External-Memory Search Trees with Fast Insertions

by

Jelani Nelson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author . . . .
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
Bradley C. Kuszmaul
Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

External-Memory Search Trees with Fast Insertions

by

Jelani Nelson

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis provides both experimental and theoretical contributions regarding external-memory dynamic search trees with fast insertions. The first contribution is the implementation of the buffered repository B$^\epsilon$-tree, a data structure that provably outperforms B-trees for updates at the cost of a constant factor decrease in query performance. This thesis also describes the cache-oblivious lookahead array, which outperforms B-trees for updates at a logarithmic cost in query performance, and does so without knowing the cache parameters of the system it is being run on.

The buffered repository B$^\epsilon$-tree is an external-memory search tree that can be tuned for a tradeoff between queries and updates. Specifically, for any $\epsilon \in [1/\lg B, 1]$ this data structure achieves $O((1/\epsilon B^{1-\epsilon})(1+\log_B(N/B)))$ block transfers for Insert and Delete and $O((1/\epsilon)(1+\log_B(N/B)))$ block transfers for Search. The update complexity is amortized and is $O((1/\epsilon)(1+\log_B(N/B)))$ in the worst case. Using the value $\epsilon = 1/2$, I was able to achieve a 17 times increase in insertion performance at the cost of only a 3 times decrease in search performance on a database with 12-byte items on a disk with a 4-kilobyte block size.

This thesis also shows how to build a cache-oblivious data structure, the cache-oblivious lookahead array, which achieves the same bounds as the buffered repository B$^\epsilon$-tree in the case where $\epsilon = 1/\lg B$. Specifically, it achieves an update complexity of $O((1/B)\log(N/B))$ and a query complexity of $O(\log(N/B))$ block transfers. This is the first data structure to achieve these bounds cache-obliviously.

The research involving the cache-oblivious lookahead array represents joint work with Michael A. Bender, Jeremy Fineman, and Bradley C. Kuszmaul.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Bradley C. Kuszmaul
Title: Research Scientist

Acknowledgments

I would like to thank my advisors, Bradley C. Kuszmaul and Charles E. Leiserson, for their guidance in my research. Both Bradley and Charles have made themselves available over these past two semesters whenever I wanted to talk. I was in fact led to my thesis topic during a meeting with both of them. Bradley also provided me with his B$^+$-tree implementation to benchmark the BRB$^\epsilon$T against.

I also thank those who let me describe the technical details of my research to them on numerous occasions. When I do not communicate my thoughts, they often become jumbled in my head and never clearly develop. Having people who would sit and listen as I explained my ideas to them was thus essential in the development of this work. For this I thank Kunal Agrawal, Alexandr Andoni, Arnab Bhattacharyya, Harr Chen, Jeremy Fineman, David Shin, Jim Sukha, and Vincent Yeung. There is a long list of names missing from this list of those with whom I only spoke once briefly about my work, and I would like to thank those people as well.

Michael D. Ernst was the first person to introduce me to computer science research, and I thank him for this. I thank Erik D. Demaine for showing me the fun in theoretical computer science research during my senior undergraduate year. The open problem solving sessions during his advanced data structures class brought a social aspect to theory research that made it fun.

I thank Albert R. Meyer and Ronitt Rubinfeld for excellent professor-TA relationships last semester when I was a teaching assistant for 6.042. Both professors were easy to get along with, and the demands placed on me as a TA were reasonable. I also thank my fellow TAs and my students for an overall good time.

Finally, I would like to thank several people for support that was not research-related. I thank the many TOC students and faculty who met with me frequently as I was deciding what to do with the next few years of my life. Madhu Sudan especially met with me many times, often when I was unannounced and without appointment. I would like to thank all my friends at MIT and back home in St. Thomas (and scattered elsewhere) for all the love they have shown me in the time I have known them. I especially would like to thank my family for being caring and loving.

Contents

Chapter 1

Introduction

In computers today, access times for cache and memory typically differ by about 3 orders of magnitude, and there is again a difference of 5 orders of magnitude when comparing memory with disk. Furthermore, memory hierarchies in modern machines often have four or five levels: the L1, L2, and sometimes L3 caches, physical memory, and disk. Thus, it is important to understand how we can develop algorithms and data structures that effectively use the fastest levels of the memory hierarchy.

This thesis investigates the problem of dynamic search trees that perform updates quickly and are efficient when the number of data items is large. An old solution to this problem, the B-tree [4], is ubiquitous in modern databases and file systems [13]. The B-tree, however, only optimizes across two adjacent levels of the memory hierarchy, typically memory and disk. Furthermore, the B-tree is most effective in the case where the number of queries far exceeds the number of updates.

In this thesis we discuss two data structures — the buffered repository B$^\epsilon$-tree (BRB$^\epsilon$T) and the cache-oblivious lookahead array (COLA) — both of which provably outperform the B-tree in the case where the number of updates dominates the number of queries. The COLA has the added advantage of effectively utilizing all levels of the memory hierarchy without knowing anything about the parameters of cache and memory. The BRB$^\epsilon$T was initially proposed in [10], and this thesis provides an experimental evaluation of its performance. The COLA was designed jointly with Michael A. Bender, Jeremy Fineman, and Bradley C. Kuszmaul, and has yet to be

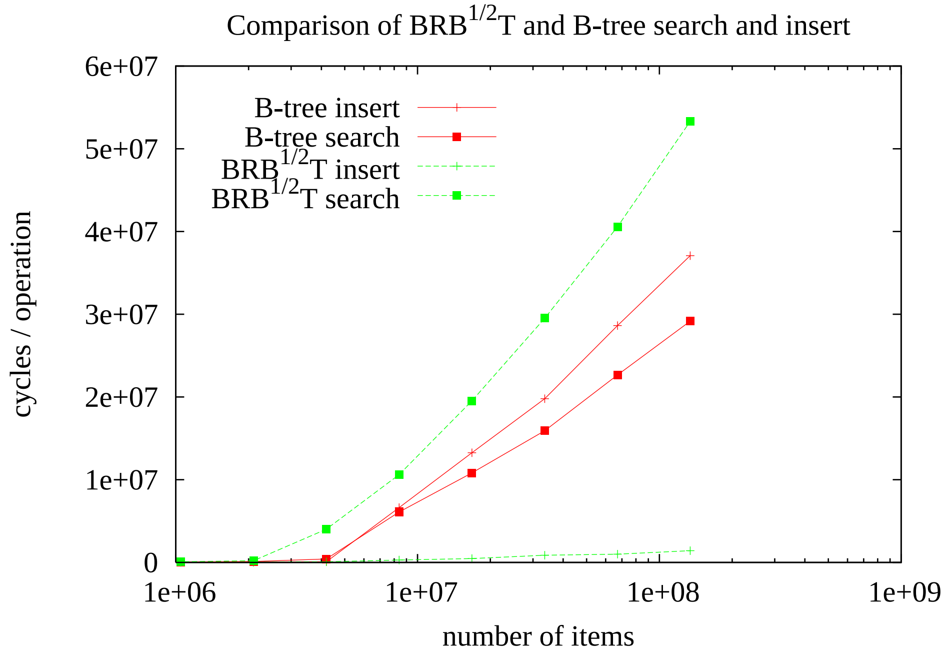


Figure 1-1: Performance comparison between the BRB$^{1/2}$T and the B-tree. These results are from a system with 128 megabytes of RAM and a 4 kilobyte block size on disk. Each (index,value) pair was 12 bytes in size.

implemented.

The first data structure, the BRB$^{\epsilon}$T, outperforms B-trees for updates at the expense of a constant factor decrease in query performance. This data structure was originally proposed in [10]. In addition to describing the theoretical ideas behind the data structure, I also provide an implementation and experimental analysis. In the largest test case, where the size of the database was over 15 times the amount of physical memory on the machine, I was able to achieve roughly a 17 times performance increase of BRB$^{\epsilon}$T insertion over B-tree insertion, while Search was slower by roughly a 3 times factor. Figure 1-1 shows experimental results comparing the two data structures.

The cache-oblivious lookahead array (COLA) makes a larger sacrifice than the BRB$^{\epsilon}$T in terms of query performance, but it is much more efficient for updates. Furthermore, it uses the entire memory hierarchy effectively without knowing the number of levels in the hierarchy or the parameters of the storage available at any

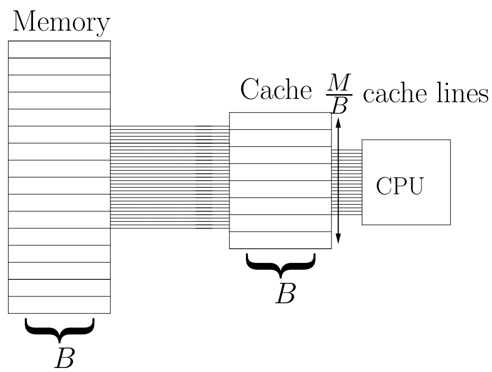


Figure 1-2: Two adjacent levels in external memory. The value B is the block size of memory in bytes and M is the size of cache.

level. The cache-oblivious lookahead array was originally proposed by Michael A. Bender and Bradley C. Kuszmaul [9], but was heavily amortized. In this thesis I show how to partially deamortize the COLA when $M = \Omega(\log^2 N)$ and $M/B = \Omega(\log N)$. This deamortization represents joint work with with Jeremy Fineman.

1.1 The External-Memory Model

The external-memory model was first introduced in [20]. This model measures an algorithm by its I/O complexity: the number of block transfers that occur between two adjacent levels of the memory hierarchy while the algorithm runs. The model is especially useful when an algorithm must perform disk I/O during execution, as the difference between disk access and memory access speeds is over 5 orders of magnitude.

We now discuss the formal details of the external-memory model. Figure 1-2 illustrates the model visually. In this model a computer contains a two-level memory hierarchy consisting of a *memory* and a *cache*. The memory is of unbounded size and is divided up into *blocks* of B bytes each. The cache is of size M and is divided into M/B *cache lines*, each of also size B. The cache is fully-associative [19, Ch. 5]; that is, any block in memory may be brought into any cache line.

Whenever the processor reads a value that resides in cache, it is read without cost. Otherwise, the algorithm pays one block transfer to bring that value, along with all other values residing in its block, from memory to cache. Should there be no free cache lines, one cache line's contents are evicted to memory using some cache replacement strategy, which we assume is offline optimal [5]. This assumption of an offline optimal cache replacement strategy was introduced in [17] and is known as the *ideal cache model*. Frigo *et al.* showed that the ideal cache model is well-justified by using a result [28] due to Sleator and Tarjan showing that the popular *least recently used* [19, p. 378] replacement strategy is 2-competitive with the offline optimal strategy on a cache of half the size. When we analyze an algorithm using the external-memory model, we analyze its I/O complexity: the number of block transfers that occur from memory to cache throughout the execution of the algorithm. Thus, this model provides a useful means of predicting real performance when the input to the algorithm is too large to fit in cache and I/O becomes a bottleneck.

A *cache-oblivious* algorithm runs without using knowledge of the parameters M and B [17, 25]. Algorithms that do use knowledge of these parameters are *cache-aware*. When analyzing a cache-oblivious algorithm, we always use the ideal cache model and assume that the system uses an offline optimal cache replacement strategy.

The notion of cache-obliviousness was first introduced in [17, 25], where the authors provided efficient cache-oblivious solutions to several problems including sorting, matrix transposition, matrix multiplication, and performing discrete Fourier transforms. Experimental results [23, 27] indicate that generally cache-oblivious algorithms perform better than cache-aware ones when the data set is too large to fit in main memory, but are slower when only the higher levels of the memory hierarchy are utilized.

1.2 Preliminaries

This section introduces a formalization of the dynamic search tree problem, in which we wish to *dynamically* maintain a multiset of items that are $(index, value)$ pairs subject to the insertion and deletion of items. This thesis often refers to this multiset of items as the *database*. Indices are unique and come from a totally ordered universe;

that is, for two indices k_1, k_2, we are allowed to perform the tests $k_1 < k_2$, $k_1 > k_2$, and $k_1 = k_2$, and exactly one holds. This section also states some assumptions and notational conventions used throughout the remainder of this thesis.

The operations a dynamic search tree should support are split into two categories: *queries* and *updates*. The operations INSERT and DELETE will be collectively referred to as updates, and all other operations we discuss are queries, since they do not alter the database. The data structures we consider support the following operations:

- INSERT (k, v): If an item with key k exists in the database, change its value to v. Otherwise, insert a new item into the database with key k and value v.

- DELETE (k): If an item with key k exists in the database, delete it. Otherwise, do nothing.

- PREDECESSOR (k): Return the index and associated value of the item in our database with the highest index strictly less than k, or report if no such item exists.

- SEARCH (k): Return the value of the item in our database with index k, or report if no such item exists.

- RANGE-SEARCH $(k_1,\ k_2)$: Return the indices and values of all items in our database with indices in the range $[k_1, k_2]$.

This thesis assumes certain properties of the parameters of the memory hierarchy. Some variables are also consistently used throughout this thesis. The variable B is always used to refer to the block size on memory, M refers to the size of cache, and N denotes the number distinct indices of items appearing in the database. We assume that $M = \Omega(\log N)$ so that at least a constant number of pointers can fit in cache. We also use $\log_c x$ to represent $\max\{0, \log_c x\}$ and assume that $N = \Omega(B)$ for simplicity in stating bounds.

	B-tree	BRB$^\epsilon$T	COLA
Queries	$O(\log_B N)$	$O((1/\epsilon)(1+\log_B(N/B)))$	$O(\log(N/B))$
Updates	$O(\log_B N)$	$O((1/\epsilon B^{1-\epsilon})(1+\log_B(N/B)))$	$O((1/B)\log(N/B))$
Introduced by	Bayer & McCreight, '72 [4]	Brodal & Fagerberg, '03 [10]	**this thesis**

Figure 1-3: I/O bounds for external-memory data structures. The value B is the block size in memory, N is the number of items in the database, and $\epsilon \in [1/\lg B, 1]$ is a tunable parameter of the BRB$^\epsilon$T.

1.3 Contributions

This thesis provides both theoretical and experimental contributions in the area of external-memory dynamic search trees. I have implemented and experimentally evaluated the buffered repository B$^\epsilon$-tree [10], which provably asymptotically outperforms the B-tree for updates at the cost of a constant factor decrease in query performance. I have also jointly developed a new data structure, the cache-oblivious lookahead array, which also provably outperforms the B-tree for updates but at a cost in query performance that is logarithmic in the block size B. The lookahead array was developed jointly with Michael A. Bender, Jeremy Fineman, and Bradley C. Kuszmaul. For each data structure we discuss, the bounds for Insert and Delete are the same, and the bounds for Search and Predecessor are the same. Henceforth, when comparing bounds we will refer to bounds as being for queries (Predecessor and Search) or for updates (Insert and Delete).

Although the B-tree provides provably optimal I/O complexity for Search [2] (and even without assuming the comparison model [16]), its asymptotic complexity for updates can be improved at the cost of only a constant factor in search performance for queries using the buffered repository B$^\epsilon$-tree (BRB$^\epsilon$T) [10]. In my experiments, I was able to witness a 17 times increase in insertion performance at the cost of a 3 times slowdown in search performance when comparing the BRB$^\epsilon$T with a B$^+$-tree implementation provided by Bradley C. Kuszmaul. The B-tree achieves $O(\log_B N)$ worst-case block transfers for both queries and updates. The B$^\epsilon$-tree achieves $O((1/\epsilon)(1 + \log_B(N/B)))$ I/O complexity for these same operations in the worst case, but the amortized complexity of updates reduces to $O((1/\epsilon B^{1-\epsilon})(1 + \log_B(N/B)))$. Let ϵ

be a parameter to be set during implementation that can take values in the range $[1/\lg B, 1]$. Note that for any fixed constant ϵ, the worst-case performance of all operations is within a constant factor of that of a B-tree, while the amortized number of block transfers for updates is asymptotically better. Chapter 5 discusses in detail how the theoretical analysis maps to real-world performance benefits.

While the design of the B$^{\epsilon}$-tree is somewhat similar to that of a B-tree, the lookahead array is much different. Unlike the B-tree and B$^{\epsilon}$-tree, the COLA does not actually have a tree-like structure. The specific details are covered in Chapter 6. The COLA achieves the same bounds as the B$^{\epsilon}$-tree for the setting $\epsilon = 1/\lg B$. That is, all operations incur $O(\log(N/B))$ block transfers in the worst case, while updates incur an amortized $O((1/B)\log(N/B))$ block transfers.

Though the COLA has yet to be implemented, various schemes for cache-oblivious B-trees have been proposed [6, 7, 8, 11, 26], and some experimental evidence indicates that they often perform as well as or better than their cache-aware counterpart when data does not fit into main memory [8, 22, 26]. This research gives us hope that our cache-oblivious lookahead array may in fact perform well in practice.

1.4 Outline of Thesis

What follows is an outline of the remainder of this thesis. Chapter 2 introduces the buffered repository B$^{\epsilon}$-tree and describes the invariants the data structure maintains. Chapters 3 and 4 describe how to perform queries and updates, respectively, on the BRB$^{\epsilon}$T. Chapter 5 provides implementation details and an experimental analysis of the BRB$^{\epsilon}$T. Chapter 6 introduces the cache-oblivious lookahead array and discusses how to perform operations on the data structure. Chapter 7 concludes this thesis and discusses future research directions.

Chapter 2

The Buffered Repository B$^\epsilon$-tree

The buffered repository tree (BRT) is a data structure that was introduced in [12] as a tool to obtain an efficient depth-first search algorithm in the external-memory model. The structure is similar to a B-tree, but uses buffering to a achieve better performance for inserts at the cost of search performance. In this chapter, I discuss the details of the (a, b)-BRT, a generalization of the BRT. The buffered repository B$^\epsilon$-tree, or BRB$^\epsilon$T, is a special case of the (a, b)-BRT when $a = \Theta(B^\epsilon)$ and $b = \Theta(B^\epsilon)$. Brodal and Fagerberg first proposed the BRB$^\epsilon$T without naming it in [10]. The value ϵ is a parameter that can be tuned for a tradeoff in asymptotic I/O complexity between inserts and searches.

For any constant value of the tuning parameter ϵ, the BRB$^\epsilon$T achieves an asymptotically equivalent number of block transfers as the B-tree for queries while strictly outperforming it for inserts. Specifically, for any value $\epsilon \in [1/\lg B, 1]$, the BRB$^\epsilon$T incurs $O((1/\epsilon B^{1-\epsilon})(1+\log_B(N/B)))$ block transfers for updates, $O((1/\epsilon)(1+\log_B(N/B)))$ block transfers for Search and Predecessor, and $O((1/\epsilon)(1+\log_B(N/B)+S'/B))$ block transfers for Range-Search, where S' is the number of items in the query range that were in the database within the last $N/2$ updates. We also show how to improve Range-Search to $O((1/\epsilon)(1 + \log_B(N/B) + S/B))$ block transfers, where S is the number of items in the query range currently contained in the BRB$^\epsilon$T, at the cost of making Delete as slow as Search.

An (a, b)-BRT is a search tree where each node consists of two pieces: its keys and

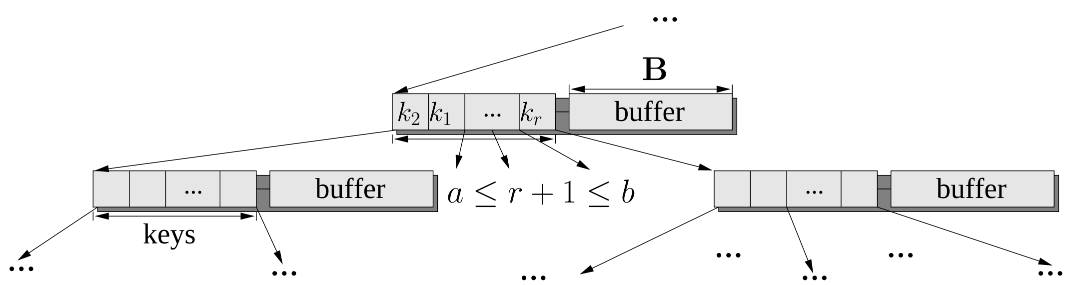


Figure 2-1: Three nodes in an (a, b)-BRT. The values $k_1, k_2, \ldots, k_r$ are the keys of the shown node.

its buffer. The buffer is an array of size B, the block size in memory. I describe an implementation where elements can be kept in any node's keys or buffers. It is also possible to keep items only in buffers and still achieve the same bounds.

Th (a, b)-BRT data structure maintains that the tree taken by ignoring all buffers is a valid (a, b)-tree, except for leaves which contain no keys and consist solely of a buffer. A BRT is a special case of the (a, b)-BRT where $a = 2$ and $b = O(1)$. An (a, b)-tree is a rooted tree satisfying the following properties, where $a, b > 1$:

1. All leaves are at equal distance from the root.

2. Every node x stores r keys $k_1[x], k_2[x], \ldots, k_r[x]$, where $1 < r + 1 \leq b$ and $index[k_i[x]] \leq index[k_j[x]]$ whenever $i < j$. If x is not the root of the tree, then the inequality $a \leq r + 1$ also holds.

3. Each internal node x stores $r + 1$ pointers $c_1[x], c_2[x], \ldots, c_{r+1}[x]$ to other nodes that we refer to as $x's$ *children*. If k_i is stored in the tree rooted at $c_i[x]$, then $index[k_1] \leq index[k_1[x]] \leq index[k_2] \leq index[k_2[x]] \cdots \leq index[k_r[x]] \leq index[k_{r+1}]$.

A more detailed treatment of (a, b)-trees can be found in [14]. I will assume throughout all analyses that $a \leq B$ and $b \leq B$ so that we can read a node from memory in at most two block transfers. For proper operation of the (a, b)-BRT, we also need that $a + 1 \leq \lfloor(\lfloor b/2\rfloor + 1)/2\rfloor$. The analysis of the EAGER-DELETE operation in Section 4.2 explains why this inequality must hold.

An (a, b)-BRT always maintains the following list of invariants:

BRT1 Every internal node is divided into a set of keys, corresponding to the keys of the (a, b)-tree, and a buffer containing at most B items. Leaves contain no keys and consist only of a buffer containing between $\lceil B/2 \rceil - 1$ and B elements, except perhaps in the special case where $N < \lceil B/2 \rceil - 1$ and the root node is itself a leaf.

BRT2 The root node is always kept in cache.

BRT3 Items in buffers obey the search tree criteria. That is, for any non root node x, if x is the ith child of its parent, then all items in x's buffer have indices with ranks between the $(i-1)$st and ith keys' indices.

BRT4 The tree taken by ignoring all nodes' buffers is a valid (a, b)-tree.

BRT5 Suppose there are two insertions into the (a, b)-BRT of items with the same index k. Then, we have the following.

(a) Both items appear in the same root-to-leaf path.

(b) The update that is more recent resides in a node that is strictly closer to the root.

(c) At least one of the two items belongs to a buffer.

In chapters 3 and 4, we show how to perform the operations SEARCH, PREDECESSOR, RANGE-SEARCH, INSERT, and DELETE on an (a, b)-BRT. Specifically, we show that SEARCH and PREDECESSOR can be implemented in $O(1+\log_a(N/B))$ block transfers, INSERT and DELETE can be implemented in $O((b/B)(1+\log_a(N/B)))$ block transfers, and RANGE-SEARCH can be implemented in $O((1/B)(1 + \log_a(N/B) + S'/B))$ block transfers, where S' is the number of items in the query range that were in the database in the last $N/2$ updates. We also show how to improve the RANGE-SEARCH bound to $O(1+\log_a(N/B)+S/B)$ block transfers, where S is the number of items in the query range that are currently in the database, at the cost of increasing the I/O complexity of DELETE to $O(1 + \log_a(N/B))$.

The (a, b)-BRT bounds stated in the previous paragraph can be better understood by a change of notation. We set $b = B^\epsilon$ and consider (a, b)-BRTs where $a = \Theta(b)$. We now rewrite the bounds in terms of the parameter ϵ. Since $2 \leq a \leq B$, we have that $\epsilon \in [1/\lg B, 1]$. The bounds can then be rewritten as $O((1/\epsilon)(1 + \log_B(N/B)))$ block transfers for Search and Predecessor, $O((1/\epsilon B^{1-\epsilon})(1 + \log_B(N/B)))$ block transfers for Insert and Delete, and $O((1/\epsilon)(1+\log_B(N/B)+S'/B))$ block transfers for Range-Search.

Chapter 3

Query Operations on an (a, b)-BRT

This chapter discusses how to implement three query operations on the (a, b)-BRT: Search, Predecessor, and Range-Search. Section 3.1 investigates an implementation of Search achieving $O(1 + \log_a(N/B))$ block transfers. Section 3.2 shows how to implement Predecessor to also achieve $O(1 + \log_a(N/B))$ block transfers. Section 3.3 shows how to implement Range-Search to achieve $O(1 + \log_a(N/B) + S/B)$ block transfers.

3.1 Search

The procedure Search works much as in an (a, b)-tree, but we must also make sure to look through a node's buffer. Figure 3-1 shows pseudocode for the Search procedure. Since the (a, b)-BRT maintains Invariant BRT5, we are guaranteed that the first item we find with a given index is the most recent one.

Definition 1. *Let T be an (a, b)-BRT. Define the* ***depth*** *of a node $x \in T$ by*

$$d(x) = \begin{cases} 0 & \textit{if } x \textit{ is the root of } T, \\ 1 + d(parent[x]), & \textit{otherwise.} \end{cases}$$

The ***height*** *of T is $h(T) = \max_{x \in T} d(x)$.*

```
Search(T, index)
x ← root[T]
while TRUE
      do for i ← 1 to bufferSize[x]
              do if index[x.buffer[i]] = index
                    then return value[x.buffer[i]]
         if isLeaf[x]
            then return NIL
         child ← 1
         for i ← 1 to numkeys[x]
              do if index[x.keys[child]] = index
                    then return value[x.keys[child]]
                 elseif index[x.keys[child]] < index
                    then child ← i + 1
                    else break
         x ← x.children[child]
```

Figure 3-1: A procedure for searching in (a, b)-BRT. The input parameter T is a pointer to the (a, b)-BRT we are searching, and *index* is the index of the item we would like to search for.

Lemma 2. *Let T be an (a, b)-BRT and N be the number of items in T. Then T contains $\Theta(N/B)$ nodes, and $h(T) \leq 1 + \log_a\lceil N/(\lceil B/2\rceil - 1)\rceil$.*

Proof. In the case that T consists of only one node, the lemma holds because $h(T) = 0 < 1 \leq 1 + \log_a\lceil N/(\lceil B/2\rceil - 1)\rceil$. Otherwise, every leaf contains at least $\lceil B/2\rceil - 1$ elements, and thus there can be at most $\lceil N/(\lceil B/2\rceil - 1)\rceil$ leaves. In any rooted tree where each internal node has at least $a > 1$ children, the number of nodes in the whole tree is less than twice the number of leaves, and so the number of nodes of T is $O(N/B)$. Since each node holds at most $B + b - 1 = O(B)$ items, there are also at least $\lceil N/(B + b - 1)\rceil = \Omega(N/B)$ nodes in T. Thus, there are $\Theta(N/B)$ nodes, showing the first part of the lemma.

A straightforward induction shows that an (a, b)-tree of height h contains at least $2a^{h-1}$ leaves. The root of an (a, b)-tree is only restricted to have degree at least 2 but might have degree smaller than a. Since $a \geq 2$ for an (a, b)-BRT, this provides us with our desired bound for the second part of the lemma. $\square$

Theorem 3. SEARCH *incurs at most* $O(1 + \log_a(N/B))$ *block transfers.*

Proof. The height bound given in Lemma 2 is $O(1+\log_a(N/B))$. We now prove that search on an (a, b)-BRT incurs at most $O(h)$ block transfers. Pseudocode for SEARCH is shown in Figure 3-1. The variable x is initialized as the root, and since each time through the `while` loop we follow a child pointer, the loop executes exactly h times. The only item we touch while in the loop is the current node, and since each node occupies at most 2 blocks of space (a size B buffer plus an additional $O(b)$ space for keys and children pointers), SEARCH incurs at most $O(h)$ block transfers. □

3.2 Predecessor

We now discuss how to implement the PREDECESSOR function for an (a, b)-BRT to incur $O(1 + \log_a(N/B))$ block transfers. The PREDECESSOR function takes as input an index k and returns the item in our database with the highest index less than k, or NIL if no such element exists.

Theorem 4. PREDECESSOR *can be implemented to incur at most* $O(1+\log_a(N/B))$ *block transfers.*

Proof. Given an index k, we first search for a node's key or a leaf buffer that contains an item with index k. If no such item with index k exists in the tree, we find the leaf x where such an item would reside if inserted. If the node x had k in its key, we follow the child pointer to that key's left and then follow the rightmost child pointers all the way down to a leaf of the tree, taking the maximum index item from all the buffers of nodes we visit while traversing downward. We then take the maximum index item between the largest index buffer item we find and the largest buffer item in the leaf we arrive at. If the node x is a leaf, we return the maximum index item over all key and buffer items on a root-to-leaf traversal from the root to x. As both these cases do not require us to visit more nodes than the height of the (a, b)-BRT, PREDECESSOR incurs $O(1 + \log_a(N/B))$ block transfers. □

3.3 Range-Search

We now discuss how to implement Range-Search, which when given two indices k and k', reports all items with indices in the range $[k, k']$. The procedure Range-Search can be implemented to incur $O(1 + \log_a(N/B) + S/B)$ block transfers when $M/B = \Omega(\log N)$, where S is the number of items to be output in the database. To prove this result, we must first must prove a lemma.

Lemma 5. *Let x and y be two leaves of an (a, b)-BRT, where items in node x have indices smaller than those in y. Consider the root-to-leaf path to x and the root-to-leaf path in y. These two paths partition the tree into three regions: the region to the left of the path to x, the region to the right of the path to y, and the region in between. If there are m leaves in the region between the two paths, then there are $O(1 + \log_a(N/B) + m)$ nodes in this region total.*

Proof. If x and y are the same node, the middle region is a path down the tree and thus contains $O(1 + \log_a(N/B))$ nodes and and we are done. Otherwise, consider the least common ancestor z of x and y. In the path downward from z to x, whenever we follow the child pointer to the left of a key k, the subtree rooted to the right of k is itself an (a, b)-tree. Similarly, in the path toward y from the z, whenever we follow the child pointer to the right of a key, the subtree rooted to the left of that key is also an (a, b)-tree. Thus, by applying Lemma 2 and also considering the $O(1+\log_a(N/B))$ nodes visited on the two root-to-leaf paths, we prove the lemma. □

Theorem 6. Range-Search *can be implemented to incur at most $O(1+\log_a(N/B)+S/B)$ block transfers when $M/B = \Omega(\log N)$.*

Proof. Given indices k and k', we first search down the tree for the leaf x that would contain an item with index k if k were in the tree. We then walk along the leaves until we reach the node y that would contain an item with index k'. For each leaf ℓ we visit, we flush all buffer items of ℓ's ancestors with indices in the range for ℓ completely down the tree. We then output all items remaining in ℓ. This procedure ensures that all deletions and value updates take place before we output our answer.

We now finish the analysis of Range-Search using the fact that $M/B = \Omega(\log N)$. Since $M/B = \Omega(\log N)$, we can hold all the ancestors for a given leaf in cache simultaneously. Over the course of visiting leaves, since we visit leaves left to right, each internal node enters our list of ancestors at most once and stops being an ancestor at most once. Thus, we only need pay one block transfer per node as we visit leaves in the desired range. If there are S items to be reported, the number of leaves between x and and the last leaf we visit must be $O(1 + S/B)$. The number of nodes potentially containing items we must report is thus $O(1 + \log_a(N/B) + S/B)$ by Lemma 5. Range-Search thus incurs at most $O(1 + \log_a(N/B) + S/B)$ block transfers. $\square$

Chapter 4

Update Operations on an (a, b)-BRT

In this chapter we see how to implement INSERT and DELETE for the (a, b)-BRT and investigate analyses of the I/O complexities of these implementations. In particular, Section 4.1 shows how to implement INSERT in $O((b/B)(1+\log_a(N/B)))$ block transfers, and Section 4.2 shows how to implement DELETE in $O(1 + \log_a(N/B))$ block transfers. Section 4.3 then discusses how to improve the performance of DELETE to $O((b/B)(1 + \log_a(N/B)))$ block transfers using an algorithm called LAZY-DELETE. The implementation of LAZY-DELETE causes the performance of RANGE-SEARCH to decrease to $O((1/B)(1 + \log_a(N/B) + S'/B)$, where S' is the number of items in the query range that have been in the database within the last $N/2$ updates. Section 4.4 discusses how to deamortize INSERT and the two DELETE implementations.

4.1 Insertion

This section describes the implementation of the INSERT procedure and provides an analysis of its I/O complexity. We discuss an implementation where INSERT incurs an amortized $O((b/B)(1 + \log_a(N/B)))$ block transfers and $O(N/B)$ block transfers in the worst case. In Section 4.4 we show how to deamortize the worst-case performance down to $O(1 + \log_a(N/B))$ block transfers.

The idea in implementing Insert is that we usually just append the item to the end of the root's buffer. Occasionally, however, the root's buffer is full. In this case, we flush the root's buffer items down to the appropriate children by using the operation FlushBuffer described in Figure 4-1. Before performing the new insertion this flushing procedure is recursive, since as we push items down to a child, its buffer may become full, and so we flush that child as well. When a leaf becomes full, we split it in half and promote the median value to be a key of the parent. This action may cause the parent to contain more than $b-1$ keys, the maximum number, and so this splitting procedure is recursive upward in the tree.

This insertion procedure is somewhat tricky to implement. The trouble is that in the middle of flushing part of a node x's buffer to an appropriate child, that child may split and cause x to gain one more key. The node x may not have room for another key, however, and may thus have to split even though we have not yet finished flushing its buffer. To avoid this problem, we maintain an invariant — the *small-split invariant* — which specifies that if FlushBuffer is called on a node x, then x must have strictly less than $\lfloor b/2 \rfloor + 1$ children. If we maintain the small-split invariant, we are guaranteed that a node never need split after we call FlushBuffer on it.

In order to maintain the small-split invariant, before recursively calling the flush procedure on a node that has more than $\lfloor b/2 \rfloor$ children, we first split it. This technique for simplifying the implementation requires that $a \leq \lfloor (\lfloor b/2 \rfloor + 1)/2 \rfloor$, since we must maintain that preemptively split nodes still have at least a children. As an example, for the BRT where $a = 2$, the maximum number b of children must be at least 6 (we will see for Delete why it in fact needs to also be at least 10). Since if $b \geq 6$, we only split nodes that have at least 4 children, a split never creates a node with fewer than 2 children. For a value $b < 6$, we have $\lfloor b/2 \rfloor + 1 \leq 3$, and splitting a node with 3 children creates a node that only has one child.

We must also take care to maintain Invariant BRT5. That is, (a) two insertions into the same index should appear in the same root-to-leaf path, (b) the more recent insertion should be at a strictly smaller depth in the tree, and (c) at most one item is in a node's key. Since these three requirements hold true vacuously when the tree

▷ Flush all items from x's buffer to the appropriate children. This operation may cause recursive flushes should children's buffers overflow when they are being flushed to.

```
FlushBuffer(x)
Sort(x.buffer)
j ← 1
bufferIdx ← 1
▷ Loop over children of x
for i ← 1 to numkeys[x] +1
        do y ← x.children[i]
           if not isLeaf[y]  and numkeys[y] +1 > ⌊b/2⌋
              then SplitNode(y)
           k ← j
           if i = numkeys[x]
              then j ← bufferSize[x]
              else  while j ≤ bufferSize[x]  and index[x.buffer[j]] ≤ index[x.keys[i]]
                           do j ← j + 1
           for l ← k to j − 1
                  do if bufferSize[y] = B
                        then if isLeaf[y]
                                then tmp ← []
                                     tmp[1 . . B] ← y.buffer[1 . . B]
                                     tmp[B + 1 . . B + j − l] ← x.buffer[l . . j − 1]
                                     SplitLeaf(y, tmp)
                                else  FlushBuffer(y)
                                      y.buffer[bufferSize[y] +1] ← x.buffer[l]
                                      bufferSize[y] ← bufferSize[y] +1
                        else  y.buffer[bufferSize[y] + 1] ← x.buffer[l]
                              bufferSize[y] ← bufferSize[y] +1
```

Figure 4-1: Pseudocode for the (a, b)-BRT flush procedure. The pseudocode for SplitNode and SplitLeaf are not provided, but their implementations are fairly straightforward. For the sake of brevity, we have omitted the pseudocode for maintaining Invariant BRT5.

is empty, we need only show how to maintain them after an insertion. We assume Invariant Brt5 holds before the insertion.

We now show why Invariants BRT5(a) and BRT5(b) are maintained after a call to Insert. After an insertion, the newly inserted item resides in the root's buffer and thus cannot participate in violating BRT5. When flushing the buffer of a node x into a node y, we make sure to delete all items in y that have the same indices as items we flush from x. If any key of y and buffer item of x both have the same index, we replace the value of that key in y with the more recent value. We also must take some care during the splitting of nodes. If during the split of a node x, some item z is moved to x's parent y, we check if any of y's buffer items have the same index as z. If so, we replace z's value with the value of that buffer item item then delete the buffer item. We thus ensure that BRT5(a) and BRT5(b) are maintained.

Since Invariant BRT5(c) holds before insertion, it could only possibly break due to a leaf splitting, since that is the only time new keys are created. No other key in the tree can have the same index as the node promoted during the split, since that would imply that the two items were at the same level at some point in the tree's history. Thus, the invariant is upheld during insertion.

Theorem 7. Insert *incurs an amortized* $O((b/B)(1 + \log_a(N/B)))$ *block transfers. The worst-case cost for any one call to* Insert *is* $O(N/B)$*.*

Proof. Since a call to Insert may potentially touch each node via recursive calls of FlushBuffer, the worst-case bound follows by Lemma 2. For the amortized bound, we give an item $\Theta((b/B)(1 + \log_a(N/B)))$ credits upon being inserted. We maintain the invariant that an item at level ℓ still has $\Theta(\ell/B)$ credits left, where the leaves are at level 0. Whenever we flush a buffer, we flush B items to at most b children. Thus, each item need only pay $\Theta(b/B)$ credits toward the block transfers for touching those children. Buffer items thus do not run out of credits until reaching the leaves, at which point they never need to participate in flushing again. We must also pay for the block transfers associated with the splitting of nodes, but this cost can be amortized against touching the node that is split. □

4.2 Eager Deletion

The EAGER-DELETE procedure works much as the (a, b)-tree DELETE procedure described in [14]. The only difference is that we must worry about buffer items maintaining the search criteria when we move keys from children to to their parents. This DELETE implementation has the advantage that when it is called, the item with the given index is deleted from the tree immediately and its associated value can be reported if necessary. The procedure EAGER-DELETE incurs an amortized $O(1 + \log_a(N/B))$ block transfers but may incur $O(N/B)$ block transfers in the worst case. The procedure can be fully deamortized to perform $O(1+\log_a(N/B))$ block transfers in the worst case, but the technique for this deamortization does not allow for the insertion deamortization technique outlined in Section 4.4. That is, one may choose whether to deamortize INSERT or EAGER-DELETE, but not both. This deletion implementation can be partially deamortized down to $O(b(1 + \log_a^2(N/B)))$ block transfers in the worst case without affecting insertion deamortization though.

When we recursively call EAGER-DELETE on a subtree, we maintain that the root of that subtree has at least $a+1$ and no more than $\lfloor b/2 \rfloor$ children. We maintain these invariants since we will see that at some points during deletion, we may need to flush the buffer of a node or a node may have a key stolen by a child deeper in the recursion. We saw during insertion that FLUSHBUFFER should only be called on nodes with at most $\lfloor b/2 \rfloor$ children so that the node does not split while its buffer is being flushed. By also enforcing that a node has $a + 1$ children before visiting it recursively with the deletion algorithm, we ensure that even if a child of that node we later visit steals one of its keys, it still has at least a children.

We can now understand why the inequality $a + 1 \leq \lfloor (\lfloor b/2 \rfloor + 1)/2 \rfloor$ must hold. We preemptively split nodes that cannot afford to have all their children split without splitting themselves, which are nodes with at least $\lfloor b/2 \rfloor + 1$ children. We also preemptively increase the number of keys of a node by at least 1 if the node cannot afford to have a key stolen by a child. These two actions give rise to the inequality relating a and b. In particular, in order to implement a BRT with $a = 2$, we need

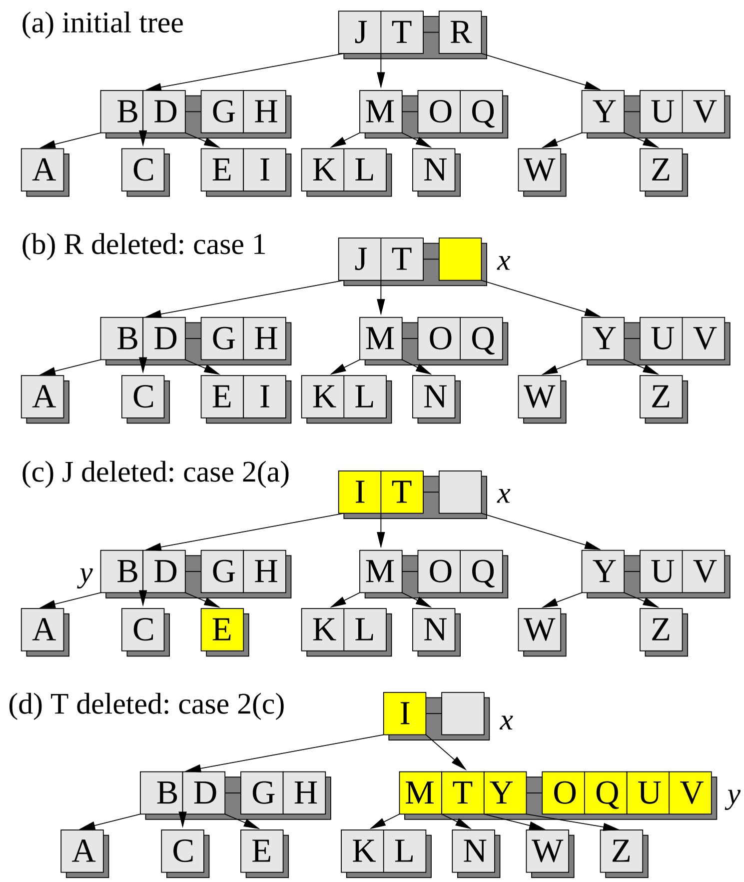
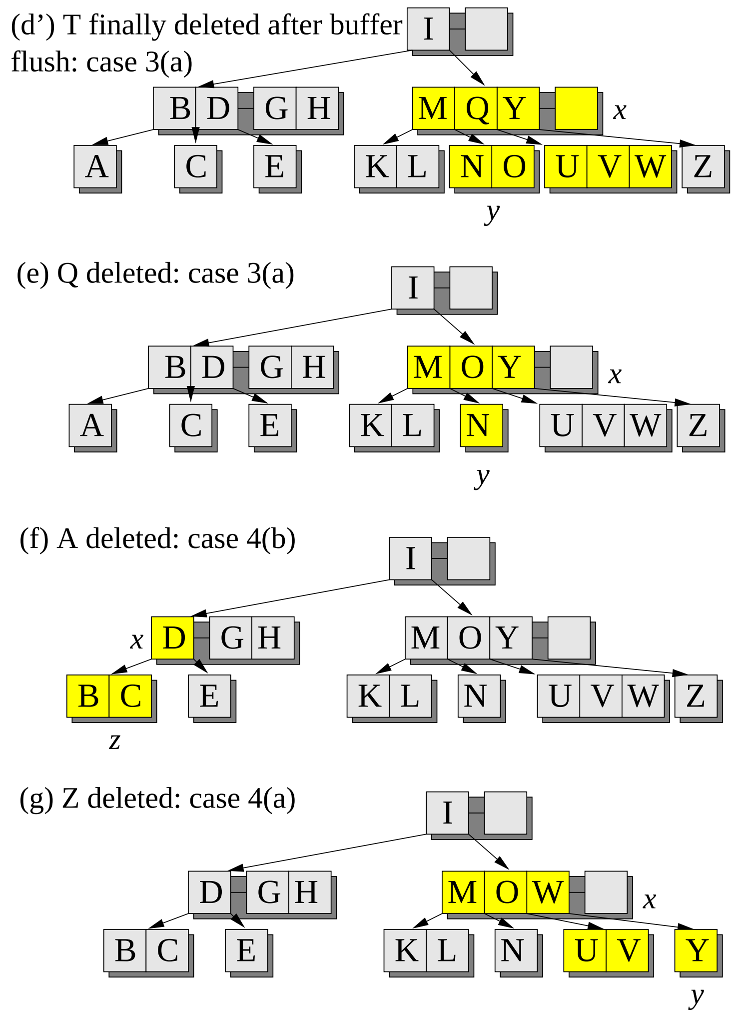


Figure 4-2: Illustration of the cases for the EAGER-DELETE operation on an (a, b)-BRT. Each node consists of its keys on the left and a buffer on its right, except for leaves which only contain buffers. The value B is 3, forbidding buffers from containing more than 3 items and leaves from containing less than 1 item. The value a is 2, and b is large enough that we need not split nodes in any examples. Keys and buffers whose contents changed between a deletion are highlighted in yellow. Some nodes are labeled x, y, or z to match the node labels in the description of the cases given.

that $b \geq 10$.

The EAGER-DELETE procedure is recursive down the tree and never need traverse back upward. We describe the procedure when asked to delete an item with index k. If we are currently at some node x, there are four cases we proceed by. These cases are all quite similar to those found in [14]. Figure 4-2 provides examples to help visualize the cases.

1. The index k appears in node x's buffer. In this case we simply remove the item from the buffer. If node x happens to be a leaf, we also terminate at this point.



2. The index k appears in one of x's keys, and x's children are not leaves. We proceed according to the following three subcases.

   (a) The child y of x immediately before k has at least $a+1$ children. In this case let k' be the predecessor of k in the subtree rooted at y. We find and recursively delete k'. Now, replace k with k' in x.

   (b) The child y of x immediately after k has at least $a+1$ children. We act as in the previous case, except we let k' be the successor of k amongst y's keys.

   (c) Each of the children immediately before and after k have exactly a children. In this case we merge the two children and move k down to be the median key by index in the newly created child, which we call y. If x is the root and k was its only index, flush x's buffer first. Since y's buffer contains elements that were merged from two previous nodes, its buffer may be overflowing. If so, call FlushBuffer(y). Now, recursively call Eager-Delete on k in y. If x was the root node and k was its only key, we delete the old root and make y the new root node. This case causes the height of the tree to decrease by 1.

3. The index k appears in one of x's keys, and x's children are leaves. This case is similar to case 2 and also has three subcases. The difference is that for cases (a) and (b) we look to see that the children before or after k have at least $\lceil B/2 \rceil$ children instead of $a+1$ children. Case 3(c) is different from case 2(c) in that we never need to call FlushBuffer. If both the children immediately before and after x have exactly $\lceil B/2 \rceil - 1$ items in their buffers, we simply merge them then delete k from x.

4. The index k does not appear in one of x's keys but appears in the subtree rooted at x. Let y be the child of x such that k appears in the subtree rooted at y. If y is an internal node and has fewer than $a+1$ children, or is a leaf with less than $\lceil B/2 \rceil$ items, we proceed according to the following two cases. Otherwise, we recursively delete from y, first splitting y if it contains more than $\lfloor b/2 \rfloor$ children and is an internal node.

   (a) The node y is has a sibling z that contains at least a keys if z is an internal node, or $\lceil B/2 \rceil$ items if z is a leaf. Let r be the key of x separating y from z. Without loss of generality assume that z is the right sibling (the other case is symmetric). Let k' be the smallest key in z. We move r to be the largest key in y and replace r with k' in x. If z is a leaf, we delete k from z and are done. Otherwise, we also move a child pointer in z so that the child immediately before k' in z now becomes the child immediately after r in y. We also move the buffer items in z with indices greater than r and less than k' from z to y, calling FlushBuffer(y) should its buffer become full during this movement of items. This calling of FlushBuffer on y does not cause y to split since it only has $a+1$ children, and we chose $b = 10 > 2a + 2 = 6$, which means y has enough key room even if every one of its children splits. We then recursively delete k from y.

(b) The node y does not have a sibling with at least $a + 1$ children. In this situation we merge y and one of its siblings into a new node z and move the appropriate key of x down to be the median key of z. If x was the root, it may now contain no keys, in which case we delete x and make z the new root, decreasing the height of the tree by 1. If y and z were not leaves, merging nodes may have caused z's buffer to have more than B items, in which case we call FlushBuffer on z. We now recursively delete k from z.

We initiate the procedure outlined above by calling Delete at the root. Recall Invariant BRT5, which says that for a given index k there is at most one key in the tree with index k. Furthermore, since an item in a key must have at some point been in a leaf, any other item in the tree with index k must have been inserted more recently and thus reside in the buffer of a node that is not a leaf. Otherwise, both the key item and the buffer item belong to the same node at some point in the (a, b)-BRT's history, which contradicts Invariant BRT5. Thus, during the deletion algorithm we are in Case 2 or 3 exactly once. All other insertions with index k are deleted in case one. It then follows that, aside from the flushing of buffers which is paid for by an amortized argument, the deletion algorithm makes exactly one pass down the tree. Deletion thus incurs an amortized $O(1 + \log_a(N/B))$ block transfers.

4.3 Lazy Deletion

This section describe the procedure Lazy-Delete, which is lazy and buffers deletions. This implementation of Delete incurs an amortized $O((b/B)(1+\log_a(N/B)))$ block transfers and can be deamortized to perform $O(1+\log_a(N/B))$ block transfers in the worst case. The deamortization technique in this scenario does not interfere with the INSERT deamortization outlined in Section 4.4. The procedure LAZY-DELETE suffers from the drawback, however, that the space taken by the deleted item is not reclaimed immediately. Furthermore, LAZY-DELETE has a negative effect on RANGE-SEARCH performance.

The $O(1+\log_a(N/B))$ cost of EAGER-DELETE can in fact be improved to $O((b/B)(1+\log_a(N/B)))$ amortized block transfers, matching the performance of insertion. We refer to this second DELETE implementation as LAZY-DELETE, which operates by buffering deletion operations. We show how to implement LAZY-DELETE when using an (a, b)-B$^+$RT, or an (a, b)-BRT where items are only kept in buffers and never in a node's keys. Node keys then do not actually contain $(index, value)$ pairs but just indices that help guide us toward the correct leaf during a query. This notion of only keeping items in leaves is an old idea, see for example the B$^+$-tree of Bayer and McCreight [4].

First, we see how to achieve $O((1/B)(1+\log_a(n/B)))$ amortized block transfers per deletion, where n is the number of items that have been inserted into the (a, b)-BRT throughout its history, as opposed to the current number of items in the (a, b)-BRT. We treat deletion just like insertion. When we call DELETE on index k, we actually call INSERT on an item with index k and value `delete`. During FLUSHBUFFER, when an item with key k and value `delete` is pushed to the buffer of a node y, any items in y's buffer with key k are deleted from the buffer. If y is a leaf, the `delete` item is also removed from the data structure. This procedure never causes the tree to shrink, and thus the tree may have size $\Omega(n)$. Hence LAZY-DELETE incurs $O((1/B)(1 + \log_a(n/B)))$ block transfers instead of $O((b/B)(1 + \log_a(N/B)))$.

In order to achieve achieve an amortized $O((b/B)(1+\log_a(N/B)))$ block transfers, we rebuild the data structure after every $N/2$ updates to maintain that our (a, b)-BRT is always $\Theta(N)$ in size. In order to rebuild the (a, b)-BRT, we first allocate an array of size N and do a tree-walk over the (a, b)-BRT, placing all items in this array. When placing an item in the array, we first turn it into a tuple where the first coordinate is the actual item and the second is the item's depth in the tree. An array item x

is compared to item y primarily by index, and ties are broken by depth (items with smaller depth are smaller). We now sort the array using $O((N/B)\log_{M/B}(N/B))$ block transfers using the external-memory sorting procedure of Aggarwal and Vitter [2]. Finally, we scan the array and insert the items into a brand new (a,b)-BRT, skipping items with a `delete` value or index equal to that of the item in the previous position. This rebuilding of the (a,b)-BRT costs a total of $O((N/B)(1+\log_a(N/B)))$ block transfers, which can be amortized against the $N/2$ updates.

The procedure Lazy-Delete adversely affects the performance of Range-Search. If we perform a Range-Search on the range $[k,k']$, the bound for Range-Search becomes $O(1+\log_a(N/B)+S'/B)$, where S' is the number of items with indices in the range $[k,k']$ that were in the (a,b)-BRT since the last time we rebuilt the data structure. Since deletions are lazy, many leaves may exist that contain indices in the range $[k,k']$, even though those items have actually been deleted.

4.4 Deamortization of Updates

Though the amortized insert bound of the (a,b)-BRT is superior to that of the B-tree, the worst-case performance of a single Insert or Delete may make the data structure undesirable. The worst-case updates for B-tree incur $\Theta(\log_B N)$ block transfers, proportional to the height of the tree. The worst-case updates for an (a,b)-BRT can also be made to be the height of the tree.

We show how to deamortize the FlushBuffer procedure. During the flush of a node x, some child of x must own at least $\lceil B/b \rceil$ of the buffer's items, by the Pigeonhole principle. Thus, we can maintain the same amortized Insert performance by only flushing to this *heavy child*. The worst-case Insert then becomes the height of the tree, which is $O(1+\log_a(N/B))$, while the amortized complexity remains unchanged.

Before discussing how to deamortize Eager-Delete, let us first see why the worst-case number of block transfers is not $O(1+\log_a(N/B))$. There are two cases when a node receives extra items in its buffer during deletion, which then cause us to call FlushBuffer. Both cases involve the scenario when we want to recursively call Delete on a node x with only a children. In the first case, x has some sibling y with at least $a+1$ children. In this case, we move one key from y to x's parent, then one key from that parent to x. This moving of keys causes us to move buffer items from y to x, and there could be as many as B of these buffer items. Thus, x's buffer could reach a size of $2B$. Simply flushing buffer items to x's heaviest child may not bring x's buffer size back down to B, and thus we may have to flush as many as B items. In the second case, x had no sibling with at least $a+1$ children, and so we merged a with some sibling. This scenario may also cause x's buffer to have $2B$ items. We can remedy both of these scenarios by calling the deamortized FlushBuffer until x's buffer has at most B items, causing at most b calls to FlushBuffer for a total cost of $O(b(1+\log_a(N/B)))$ block transfers when visiting x. Since we visit $O(1+\log_a(N/B))$ nodes from a root-to-leaf path during Delete, the worst case then becomes $O(b(1+\log_a^2(N/B)))$.

In order to completely deamortize Eager-Delete to incur $O(1+\log_a(N/B))$ block transfers in the worst case, we avoid needing to call FlushBuffer during deletion altogether. We avoid calling FlushBuffer by maintaining an additional invariant that for any node x, none of x's children own more than $\lceil B/b \rceil - 1$ of the items in x's buffer. I refer to this invariant as the *aggressive flushing invariant* (AFI). We maintain the AFI by adjusting the insertion algorithm so that whenever we visit a node x, we flush exactly $\lceil B/b \rceil$ buffer items to any child that owns at least $\lceil B/b \rceil$ items in x's buffer. This aggressive flushing may cause us to flush to many children, which could then have a cascading effect throughout the tree. The worst-case performance of Insert then becomes $O(N/B)$ block transfers, as we may have to flush buffers throughout the entire tree. With the AFI maintained though, it will still be maintained when merging nodes or adding a key to a node during deletion, thus avoiding calls to FlushBuffer during a call to Eager-Delete.

We now also discuss how to deamortize Lazy-Delete. This issue is very much related to the issue of space reallocation for the data structure. When we delete half of the items in the data structure, we would like its size to shrink, and similarly we want the data structure to double in size when N doubles. We use the common technique of *global rebuilding* [24, Ch. 5]. Global rebuilding is some operation a data structure must undergo at regular intervals before the data structure can handle future queries or updates. We already saw that lazy deletion causes us to undergo a type of global rebuilding every $N/2$ operations. As for growing the data structure, we initially allocate some constant number of nodes for the BRT. Whenever N doubles we completely rebuild the BRT and allocate double the amount of space. De Berg *et al.* [15] provide a useful technique for the deamortization of global rebuilding. Their theorem involves deamortizing the running time of global rebuilding, but the theorem naturally extends to deamortizing block transfers as well. We state the relevant theorem here without proof.

Theorem 8. *Let D be a data structure such that an update incurs at most r amortized block transfers and t block transfers in the worst case, D must undergo global rebuilding after every at most m updates, and global rebuilding incurs at most mr block transfers. Suppose global rebuilding can be paused and re-continued during its operation, and we know how many steps of the global rebuilding algorithm we can run before incurring $\Omega(r)$ and $O(t)$ block transfers. Then, there exists another data structure D' supporting all the same operations such that the amortized complexity of all operations is unchanged and D' incurs $O(t)$ block transfers per update in the worst case and $O(r)$ amortized block transfers per update.* □

In our case $m = \Theta(N)$, $t = O(1+\log_a(N/B))$, and $r = O((b/B)(1+\log_a(N/B)))$. For lazy deletion, we saw that global rebuilding incurs $O((Nb/B)(1 + \log_a(N/B)))$ block transfers. Reallocation of space for the BRT when it doubles in size costs $O(N/B)$ block transfers since we merely have to copy our data structure into a region that contains more allocated space. Both of these quantities are $O(mr)$.

Chapter 5

Experimental Methodology and Results

I implemented a memory-mapped (`mmap`'d) version of the BRB$^\epsilon$T and compared its performance with that of an `mmap`'d B$^+$-tree provided by Bradley C. Kuszmaul. I measured performance between disk and memory for disk block sizes of 1 kilobyte, 4 kilobytes, and 16 kilobytes. For each of these block size values, I was able to get a performance increase for insertion for the BRB$^\epsilon$T compared with the B-tree at a small constant factor loss in search performance. In the largest test case, where the size of the database was over 15 times the amount of physical memory on the machine, I was able to achieve roughly a 17 times performance increase of BRB$^\epsilon$T insertion over B-tree insertion, while SEARCH was slower by roughly a factor of 3.

This chapter is organized as follows. Section 5.1 provides details of the implementation and the experimental setup. Section 5.2 then discusses the results of various experiments.

5.1 Experimental Setup

This section describes experimental setup that was used for experiments comparing the BRB$^\epsilon$T, BRT, and B$^+$-tree. We also discuss some implementation details of the data structures.

All data structures were implemented via memory mapping (`mmap`). The C function `mmap` allows a user to map portions of a file to pages in memory. The user can then read and write bytes of the file using pointers as if they were just bytes in memory. In the BRT implementation I used a $(2, 6)$-tree instead of the $(2, 10)$-tree that we saw was necessary in Section 4.2. Since I did not experimentally evaluate the DELETE operation, this setting of b for the (a, b)-tree base of the BRT sufficed for proper operation of INSERT and SEARCH.

All experiments were performed on an AMD Athlon 4800+ dual-processor machine booted with 128 megabytes of main memory and running the Linux 2.6.12 kernel. The hard disk was SCSI with a 4-kilobyte block size. I also ran experiments with node sizes of 1 kilobyte and 16 kilobytes to compare the BRT, BRB$^\epsilon$T, and B$^+$-tree for varying block sizes.

I tested INSERT and SEARCH performance when the database contained $N = 2^k$ elements, for $k = 10, 11, \ldots, 27$. Each item was 12 bytes in size, with keys taking 8 bytes and values taking 4 bytes. Since a database containing N values for the largest value of N takes over 2 gigabytes of space, it was prohibitively slow to build the databases from scratch using 128 megabytes of memory in order to test INSERT performance. I thus booted the machine with 4 gigabytes of memory in order to create all the databases before rebooting with 128 megabytes of memory to run experiments.

The experiment setup was as follows:

1. Boot the machine with 4 gigabytes of memory and create databases with N items for $N = 2^{10}, 2^{11}, \ldots, 2^{27}$.

2. Reboot the machine with the `mem=128M` boot option.

3. For each database, perform k random searches where $k = \min\{\lfloor N/10 \rfloor, 2^{16}\}$. Measure the average cycle count per SEARCH by measuring the cycle count before running the searches and afterward using the `rtdsc` assembly call.

4. For each database, insert k items at random. Then, call a synchronous `msync` on the database file which causes all dirty pages to be written to disk. The average cycle count per INSERT includes the cycles spent during `msync`.

		PIR	OIR	PSR	OSR
$B = 1$ kilobyte	BRT	5.8	5.9–7.7	3.5	4.4–4.7
	BRB$^{1/2}$T	5.1	6.1–6.6	2.8	3.3–3.5
$B = 4$ kilobytes	BRT	18.7	18.4–23.6	4.5	3.4–6.7
	BRB$^{1/2}$T	10.8	16.7–26.0	2.5	1.8–3.1
$B = 16$ kilobytes	BRT	61.8	56.6–93.4	5.5	2.0–6.1
	BRB$^{1/2}$T	23.9	51.7–212.3	2.4	0.4–1.9

Figure 5-1: Comparison between observed and predicted performance differences between each of the BRT and BRB$^{1/2}$T with the B$^+$-tree. The column PIR shows the predicted ratio between the insertion performance of the B$^+$-tree and the given data structure, and OIR gives the observed ratio. The column PSR shows the predicted ratio between the search performance of the given data structure and the B$^+$-tree, and OSR gives the observed ratio.

5. Space used by the database is reported after the k insertions.

I started all experiments with the database `mmap`'d into a region large enough so that global rebuilding for space reallocation never needed to take place. In an amortized sense, the number of block transfers for this global rebuilding should be insignificant, since each item must only pay an amortized $O(1/B)$ block transfers toward global rebuilding, while the costs of both searches and updates surpass this value by at least a logarithmic factor.

During FlushBuffer, I sorted the buffer in order to decide which buffer items should go to which children. The quicksort implementation I used for this sorting was provided by Michael Tokarev [29].

5.2 Experimental Results

I tested three values of the block size: 1 kilobyte, 4 kilobytes, and 16 kilobytes. The actual block size on the disk of the computer was 4 kilobytes. Figures 5-2, 5-3, and 5-4 show comparisons between the BRB$^{1/2}$T and the B$^+$-tree insertion and search performance for various block sizes for values of N large enough so that the database does not fit into main memory. Figures 5-5, 5-6, and 5-7 show comparisons between the BRT, BRB$^\epsilon$T, and B$^+$-tree visually, where I experimented with several values of the parameter ϵ.

We now discuss predictions of the behavior of the BRT, BRB$^{1/2}$T, and B$^+$-tree. In the implementation of each of these data structures, each node fits tightly into a block. Thus, when B is 1 kilobyte, at most $\lfloor 1024/12 \rfloor = 85$ items can fit into a single node for the BRT and BRB$^{1/2}$T. In the B$^+$-tree, since internal nodes only hold the keys of items, each internal node can hold $\lfloor 1024/8 \rfloor = 128$ items. Since each internal node of a BRT carries between 2 and 6 children, for the sake of estimation we assume the average branching factor in the tree is 4. The expected performance increase of insertion for the BRT over the B$^+$-tree for large values of N is then $((85 - 4)/4) \cdot (\log_{128} N / \log_4 N) \approx 5.8$. The experimentally observed performance increase was between 5.9 and 7.7 for large values of N. We expect the search performance of the BRT to be worse by a factor of $\log_4 N / \log_{128} N \approx 3.5$. Experimental results showed a performance decrease by a factor between 4.4 and 4.7. For the BRB$^{1/2}$T, $\lfloor \sqrt{\lfloor 1024/12 \rfloor} \rfloor = 9$ and the minimum number of children an internal node contains is thus $\lfloor (\lfloor 9/2 \rfloor + 1)/2 \rfloor = 2$. We again approximate and say the average branching factor is 5.5. We thus expect the performance increase for BRB$^{1/2}$T insertion to be $((85 - 5.5)/5.5) \cdot (\log_{128} N / \log_{5.5} N) \approx 5.1$. A similar calculation to the BRT also shows that search is expected to be 2.8 times slower. Experiments showed a 6.1–6.6 times difference in insertion performance and 3.3–3.5 difference in search performance. Comparisons between predicted and observed performance differences are shown for other values of B in Figure 5-1.

One may notice that there are some discrepancies between the predicted and observed performance numbers shown in Figure 5-1. The actual block size on disk used by the experiments is 4 kilobytes, and so one potential reason may be that our bounds do not accurately predict performance for the case when B is 1 or 16 kilobytes. Another issue is that both the BRT and BRB$^\epsilon$T are amortized, while the B$^+$-tree has equivalent worst-case and amortized performance. Since I was not able to time all N insertions into the BRT and BRB$^\epsilon$T, as doing so would have taken a prohibitively long amount of time, the insertions that were timed may have been at times when the data structure had much potential built up or was spending potential built up from previous, untimed insertions. I hoped to avoid such a scenario by using random insertions, causing the data structure to be in an “average” state after N insertions. I have not, however, rigorously proven that this approach reports results near the expected results with high probability.

N	B$^+$-tree / BRB$^{1/2}$T insert ratio	BRB$^{1/2}$T / B$^+$-tree search ratio
2^{23}	6.4659	3.27802
2^{24}	6.6104	3.44688
2^{25}	6.64002	3.5089
2^{26}	6.15954	3.32124
2^{27}	6.06808	3.3162

Figure 5-2: Comparison of BRB$^{1/2}$T and B$^+$-tree Insert and Search performance with a 1-kilobyte block size. Performance is measured as cycles/operation.

N	B$^+$-tree / BRB$^{1/2}$T insert ratio	BRB$^{1/2}$T / B$^+$-tree search ratio
2^{23}	20.3807	1.78757
2^{24}	25.9846	1.93805
2^{25}	18.8904	2.2635
2^{26}	20.1987	2.64267
2^{27}	16.6608	3.06705

Figure 5-3: Comparison of BRB$^{1/2}$T and B$^+$-tree Insert and Search performance with a 4-kilobyte block size. Performance is measured as cycles/operation.

N	B$^+$-tree / BRB$^{1/2}$T insert ratio	BRB$^{1/2}$T / B$^+$-tree search ratio
2^{23}	52.8663	0.350202
2^{24}	153.796	0.420194
2^{25}	170.755	0.453863
2^{26}	212.251	0.804504
2^{27}	51.6713	1.87277

Figure 5-4: Comparison of BRB$^{1/2}$T and B$^+$-tree Insert and Search performance with a 16-kilobyte block size. Performance is measured as cycles/operation.

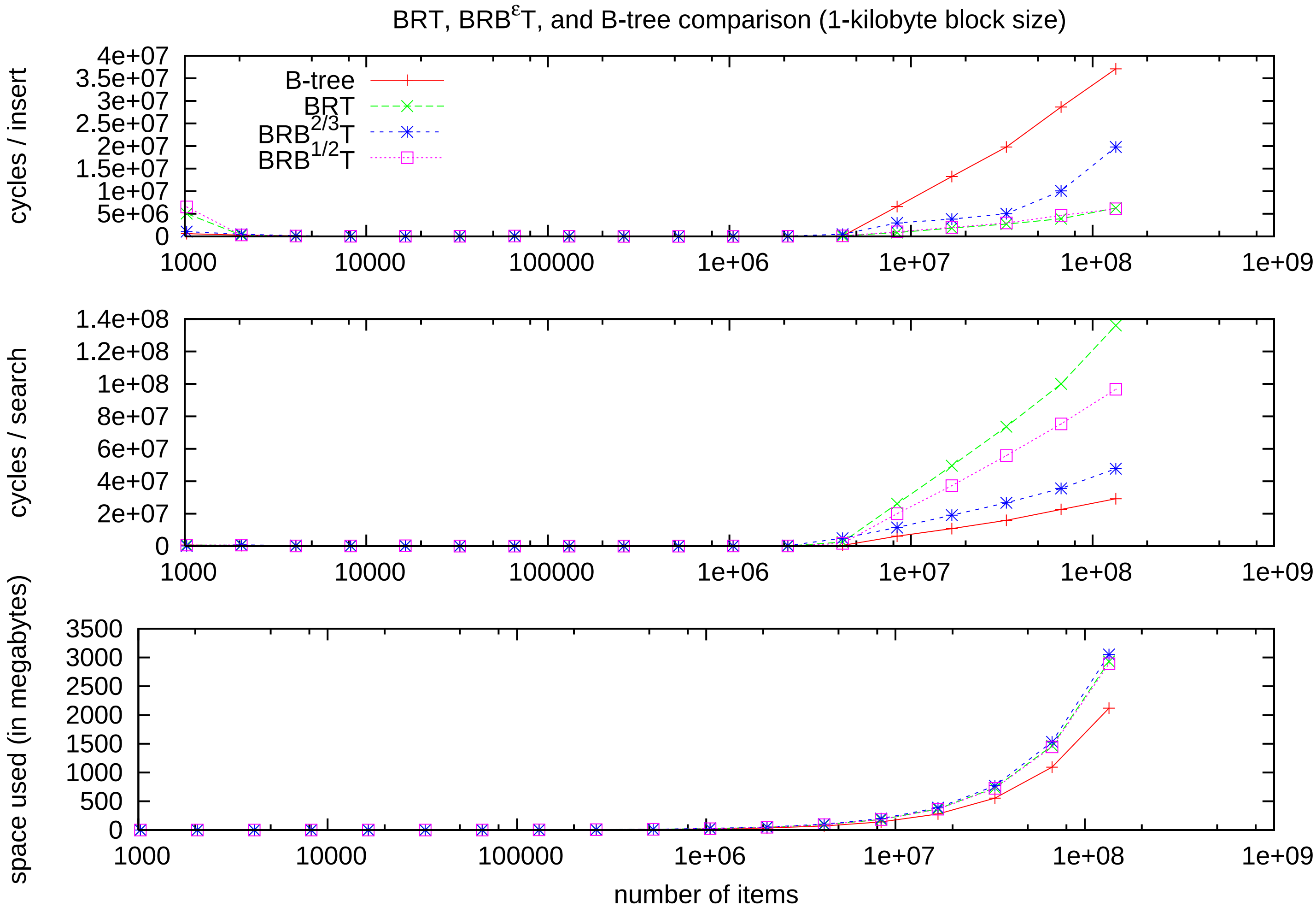


Figure 5-5: Update/query performance comparisons of the BRT, B^{ϵ}-tree, and B^{+}-tree with a 1-kilobyte block size.

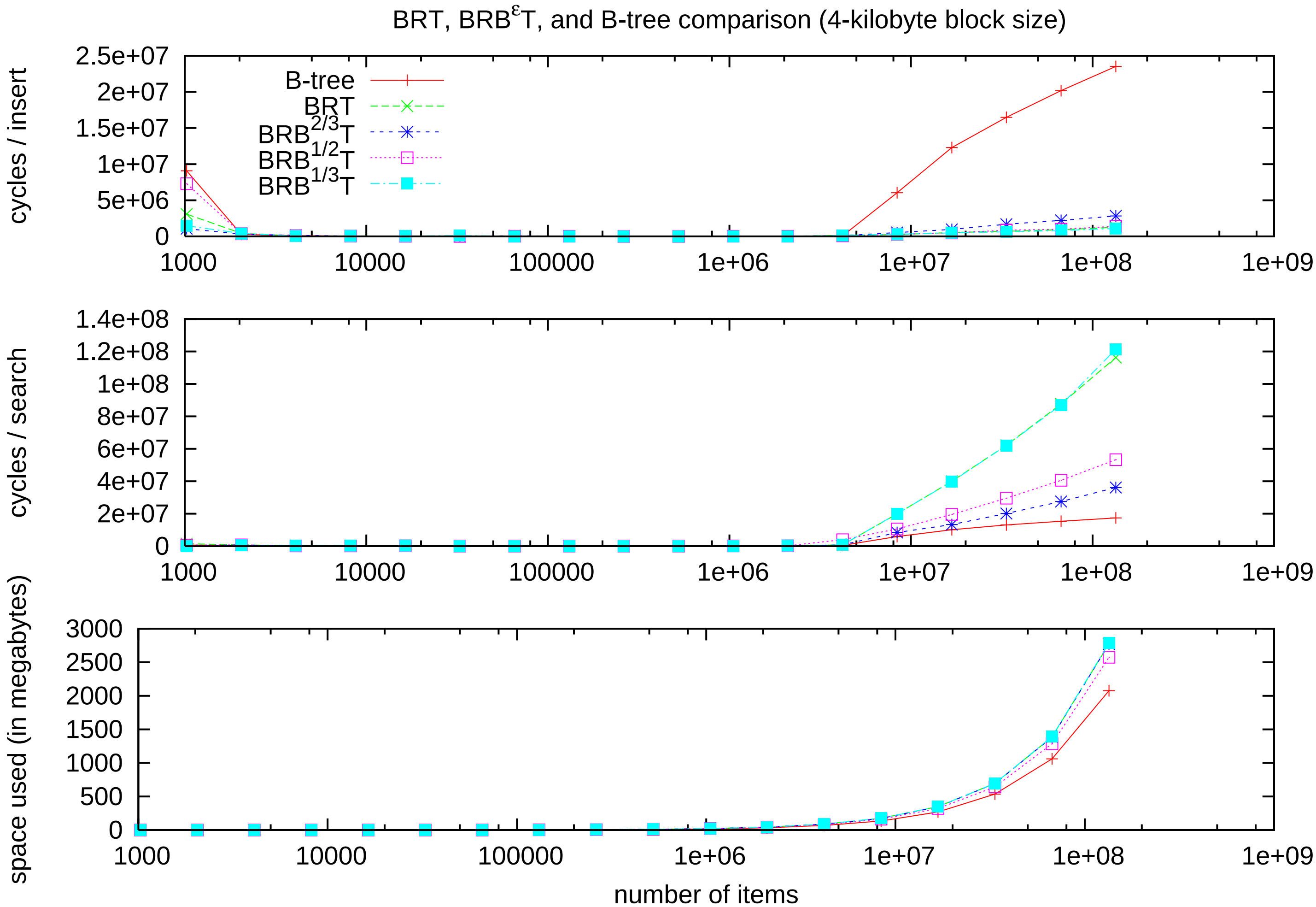


Figure 5-6: Update/query performance comparisons of the BRT, B$^\epsilon$-tree, and B$^+$-tree with a 4-kilobyte block size.

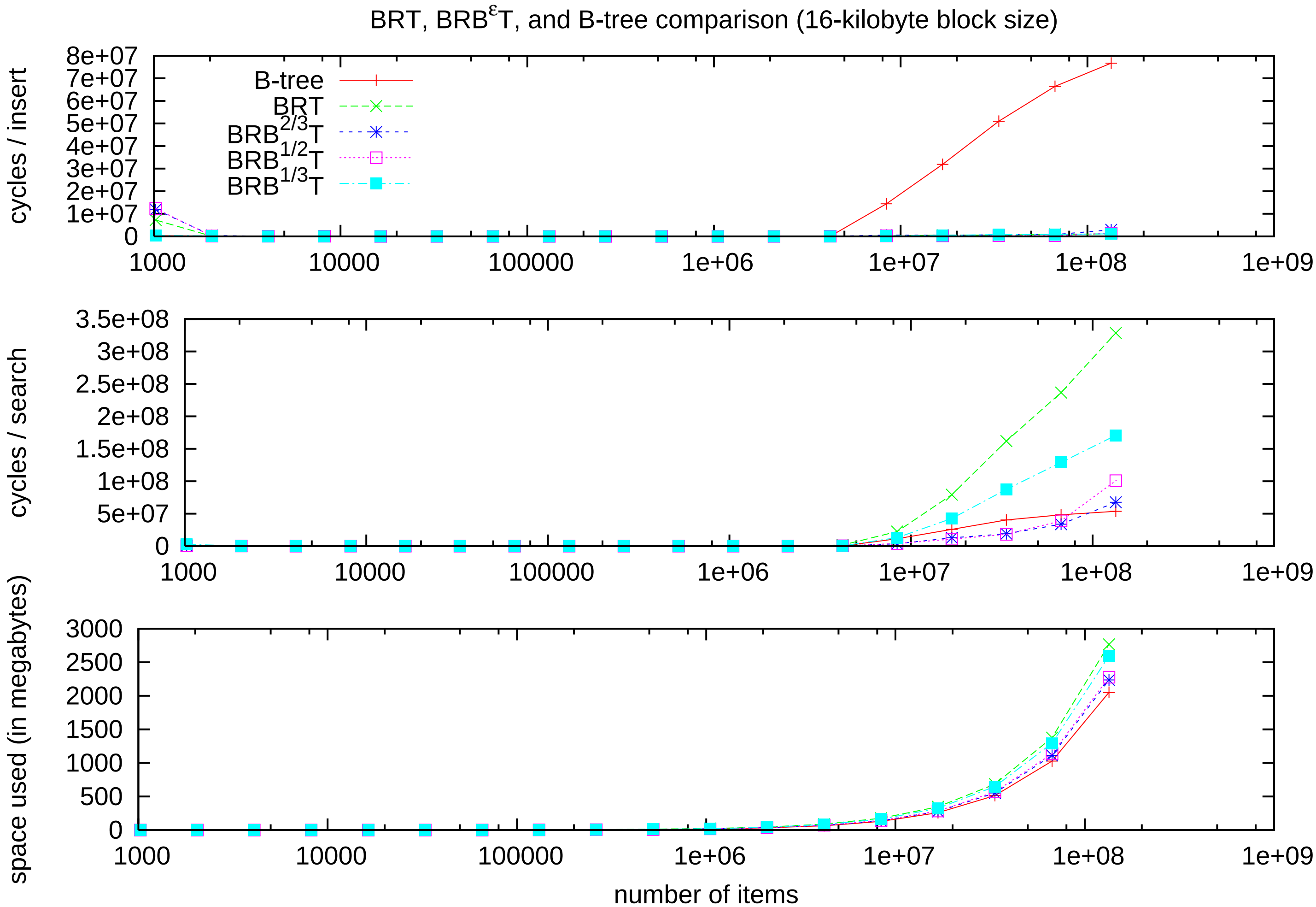


Figure 5-7: Update/query performance comparisons of the BRT, B$^{\epsilon}$-tree, and B^{+}-tree with a 16-kilobyte block size.

Chapter 6

The Cache-Oblivious Lookahead Array

This chapter describes a cache-oblivious structure, the cache-oblivious lookahead array (COLA) [9], which achieves the same I/O bounds as the basic buffered repository tree. The data structure was initially proposed by Michael A. Bender and Bradley C. Kuszmaul. Their data structure had slow worst-case performance, however, and its deamortization is a contribution of this thesis and was done jointly with Jeremy Fineman.

We study the COLA in three stages. Section 6.1 presents the basic idea which has the desired $O((1/B)\log(N/B))$ Insert performance, but with $O(N/B)$ worst-case performance for Insert and poorer Search performance than the BRT. Section 6.2 discusses how to deamortize Insert to yield $O(\log(N/B))$ worst-case performance, matching that of the BRT. We investigate how to speed up Search by using what we refer to as "lookahead pointers" in Section 6.3. Section 6.4 discusses how to resolve complications that arise when combining the deamortization technique of Section 6.2 with the lookahead pointers of Section 6.3.

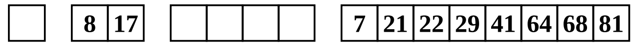


Figure 6-1: COLA without lookahead pointers with $N = 10$.

6.1 The Basic Data Structure

The basic COLA consists of $\lceil \lg N \rceil$ arrays, each of which is either completely full or completely empty. The kth array is of size 2^k and the arrays are stored contiguously in memory. Figure 6-1 shows an example of a basic COLA when the number of items N is 10. Two invariants we maintain are the following:

COLA1 If we write N in binary, the kth array contains items if and only if the kth least significant bit of N is a 1 (where the rightmost bit is the 0th least significant bit).

COLA2 Each array contains its items in sorted order.

To maintain these invariants, when a new item is inserted, we merge the items in the longest prefix of full arrays, together with the new item, into the smallest array that is empty.

DELETE is much like LAZY-DELETE for the BRB$^\epsilon$T. When deleting an index k, we insert an item with index k and value `delete`. When merging arrays across levels we may need to skip over some older copies of items with the same key or items that have been deleted. We can tell the order of an insertion versus a deletion since the more recent update resides in an earlier level, and there is at most one item with a given index at each level. Due to this skipping though, the array we merge into may not be completely full after the merge. We solve this problem by filling any remainder slots in the array we merge into with dummy items with ∞ indices. In order to maintain that the data structure is always $\Theta(N)$ in size, we rebuild the entire data structure after every $N/2$ updates. This rebuilding can be deamortized by the result of Theorem 8.

Lemma 9. *Each of the* INSERT *and* DELETE *procedures for the COLA incur an* *amortized* $O((1/B)\log(N/B))$ *block transfers and* $O(N/B)$ *block transfers in the worst case.*

Proof. We only analyze Insert, as the analysis for Delete is identical. First, let us assume that $\lg(N/B) < cM/B$ for some sufficiently small constant c so that we have enough cache lines to store one block in cache from each array of the COLA. We give an item $\Theta((1/B)\log(N/B))$ credits upon insertion. Once we move an item $\lceil \lg N \rceil - 1$ times to bigger and bigger arrays, it is then in the last array. Whenever an item in the last array is moved, the data structure must have doubled in size. In this case, since the newly inserted items can pay for the movement of the elements in the last array, items need only pay for their first $\lceil \lg N \rceil - 1$ moves.

For the first $\lg B$ moves of the item, since those $\lg B$ arrays fit into a constant number of blocks which we can assume are always kept in cache, those moves incur no cost. Otherwise, we need to merge all items from $O(\log N)$ levels into some empty level. Since we have enough cache lines in cache to hold at least one block from each level, we do not incur any block transfers when looking for the next smallest element in the merge. Thus, merging a total of k items from all these levels only incurs $O(1 + k/B)$ block transfers. Since $k > B$, this cost can be paid by each item contributing $1/B$ credits toward moving. An item is only charged in this way $\lceil \lg N \rceil - \lg B = O(\log(N/B))$ times for a total amortized complexity of $O((1/B)\log(N/B))$.

We can eliminate the assumption that we have $\Omega(\log(N/B))$ cache lines at the cost of doubling the space our data structure uses. Instead of merging all arrays at once, we only merge two at a time and have two arrays of each size (two arrays of size 1, two of size 2, etc.). Thus, when merging we are only working with two active blocks at a time and only need that $M/B = \Omega(1)$.

As for the worst case, due to global rebuilding we always maintain that the COLA is $\Theta(N)$ in size. Thus, during an update, we only ever need to merge at most $O(N)$ items, which can be done in $O(N/B)$ block transfers. □

The trouble with the basic COLA is that queries are slower than the BRT. For example, we can implement the Search procedure by binary searching through each array separately. Since the most recent updates to an index reside in earlier levels, we should return the first item of matching index we find. Once binary search reaches a level of recursion where only B items in the array are left, these items fit into at most 2 blocks. Thus, the number of block transfers incurred by binary searching through one array is $O(\log(N/B))$, since the largest array is of size $\Theta(N)$. There are $O(\log N)$ arrays total, but searching through the first $\lg B$ can be done in at most two block transfers. Thus, SEARCH incurs $O((\log(N/B))(\log N - \log B)) = O(\log^2(N/B))$ block transfers. In section 6.3 we show how to speed up this performance to $O(\log(N/B))$ block transfers to match the SEARCH performance of the BRT.

We can implement PREDECESSOR and RANGE-SEARCH similarly to SEARCH. For PREDECESSOR, we binary search through each array for the given key, and the predecessor in a given array is the index right before the one we find. We then have $O(\log N)$ items, one in each array, and we should take the maximum index item to find the predecessor. We break ties in indices by treating the item in an earlier level as being larger. The PREDECESSOR procedure thus incurs an equal number of block transfers as SEARCH. In order to implement RANGE-SEARCH, given two keys k and k' to search between, we binary search for each key in each array. We thus find a contiguous segment of each array containing the items we should return. Since some of these items may have duplicate indices though, we should merge the segments into one array, skipping over older copies of items with equal indices, before outputting our answer. Since there are $O(\log N)$ copies of each item (at most one per level), RANGE-SEARCH incurs $O(\log^2(N/B) + \log N(S'/B))$ block transfers, where S' is the number of data items with an index in the range $[k, k']$ that have been in the COLA since the last global rebuilding (which was at most $N/2$ updates ago). We pay this cost since some items which have been deleted may still be taking up space in the data structure.

6.2 Deamortization

In this section we see how to partially deamortize the INSERT performance of the cache-oblivious lookahead array when $M = \Omega(\log^2 N)$ and $M/B = \Omega(\log N)$, thereby improving the worst-case bound from $O(N/B)$ to $O(\log(N/B))$. This bound matches the worst-case INSERT bound of the buffered repository tree. We cannot hope to incur $O((1/B)\log(N/B))$ I/O's in the worst case since this bound is quite often sub-constant.

The idea is as follows. For each level k of the lookahead array we keep two arrays each of size 2^k. Whenever a level contains items in both of its arrays, we try to merge those two arrays into the next level. Thus, each level is in one of two modes, which we refer to as *unsafe* and *safe*. Informally, a level that is in the middle of merging its two arrays is unsafe. More formally, initially all levels are safe. Level k become unsafe once it contains exactly 2^{k+1} items, and an unsafe level becomes safe only once it is completely empty again.

Inserts are done into level 0 and give us m credits. (The value m will be chosen later.) These credits can only be spent during the INSERT they were obtained in. During an INSERT, we scan the levels from left to right merging items from unsafe levels to the next level, and stop once either no unsafe levels are left or we have moved m items during the INSERT. After an insertion, since some levels may still be in the middle of merging, we must remember the current indices we are at in the merge at a particular level for the next time we visit that level. We thus keep three arrays of size $\lceil \lg N \rceil$. One array keeps a bit in the ith entry to keep track of whether level k is safe or unsafe. The other two arrays' ith entries hold the indices we are at in the merge of the two arrays at level k. These three arrays combined thus take $O(\log^2 N)$ space (the arrays are each $O(\log N)$ in size, and in two of the arrays each position contains a $\log N$-bit index). Assuming that $M = \Omega(\log N \log(N/B))$, we can assume the three arrays are always in cache.

Lemma 10. *If a lookahead array contains k levels, we can choose $m = 2k$ to guarantee that two adjacent levels are never simultaneously unsafe.*

Proof. Since we resolve unsafe levels from left to right, we may choose an inductive approach. Our hypothesis claims that levels i and $i+1$ are never simultaneously unsafe. One may verify the claim for $i = 0$, since if we take a snapshot of the data structure after each insertion, the pattern of safeness and number of items of levels 0 and 1 is periodic.

We wish to show that if level ℓ is unsafe, it becomes safe before level $\ell - 1$ can become unsafe. When level ℓ becomes unsafe, level $\ell - 1$ is empty. Furthermore, for us to have moved items from level $\ell - 1$ to level ℓ, all previous levels had to be safe (since we process unsafe levels from left to right) and thus contain at most $\sum_{j=0}^{\ell-2} 2^j = 2^{\ell-1} - 1$ items. Thus, for level $\ell - 1$ to become unsafe again, there must be at least $2^{\ell-1} + 1$ inserts. We show that level ℓ becomes safe after at most $2^{\ell-1}$ insertions. After $2^{\ell-1}$ inserts, we have accumulated at least $m\ell 2^{\ell-1}$ moves, and no move will go unused as long as level ℓ is unsafe.

After $2^{\ell-1}$ insertions there can be at most $2^{\ell} - 1$ items in levels $\ell - 1$ and below, and each one could have used at most $\ell - 2$ moves to get to its level. Thus, items below level ℓ could have used a total of at most $(\ell - 2)(2^{\ell} - 1)$ moves. We want to have at least $2^{\ell+1}$ credits left over to move all the items from level ℓ to the next level, so we want $m2^{\ell-1} \geq (\ell - 2)(2^{\ell} - 1) + 2^{\ell+1}$. One may check that this inequality is satisfied when $m = 2\ell$. □

Theorem 11. *The cache-oblivious lookahead array can be deamortized to perform only $O(\log(N/B))$ block transfers per* Insert *in the worst case.*

Proof. By Lemma 10, an unsafe level never wants to spill over into another unsafe level. Since the number k of total levels is at most $\lceil \lg N \rceil$, we only move $O(\log N)$ items per insert. Since the first $\lg B$ levels fit in one block though, any moves in those levels need only be paid for once. Thus, Insert performs $O(\log N - \log B) = O(\log(N/B))$ block transfers in the worst case. □

Theorem 12. *Suppose $M = \Omega(\log^2 N)$ and $M/B = \Omega(\log N)$. Then, the deamortized cache-oblivious lookahead array has an amortized performance of $O((1/B)\log(N/B))$ for* Insert.

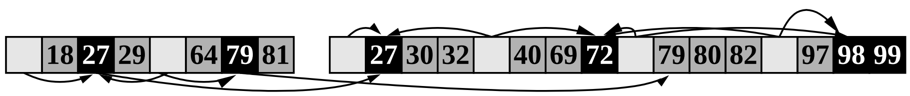


Figure 6-2: Levels 3 and 4 of a COLA with lookahead pointers. Empty cells represent duplicate pointers and point to the nearest real lookahead pointers to their left and right. Cells with black backgrounds are real lookahead pointers. All other cells contain normal items.

Proof. The argument is roughly the same as for the original lookahead array. The only difference is now notice that we may incur a block transfer at a level beyond the $\lg B$th level without actually getting to move B items (because of the limit of how much we can move per INSERT). Since $M = \Omega(\log^2 N)$, we have enough space in cache to keep track of where at each level we paused any ongoing merges. Since $M/B = \Omega(\log N)$, we can also fit one block from each level. We can thus assume that we retain the blocks from which we moved last at each level in cache so that the amortized move of one item is always $1/B$. If such a block should be evicted, we can charge the evicting block transfer to put that block back in cache. $\square$

In Theorem 11, the setting $m = 2k$ is most likely not the best possible. In the proof of Theorem 11, we did not aim to optimize the constant multiplying k. We believe that $m = k + 1$ suffices, but have yet to prove it. We have proven by computer that $m = k + 1$ suffices when $N \leq 10^9$.

6.3 Lookahead Pointers

Thus far, we have shown how the COLA achieves the same INSERT performance, both amortized and worst-case, as the BRT, but cache-obliviously. We now show how to modify the data structure in order to implement query procedures that match the performance of BRT query procedures.

The idea is to use what we refer to as *lookahead pointers*. Figure 6-2 shows two levels of a COLA with lookahead pointers. A constant fraction (exactly half) of each array in the COLA is dedicated to holding these lookahead pointers. We maintain that each 8th element in the $(k + 1)$st array appears in the kth array in its proper place according to its sorted rank, with a pointer to its place in the $(k+1)$st array. We call the copy of this element in the kth array a *real lookahead pointer*. Each 4th element in the kth array is a *duplicate lookahead pointer*, which holds pointers to the nearest real lookahead pointers to its left and right.

Now, we must make sure that these lookahead pointers do not take too much extra space. In an array of size m, every 4th cell is a duplicate lookahead pointer, for a total of $m/4$ lookahead pointers. Every 8th cell of the next array causes a real lookahead pointer to appear, which creates an additional $2m/8 = m/4$ lookahead pointers. Thus $m/4 + m/4$ cells, or exactly half of the array, is devoted to lookahead pointers while the other half can contain real items. To avoid special cases we also say that only the arrays of size 4 and above contain lookahead pointers. The array at level 0 is never used, and the array at level 2 contains at most 1 real item and no lookahead pointers. Thus, arrays at each level are exactly half full with real items.

Lemma 13. *In a COLA with lookahead pointers, we can implement* Search *to incur* $O(\log(N/B))$ *block transfers.*

Proof. The key aspect of lookahead pointers is that they allow us to only look at a constant number of cells of each array in the COLA during a Search. In fact, we only need to search at most 8 cells of each array. We prove inductively that if we are searching for index k in the ith array, we need only look at at most 8 contiguous cells in array i. That is, we can implement Search to only look at 8 contiguous cells with indices $k^i_j, k^i_{j+1}, \ldots, k^i_{j+7}$ in array i such that the first and last of the 8 cells are not duplicate lookahead pointers and $k^i_j \leq k \leq k^i_{j+7}$. The value k^i_j represents the index of the jth cell in array i, where j is 1-indexed.

The claim is certainly true amongst the first 4 arrays, as their sizes are all at most 8. At the ith level, where $i \geq 4$, if we search through $k^i_j, \ldots, k^i_{j+7}$ then either we find an ℓ so that $k^i_\ell = k$, or either $k^i_\ell < k < k^i_{\ell+1}$ and $\ell < 2^i$, $k^i_\ell > k$ and $\ell = 1$, or $k^i_\ell < k$ and $\ell = 2^i$. In the first two cases we follow the nearest lookahead pointers to the left and right of ℓ into the $(i+1)$st array, which again gives us exactly 8 cells to search in the $(i+1)$st array. When in the third and fourth cases we must search the first 8 or last 8 cells, respectively, in the $(i+1)$st array. $\square$

The $O(\log^2(N/B))$ terms in the PREDECESSOR and RANGE-SEARCH bounds can also be improved to $O(\log(N/B))$ using a similar method as for SEARCH. We can replace binary searches at each level with the usage of lookahead pointers to guide us instead.

We now discuss how to maintain lookahead pointers during insertions into the amortized data structure described in section 6.1. Section 6.4 then discusses how to alter our deamortization technique from Section 6.2 to deal with lookahead pointers.

Suppose we are merging levels $0, 1, \ldots, j$ into level $j+1$. Once the merge is complete, we need to populate the arrays in levels $0, 1, \ldots, j$ with lookahead pointers. We first do two simultaneous scans over levels j and $j+1$, where we scan over level $j+1$ 8 times as fast. For each 8th element in the array at level $j+1$, say those with indices in the array which are 1 (mod 8), we add a lookahead pointer to the array of level j to that item. Once this is complete, we do a scan over level j to set the duplicate lookahead pointers to the nearest real lookahead pointers to their right and left. We then repeat this process between levels j and $j-1$, then $j-1$ and $j-2$., etc., down until the array at level 2 has been populated with lookahead pointers. There is a special case when $j+1$ is the new last level of the array. In this case, since level $j+1$ had no lookahead pointers when we merged into it, it is only half full after the merge and contributes $2^j/8$ lookahead pointers to level j instead of $2^j/4$. In order to avoid this special case, when merging into a new last level of the COLA, we fill the empty half of the array with dummy items of index ∞.

6.4 Deamortization with Lookahead Pointers

We have seen how to speed up searches using lookahead pointers and how to partially deamortize the insertion performance of the COLA when there are no lookahead pointers. It is not *a priori* obvious, however, that these two techniques can be used simultaneously. For example, after an insertion some levels of the COLA are in the middle of being merged. How does one keep lookahead pointers maintained during such transitions so that the data structure can still be queried? Also, in the amortized structure, once a merge was complete we laid down lookahead pointers in all the empty arrays previous to the last level involved in the merge. In the deamortized structure, some previous levels may themselves have items in them that could get in the way. Despite these and a few other issues, we show that in fact lookahead pointers and the deamortization technique of Section 6.2 can be used simultaneously.

While it may be possible to directly attack the issues discussed above, such an approach may be tricky to get right. We provide an approach that completely hides the details of the deamortization from the queries. From the viewpoint of a query, no level is in the middle of merging its arrays. We accomplish this by using what we refer to as *shadow arrays*.

At each level we now maintain three arrays instead of two. Each array at level k is still of size 2^k. At least one array is a shadow a array and at least one is a *visible arrays* (unless level k is beyond the levels being used by the data structure yet, in which case all three arrays are considered shadow). This labeling of shadow versus visible is non constant throughout time though, and at times a shadow array may become visible, and vice versa. Since no items ever move from visible arrays, the data structure induced by only looking at visible arrays appears exactly as a COLA with lookahead pointers without deamortization. We can thus perform queries similarly to the method described in Section 6.3. The key difference from the deamortization method of Section 6.2 is that when level k becomes unsafe, we do not move items from level k to $k + 1$; we instead *copy* them to a shadow array of level $k + 1$. Thus, from the point of view of a query, the merge is not taking taken place.

More formally, all levels start as being *safe* and all arrays start as being shadow arrays. Level k becomes unsafe once two of its arrays become full. Two full arrays at an unsafe level would like to have their items merged into some shadow array A of level $k + 1$. If $k + 1$ is not the last level of the array, A is chosen to be an array that contains lookahead pointers. Otherwise, A can be any shadow array. After the merge, lookahead pointers are copied from A into an empty shadow array at level k. Level k does not become safe again until both the merge into A and the placement of lookahead pointers from A into level k are both complete. At this point, we say the shadow array of level k that received lookahead pointers becomes *linked* to A.

We now discuss the transition from shadow to visible and vice versa. Level 0 only contains two arrays, both of which are always visible. A shadow array at a higher level does not become visible until there is a sequence of linked arrays beginning at at level 0 to that array. When a shadow array does become visible, there are two cases. If there are two other visible arrays in that level, their statuses are both changed to shadow, and both arrays are then considered empty. Otherwise, if there are 0 or 1 other visible arrays, they remain visible.

Observe that safeness of a level in this scenario is similar to safeness in deamortization without lookahead pointers. They are similar in the sense that, in both scenarios, if level k is unsafe then we must scan $\Theta(2^k)$ items before level k becomes safe again (note that placing lookahead pointers can be done by two simultaneous scans). Thus, we can use the proof idea of Lemma 10 to state that two adjacent levels are never unsafe if we choose to move $\Theta(\log N)$ items per insertion. In order to show that we only need three arrays at each level, we use the following lemma.

Lemma 14. *Suppose level k becomes unsafe and merges into shadow array A at level $k+1$. Then, A becomes visible before another merge into level $k+1$ occurs.*

Proof. We prove the claim by induction on level. If we merge from level 0 to an array A at level 1, since an array at level 0 then becomes linked to A, the array A becomes immediately visible. Now, suppose we just merged two arrays from level k into array A at level $k+1$. At the end of the merge, level k has two arrays with items (the arrays that engaged in the merge) and one shadow array B that is linked to A. Since there are never two adjacent unsafe levels, B has its lookahead pointers completely intact before a merge into level k can occur. When a merge into level k does occur, it is into B, and the other two arrays at level k then become shadow. For another merge into level $k+1$ to occur, there must be another merge into level k after the merge into B in order to make level k unsafe. By our induction hypothesis, B becomes visible by this time, and thus so does A since B is linked to A, thus proving our claim. □

We therefore only need three arrays per level. When two arrays A, B at level k are both full, level k becomes unsafe, and we need an extra array C to keep lookahead pointers and link to the array we merge into. There then later may be a merge into C. After that, should there be another merge into level k, by Lemma 14 the array C must be already be visible by that point and we can recycle A and B as shadow arrays.

We also need to slightly modify the query algorithms from Section 6.3 to work when we have two arrays at each level. The idea is simple: we have a "main" array at each level. Should there only be one visible array at a level k, we consider that array to be the main array. Otherwise, if there are two visible arrays at level k, the one that was merged into first is the main array and the other array is the "secondary" array.

The main array contains lookahead pointers into both the main and second arrays of the next level, and the secondary array, if it exists, contains no lookahead pointers. In order to maintain that each array is exactly half-full with real items, we only use half the space of the secondary array. When merging into what will become the main array of level $k+1$, every 8th element in this main array appears as a lookahead pointer in a shadow array of level k. When merging into what will become the secondary array of level $k+1$, every 16th element of each of the main and secondary arrays of level $k+1$ appears as a lookahead pointer in the shadow array at level k. Thus, main arrays always contain exactly half real items and half lookahead pointers. Duplicate lookahead pointers at level k point to the nearest real lookahead pointers to their left and right in each of the main and secondary arrays of level $k+1$. Thus, queries are performed much like in Section 6.3, but we have to search both arrays at a level in parallel.

Chapter 7

Conclusion and Future Directions

We have presented two data structures, the buffered repository B$^\epsilon$-tree and the cache-oblivious lookahead array, that each can be used to provide efficient databases in the external-memory model. The cache-oblivious lookahead array also has the added advantage of being theoretically efficient across all adjacent levels of the memory hierarchy simultaneously.

While we have shown the BRB$^\epsilon$T to be efficient in practice, no experimental evaluation of the COLA has yet been performed. Though the COLA theoretically appears to have no large constant overhead factors in terms of I/O transfers, its space consumption may make it undesirable. The basic COLA without deamortization or lookahead pointers uses at most $2N$ space, which is better than the worst cases of both the B-tree and BRB$^\epsilon$T. Adding lookahead pointers wastes an additional factor of 2 in space, and combining deamortization with lookahead pointers again adds an additional factor of 3. These factors combine to yield a data structure that in the worst case consumes roughly $12N$ space, which might be undesirable when dealing with large databases. It thus remains to obtain a cache-oblivious search tree with fast insertions that uses an amount of space closer to that of the B-tree and BRB$^\epsilon$T.

Another direction to take from this work is to seek external-memory data structures with fast updates for other scenarios. For example, there are several known external-memory geometric data structures such as the R-tree, O-tree, cache-oblivious R-tree, and external memory kd-tree [1, 3, 18, 21]. It would be interesting to investigate whether any of these data structures can be altered to support much faster updates at small costs in query complexity. One should note that although the O-tree claims an optimal tradeoff between queries and updates, this claim holds only amongst a certain class of approaches they refer to as "non-replicating index structures".

Bibliography

[1] Pankaj K. Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Symposium on Computational Geometry*, pages 237–245, San Diego, California, June 2003.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 1988.

[3] Lars Arge, Mark de Berg, and Herman J. Haverkort. Cache-oblivious r-trees. In *Symposium on Computational Geometry*, pages 170–179, Pisa, Italy, June 2005.

[4] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[5] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2), 1966.

[6] Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming,* volume 2380 of *Lecture Notes in Computer Science*, pages 195–207, Málaga, Spain, July 2002.

[7] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[8] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th Annual*

ACM-SIAM Syposium on Discrete Algorithms, pages 29–38, San Francisco, California, January 2002.

[9] Michael A. Bender and Bradley C. Kuszmaul. Personal Communication, 2005.

[10] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554, 2003.

[11] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[12] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[13] Douglas Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2nd edition, 2001.

[15] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[16] Erik D. Demaine and Mihai Pătraşcu. Logarithmic lower bounds in the cell-probe model. *SIAM Journal of Computing*, 35(4):932–963, 2006.

[17] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[18] Antonin Guttman. A dynamic index structure for spatial searching. In *ACM SIGMOD Conference*, pages 47–57, Boston, Massachusetts, June 1984.

[19] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[20] Jia-Wei Hong and H.T. Kung. The red-blue pebble game. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Theory of Computing*, pages 326–333, Milwaukee, Wisconsin, May 1981.

[21] Kothuri Venkata Ravi Kanth and Ambuj K. Singh. Optimal dynamic range searching in non-replicating index structures. In *ICDT*, pages 257–276, Jerusalem, Israel, January 1999.

[22] Zardosht Kasheff. Cache-oblivious dynamic search trees. Master's thesis, Massachusetts Institute of Technology, June 2004.

[23] Jesper Holm Olsen and Søren Skov. Cache-oblivious algorithms in practice. Master's thesis, University of Copenhagen, December 2002.

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

[25] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[26] Naila Rahman, Richard Cole, and Rajeev Raman. Optimised predecessor data structures for internal memory. In *Proceedings of the 5th International Workshop on Algorithm Engineering,* volume 2141 of *Lecture Notes in Computer Science*, pages 67–78, Aarhus, Denmark, August 2001.

[27] Frederik Rønn. Cache-oblivious searching and sorting. Master's thesis, University of Copenhagen, July 2003.

[28] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2), 1985.

[29] Michael Tokarev. http://www.corpit.ru/mjt/qsort.html, 2004.