that $\text{Size}(S'_{\leq}) \geq \text{Size}(S''_{\leq})$. Thus, by (130), we obtain

$$B/2 < \text{Size}(S'_{\leq}) \leq 3B/2 \quad \text{and} \quad \text{Size}(S''_{\leq}) < B.$$

Therefore, we can split this type-II group into two groups: one type-I group $\{S'_{\leq}\}$ and one type-II group $\{K_{>} \bigcup S''_{\leq}\}$.

We next present how to insert a key in the type-I group.

1. If the inserted key K_{new} has length bigger than $B/2$, then it is same as the second case of inserting in the type-II group. That is, either we do not need to split this group, or this group can be split into one type-I group and one type-II group.
2. If the inserted key K_{new} has length less than $B/2$, we have

$$\text{Size}(K_{\text{new}}) + \text{Size}(S_{\leq}) \leq 2B, \tag{131}$$

because $\text{Size}(K_{\text{new}}) \leq B/2$ and $\text{Size}(S_{\leq}) \leq 3B/2$ by the property (1). This group need to be split as long as $\text{Size}(K_{\text{new}}) + \text{Size}(S_{\leq}) > 3B/2$. Now we separate those keys into two sets as follows. We put the first key of this group into the first set $S'_{\leq}$. Observe that the length of the first key is less than $B/2$. We continuously put the second key into $S'_{\leq}$ and so on, until the first time when the total length of the set $S'_{\leq}$ is bigger than $B/2$. Other keys belong to the second set $S''_{\leq}$. Thus, because each key has length less than $B/2$, we have

$$B/2 < \text{Size}(S'_{\leq}) < B. \tag{132}$$

Noticing the fact

$$K_{\text{new}} \bigcup S_{\leq} = S'_{\leq} \bigcup S''_{\leq} ,$$

and by (131), we obtain

$$3B/2 < \text{Size}(S'_{\leq}) + \text{Size}(S''_{\leq}) \leq 2B. \tag{133}$$

Therefore, by (132) and (133), we get

$$B/2 < \text{Size}(S''_{\leq}) < 3B/2.$$

In summary, this group can be split into two type-I groups.

Now we present the scheme how to do merge operation. When we delete a key from a group, we do not need merge if the key $K_>$ is still in this group. We need to do merge operation as long as there is only the set $\{S_\leq\}$ in the group and $\text{Size}(S_\leq) < B/2$. Thus, the operation to merge a set $\{S_\leq\}$ of length less than $B/2$ with the adjacent group is same as the operation to insert a key of length less than $B/2$ into the adjacent group. The split might follows right after insert operations.

Therefore, the bottom groups dynamically support operations insert/delete in $O(1)$ memory transfers. Notice that the bottom groups are dynamically maintained. Thus, we do not need extra time to construct the group structures. Furthermore, the bottom layer improves the cost to read those N keys as follows.

Lemma 55 *With the bottom layer, the cost of block transfers to read N keys are improved to*

$$O\left(\frac{N}{f}+\frac{N\hat{k}}{B}\right).$$

Proof. Because there are at most $2N/f$ groups for N keys (by Lemma 54) and the total length of representative leaf keys is less than $N\hat{k}$. Thus, by Lemma 45, the cost to read those groups is

$$O\left(\frac{N}{f}+\frac{N\hat{k}}{B}\right).$$

□

Top Layer. We first give more notation. We divide the type-II group into two subtypes, that is, one only contains one key of length bigger than $B/2$, named a type-II(a) group; the other contains keys of length less than $B/2$ and one key of length bigger than $B/2$, named a type-II(b) group. Recall that the only key which is not stored in the root nodes at the bottom layer is the key (bigger than $B/2$) in the II(b) group. Thus, We view the keys in root nodes at the bottom layer as the ***upper bottom layer*** and therefore the keys (bigger than $B/2$) in the II(b) groups as the ***lower bottom layer***.

The top layer is a dynamic greedy layout on all keys at the upper bottom layer (See Figure 40). We classify the keys at the top bottom layer in a group as a set because keys in each set are still stored in contiguous memory blocks. Thus those

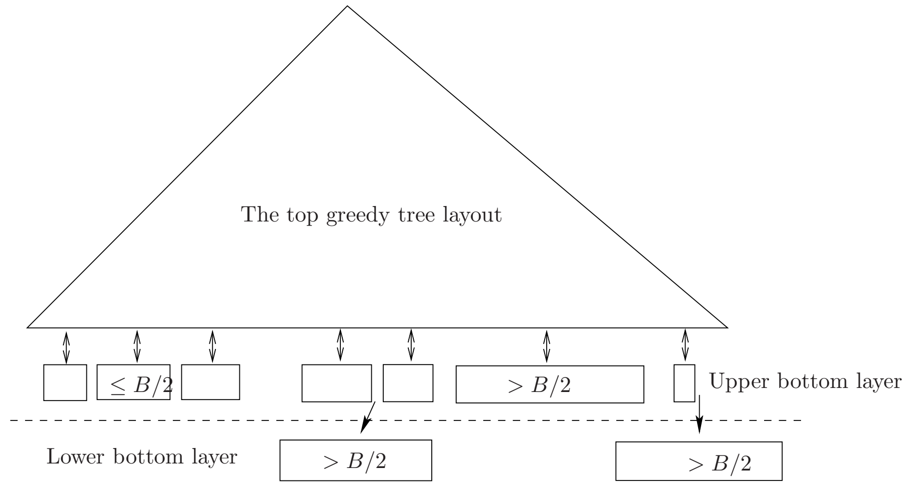

Figure 40: The top layer structure: a greedy tree layout based on the top bottom layer.

sets either contain keys less than $B/2$ or only one key bigger than $B/2$. To simplify notation, we call the set of all keys less than $B/2$ the ***type-I set*** and the set containing only one key bigger than $B/2$ the ***type-II set***. Notice that the number of sets is same as the number of groups, i.e., at most $2N/f$ group. We give the following lemma about the average length of keys at the upper bottom layer.

Lemma 56 *The average length of keys at the upper bottom layer is at most $2\hat{k}$, where $\hat{k}$ is the average length of all N keys.*

Proof. Let the number of keys at the lower bottom layer be $\#_1$, and the number of keys at the upper bottom layer be $\#_2$. Thus, we have $\#_1 + \#_2 = N$. Notice that for each key at the bottom layer, there is at least one corresponding key at the upper bottom layer because of the property of the type-II(b) group. Thus, we have $\#_1 \leq \#_2$. Therefore,

$$\#_2 \geq N/2.$$

Because that the total length of keys at the upper bottom layer is at most $N\hat{k}$. We obtain that the average length of keys at the upper bottom layer is at most

$$\frac{N\hat{k}}{\#_2} \leq \frac{N\hat{k}}{N/2} = 2\hat{k}.$$

□

In the following, we present the greedy algorithm to build the tree on keys at

the upper bottom layer in $O(N/f + N\hat{k}/B)$ memory transfers, by utilizing those at most $2N/f$ upper bottom sets. In this algorithm, we ignore the keys at the lower bottom layer.

We first check the two preprocessing algorithms.

- The algorithm to get the average length between ith and jth keys in $O(1)$. The construct time is improved to $O(N/f + N\hat{k}/B)$ because the time to read those sets is improved by Lemma 55.
- The algorithm to get the minimum-length key between ith and jth keys in $O(1)$ (RMQ). The number of keys at the upper bottom layer might be less than N and the cost to read them is reduced to $O(N/f + N\hat{k}/B)$ by Lemma 55. Thus, the construct time is improved to $O(N/f + N\hat{k}/B)$.

To guarantee the search cost, we still use the same greedy algorithm to build each tree node. Notice that the keys in each bottom set are stored in contiguous memory blocks. If a tree node only contains keys in one or two sets, we do not need branch this tree node and therefore it is treated as a leaf node (because the keys in this tree node can be fetched in contiguous memory blocks). In this way, we "trim" the original tree structure by utilizing the bottom sets.

The purpose of "trimming" the tree is to reduce the cost of building the tree to $O(N/f + N\hat{k}/B)$ memory transfers. It can be achieved by guaranteeing the number of leaf nodes in a trimmed tree at most $O(N/f)$. To do so, we make sure that each leaf node crosses two bottom sets or at least reach the boundary of a set because there are at most $O(N/f)$ bottom sets.

We need additional preprocess algorithm to tell whether a tree node contains keys in one or two sets in $O(1)$. It can be done by creating an array and the ith cell stores the set where the ith key resides. Notice that the cost to create the array is $O(N/B)$ block transfers. Thus, the total cost is dominant by the cost to read keys, i.e., $O(N/f + N\hat{k}/B)$ memory transfers.

Previously, we recursively build the greedy layout until a single key. We now present an algorithm that we stop the recursive step at a tree node whose the keys are at most in two sets (See Figure 41). Given a tree node, we first check how many sets crossed by the third preprocessing algorithm in $O(1)$. If it crosses more than two sets, we use the same greedy algorithm to branch this tree node. Specifically, we fetch representative keys as the indices to its children. Starting from the first

representative key which we calculate in $O(1)$ time, we takes additional $O(1)$ time to calculate the number of sets crossed by the first child. There are three cases. The first case is the number of crossed sets is more than 2. Then, we need take next recursive step for this child; The second case is that the number of crossed sets is 2. In this case, we treat this child as a leaf node. The difficult case is the third case where the child node just fits in one set. We need additional algorithm to deal with this case, because we require that each set is crossed by at most 2 leaf nodes.

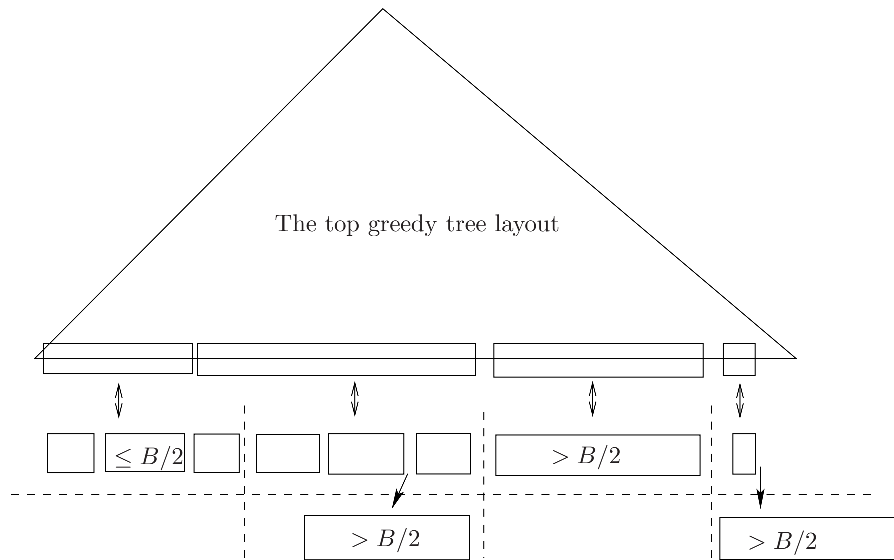

Figure 41: The top tree layout has the leaf nodes trimmed.

We now present how to deal the case that the child node fits in one set. We calculate the second representative key. By the same reason, the second child may have three cases. One case is that the second child fits in the same set also. In this case, we merge the first and second children without extra search cost by removing the first representative key in the tree node, because the first and second children must belong to the type-I set (the root node of the type-II(b) group) and it takes at most three memory transfers to read the type-I set. As long as the next child fits in the same set, we keep merging them together without extra memory transfer. The other two cases are that the second child crosses at least two sets. Assume that the set where the first child resides is S_1. Then, the set S_1 must be a type-I set because it includes more than one key. Now we merge part of keys in the second child into the

first child by replacing the first representative key by the last key in S_1. Thus, the first child reaches the boundary of the set S_1. After the merge operation, the search cost may change at this node because the size of the tree node changed and the number of keys in the first and second children changed. However, we argue that those changes do not affect the search cost by checking the two properties of the greedy layout. The first property is that the size of the tree node must be $O(B+\hat{k})$ (where $\hat{k}$ is the average length of keys under this tree node). This is true because we replace the old key by the last key in S_1, which is less than $B/2$ by the property of the type-I set. The second property is that each child has the number of keys less than $O(N/f)$. For the second child, it satisfies because the number of keys in the second child decreased. For the first child, the number of keys in it may bigger than $O(N/f)$. But we know that the first child fits in a type-I set and therefore it can be fetched in at most three memory transfers. Thus the merge operation increases the search cost at most one memory block at a leaf node.

In summary, we construct a dynamic greedy tree layout such that each leaf node crossing two sets or just reaching the boundary of one set. We now calculate the number of leaf nodes in this tree layout. Notice that each leaf node crosses one boundary of sets and there are no common keys among leaf nodes. Because there are at most $2N/f$ sets, the number of leaf nodes is also at most $O(N/f)$. Furthermore, for each tree node, it has at least two children. Therefore, the number of all inner tree nodes is less than the number of leaf nodes, i.e., the number of all tree nodes is at most $O(N/f)$.

Thus, the time to calculate all representative keys is $O(N/f)$ and the time to store all representative keys is at most $O(N/f+N\hat{k}/B)$ block transfers. Therefore, the total time to build the tree layout is $O(N/f+N\hat{k}/B)$ memory transfers.

Search. To search an element in the structure, we first search the tree in the top layer. Then we go to the corresponding bottom groups followed by the leaf node and the other group pointed by the predecessor of the leaf node.

Insert. To insert a key in the structure, we first search for the right place. Then we insert it into the corresponding group. If it becomes a key at the lower bottom level, we finish insert operation. Otherwise, it appears at the upper bottom level and

we insert it also in the top layer. The additional operation is when the bottom group split. However it only affects at most two leaf nodes at the top layer. We can split the leaf node if it crosses more than two sets.

Theorem 57 *To insert a key in the dynamic atomic-key B-tree, the number of amortized memory transfers is*

$$O\left(\left(1+\frac{\hat{k}}{B}\right)\log_{2+B/\hat{k}} N\right).$$

Proof. For each tree node of N keys and average length $\hat{k}$, we can insert $\Theta(N/f)$ keys before rebuilding the tree node. The cost to rebuild the tree is

$$O(N/f + N\hat{k}/B).$$

Thus, the amortized cost to insert a key into a tree node is

$$O\left(\frac{N/f + N\hat{k}/B}{N/f}\right) = O\left(1+\frac{f\hat{k}}{B}\right).$$

Noticing that $f = \max\{2, B/\hat{k}\}$, we have $O(f\hat{k}/B) = O(1+\hat{k}/B)$. Thus, the amortized cost to insert a key in to a tree node is $O(1+\hat{k}/B)$. Because there are roughly $O(\log_{2+B/\hat{k}} N)$ levels in the tree layout, we have the amortized cost to insert a key in the tree layout at most

$$O\left(\left(1+\frac{\hat{k}}{B}\right)\log_{2+B/\hat{k}} N\right).$$

□

5.4 Optimal Static Structure by Dynamic Programming

In this section, we give the optimal static search structure for N atomic keys, each of which has different size.

We need some notation. Assume that the set of keys is $\{\kappa_i\}_{1\leq i\leq N}$ and the length of the key κ_i is $\mathrm{Size}(\kappa_i)$. Each key has the probability p_i to be searched. Let $K_{i,j}$ be the total length of keys $\{\kappa_i, \kappa_{i+1}, \ldots, \kappa_j\}$ and $P_{i,j} = \sum_{i\leq r\leq j} p_r$. We define $T(i, j, b)$ the optimal static search tree for keys $\{\kappa_i, \kappa_{i+1}, \ldots, \kappa_j\}$ with the root node of size up to b, and $c(i, j, b)$ the average searching cost of keys $\{\kappa_i, \kappa_{i+1}, \ldots, \kappa_j\}$.

Our goal is to find the optimal tree $T(1,N,B)$ such that the search cost $c(1,N,B)$ is minimized. Notice that the tree for a larger set of keys are constructed by joining trees for smaller sets. An optimal search tree for the set $\kappa_i, \ldots, \kappa_j$ whose root occupies space less than S is constructed by joining optimal trees for the set $\kappa_{r+1}, \ldots, \kappa_j$ with space less than $S - \mathrm{Size}(\kappa_r)$ in the root and an optimal tree for the set $\kappa_i, \ldots, \kappa_{r-1}$ (See Figure 42).

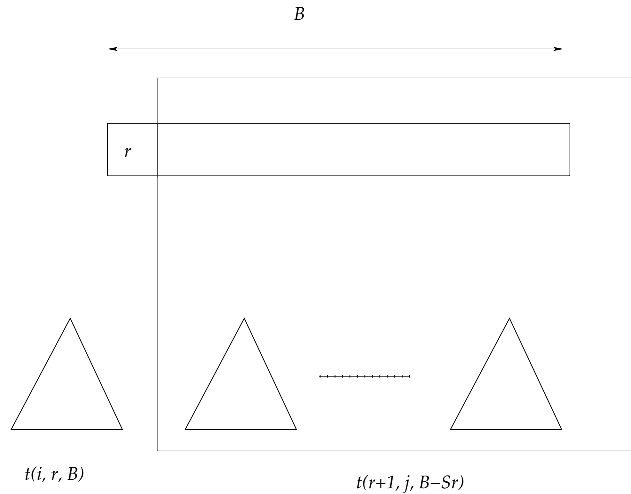

Figure 42: The optimal structure by the dynamic programming.

Therefore, by dynamic programming, we can construct the optimal search tree as follows:

$$c(i, j, b) = \begin{cases} P_{i,j} & \text{if } K_{i,j} \leq b; \\ \min\limits_{i\leq r\leq j} \{c(i, r, B) + P_{i,r} + c(r+1, j, b - \mathrm{Size}(\kappa_r))\} & \text{otherwise.} \end{cases}$$

The cost of this dynamic programming is $O(Bn^3)$.

In [11], a faster dynamic-programming scheme is proposed. This scheme could also apply here with a cost of $\mathrm{O}(Bn^{\alpha})$, with $\alpha = 2 + \log 2/\log(B+1)$.

5.5 Conclusion

We construct both static and dynamic atomic-key B-trees in linear time. The dynamic atomic-key B-tree supports the operations search, insert and delete in

$$O\left(\lceil \hat{k}/B \rceil \log_{1+\lceil B/\hat{k} \rceil} N\right)$$

amortized memory transfers. The update cost matches that of the traditional B-tree when keys have the unit size. Therefore, our atomic-key B-tree is a generalized version of the B-tree in the respect of key size. However, to construct the corresponding cache-oblivious version of the atomic-key B-tree remains open. It would be especially interesting to find the structure for a dynamic cache-oblivious atomic-key B-tree.

Bibliography

[1] P. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. On cache-oblivious multidimensional range searching. In *Proc. 19th ACM Symp. on Comp. Geom. (SOCG)*, pages 237–245, 2003.

[2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 305–314, 1987.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. 28th Ann. IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 204–216, 1987.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2–3):72–109, 1994.

[6] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In *Proc. 37th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 580–589, 1996.

[7] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. *Algorithmica*, 32(2):277–301, 2002.

[8] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 268–276, 2002.

[9] R. D. Barve and J. S. Vitter. A theoretical framework for memory-adaptive algorithms. In *Proc. 40th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 273–284, 1999.

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

[11] P. Becker. A new algorithm for the construction of optimal b-trees. *Nordic J. of Computing*, 1(4):389–401, 1994.

[12] 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 *Proc. 44th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 271–280, 2003.

[13] M. A. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th Ann. European Symp. on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 139–151, 2002.

[14] M. A. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *LNCS*, pages 195–207, 2002.

[15] M. A. Bender, E. Demaine, and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *Proc. 10th Ann. European Symp. on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 165–173, 2002.

[16] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

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

[18] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. of the 13th Annual Symposium on Discrete Mathematics (SODA)*, pages 29–38, 2002.

[19] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

[20] M. A. Bender and M. Farach-Colton. The LCA problem revisited, 2000.

[21] M. A. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string B-trees. In *Proc. 25th Symp. on Principles of Database Systems (PODS)*, pages 233–242, 2006.

[22] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In S. Vansummeren, editor, *PODS*, pages 233–242. ACM, 2006.

[23] M. A. Bender, M. Farach-Colton, and M. Mosteiro. Insertion sort is $O(n \log n)$. In *Proc. 3rd International Conference on Fun with Algorithms (FUN)*, pages 16–23, 2004.

[24] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. 17th Ann. Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, 2005.

[25] M. A. Bender and H. Hu. An adaptive packed-memory array. In *Proc. 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 20–29, 2006.

[26] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. 8th Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, 1996.

[27] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *LNCS*, pages 426–438, 2002.

[28] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th Ann. International Symp. on Algorithms and Computation (ISAAC)*, volume 2518 of *LNCS*, pages 219–228, 2002.

[29] G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 581–590, 2006.

[30] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[31] Cole and Hariharan. Dynamic LCA queries on trees. *SICOMP: SIAM Journal on Computing*, 34, 2005.

[32] Comer, D. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[33] E. D. Demaine. Cache-oblivious algorithms and data structures. Lecture Notes from the EEF Summer School on Massive Data Sets, 2002.

[34] P. F. Dietz. Maintaining order in a linked list. In *STOC*, pages 122–127. ACM, 1982.

[35] P. F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Proc. 4th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142, 1994.

[36] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Ann. Symp. on Theory of Computing (STOC)*, pages 365–372, 1987.

[37] P. F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 447 of *Lecture Notes in Computer Science*, 1990.

[38] Ferragina and Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *JACM: Journal of the ACM*, 46, 1999.

[39] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

[40] Gabow, Bentley, and Tarjan. Scaling and related techniques for geometry problems. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1984.

[41] Harel and Tarjan. Fast algorithms for finding nearest common ancestors. *SICOMP: SIAM Journal on Computing*, 13, 1984.

[42] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. 13th Ann. ACM Symp. on Theory of Computation (STOC)*, pages 326–333, 1981.

[43] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Internationl Colloquium on Automata, Languages, and Programming (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, 1981.

[44] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Israel Inst. of Tech., Haifa, May 2002.

[45] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.

[46] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison–Wesley, 3rd edition, 1997.

[47] P. Kumar. Cache oblivious algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies, LNCS 2625*, pages 193–212, 2003.

[48] P. Kumar and E. Ramos. I/O efficient construction of Voronoi diagrams. Unpublished manuscript, July 2002.

[49] M. Li and P. Vitanyi. *An introduction to Kolmogorov complexity and its applications*. Springer, 2nd edition, 1997.

[50] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[51] N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. 5th Int. Workshop on Algorithm Engineering (WAE)*, volume 2141, pages 67–78, 2001.

[52] V. Raman. Locality-preserving dictionaries: theory and application to clustering in databases. In *Proc. 18th Symp. on Principles of Database Systems (PODS)*, pages 337–345, 1999.

[53] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[54] J. E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proc. 1st Ann. International Conference on Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, pages 270–281, 1995.

[55] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proc. 11th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 829–838, 2000.

[56] S. Software. The berkeley database.

[57] S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[58] A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984.

[59] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. 16th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

[60] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[61] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2), 2001.

[62] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[63] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

[64] D. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proc. 14th Ann. Symp. on Theory of Computing (STOC)*, pages 114–121, 1982.

[65] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 251–260, 1986.

[66] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.