Cache-Oblivious Dynamic Search Trees

by

Zardosht Kashe®

Submitted to the Department of Electrical Engineering and Computer Science
in partial ful¯llment of the requirements for the degree of

Cache-Oblivious Dynamic Search Trees

by

Zardosht Kashe®

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2004, 5n partial ful¯llment of the
requirements for the degree of
Masters of Engineering 5n Computer Science and Engineering

Abstract

I have implemented a cache-oblivious dynamic search tree as an alternative to the ubiquitious B-tree. I use a binary tree with a \van Emde Boas" layout whose leaves point to 5ntervals 5n a \packed memory structure". We refer to the data structure as a COB-Tree. The COB-Tree supports e±cient lookup, as well as e±cient amortized 5nsertion and deletion. E±cient implementation of a B-tree requires understanding the cache-line size and page size and is optimized for a speci¯c memory hierarchy. In contrast, the COB-Tree conta5ns no machine-dependent variables, performs well on any memory hierarchy, and requires minimal user-level memory management. For random 5nsertion of data, my (ree.)-43(structure)-344(p)-27(erforms)-344(
vrnagoas"344(In)-653(dduitins,)3725(the)-343((ree.)643(structure)-643ise)-643easye toobusey memor

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The COB-Tree is tested on large test cases, comparing performance with the standard solution, B-trees, and analyzing asymptotic properties. One set of experiments involving disk access shows comparisons between the cache-oblivious search tree and B-trees. I compare

Chapter 2

Description

We focus on the problem of creating a data structure that supports e±cient data scans, searches, insertions, and deletions. The traditional solution, B-trees, has limitations. B-trees perform sub-optimally on machines with complex memory hierarchies and employ machine

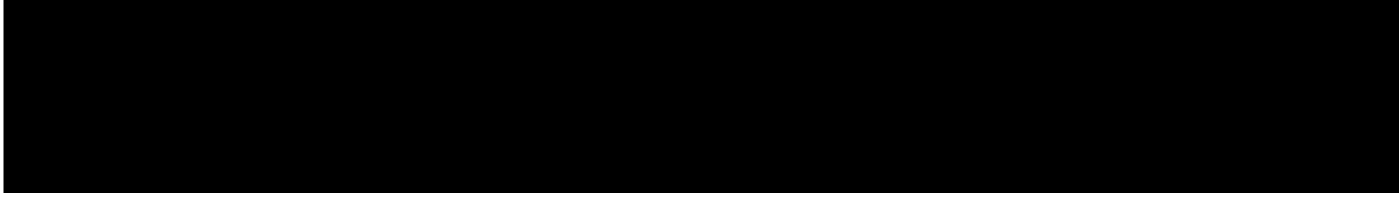


Figure 2-2: B-tree with capacity of 3 keys per node.

using a multilevel memory hierarchy, the programmer of a B-tree must decide which level of memory is the bottleneck and optimize accordingly. To program e±ciently under a multilevel memory hierarchy requires the user to consider multiple block sizes $B_1; B_2; : : : ; B_n$. To crear7s27.97Tf3o

¯rst layout the top half recursively. Then layout the remaining $2^{h=2}$ subtrees recursively in some $c > 1$. The remaining fraction of the array, $1 ¡ 1=c$, is blank. Let T be the size of the array. We specify T to be a power of 2 at all times. Divide T into equally sized sections of size $s = £(\log^2 T)$ such that s_{size}

node that is within threshold. *Rebalance*

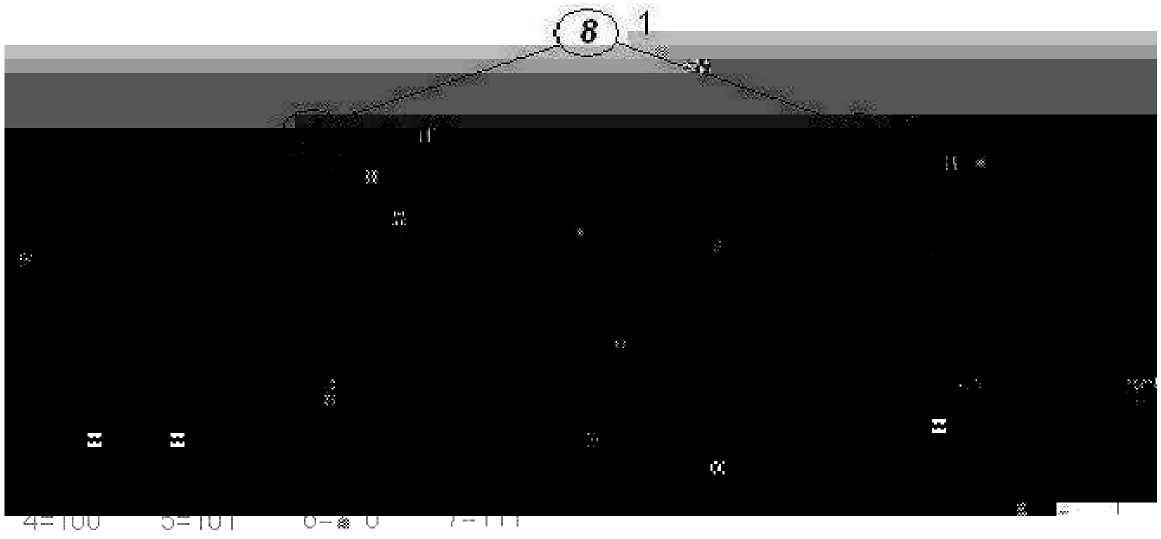


Figure 2-4: An example of the packed memory structure containing the values 1 through 16. The array contains 8 sections. The binary tree is labeled with a breadth-¯rst layout along with the bit representations of the layout. The sections are labeled below the array. The numbers in bold italics in the nodes are values held by the node.

Algorithms

Data Query Data query is simple. To search for a particular element i, ¯rst search the binary tree to ¯nd which appropriate section i belongs. To do so, we traverse a path of the tree. If i is less than or equal to the key at a node of the tree, go left. Otherwise, go right. Figure 2-4 shows an example of the packed memory structure. Once the section is found, perform a binary search within the section.

To search for a range of elements $[a; b)$, search for the element a

Chapter 3

Results

3.2.2 Insertion-at-Head Pattern

We focus on runtime. Figure 3-3 shows the average time for inserting elements using the insertion-at-head pattern. We don't see the same dips and sharp increases as Figure 3-1 because rebalances occur much more frequently. Figure 3-4 shows the average runtime for the insertion-at-head pattern normalized by dividing by $\lg^2$

Figure 3-2: (Average time for insertion)/(lg^2

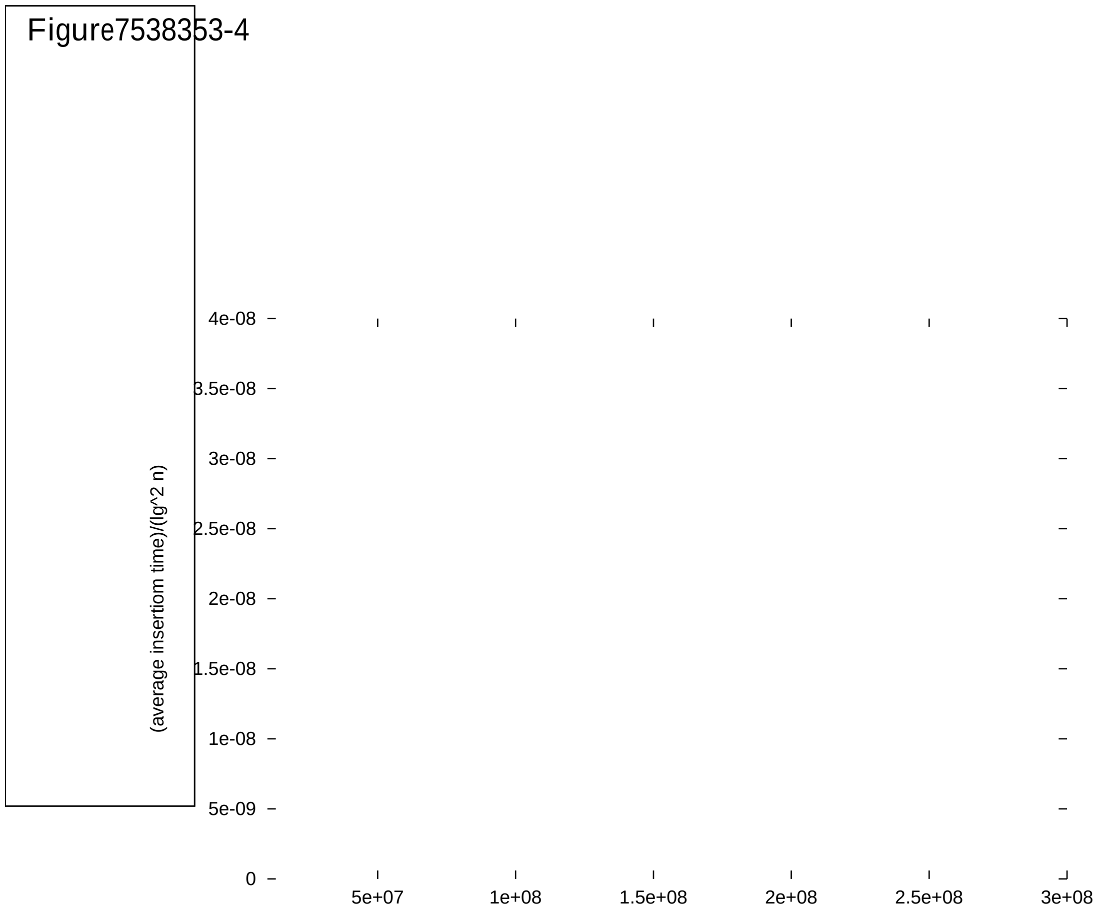


Figure 3-4: (Average time for insertion)/5aaFs97

600
550
500
450
400
350
300
250
200
150
100

Figure 3-10 plot((Average of balancing sum)/(number of elements)) pattern with insertion-at-head

Chapter 4

Static Cache-Oblivious Binary Tree Implementation

Chapter 2 described the two data structures that form the COB-Tree, a static cache-oblivious binary tree with a van Emde Boas layout, and a packed memory structure. Chapter 3 provided experimental results. This chapter presents implementation details of a tree with a van Emde Boas layout. Chapter 5 presents implementation details of the packed memory structure.

The tree is represented in memory as an array. The value at location *i* of the array corresponds to some node of the tree. We need a way of computing the location of the left and right children of node *i*. One solution is to have the array store pointers, but pointers cost space. Instead, we wish to have an array such that the root of the tree is the ¯rst element of the array, and for a given node located at array location *i*, the locations of the node's two children are easily found. This chapter provides details.

Consider the breadth-¯rst layout, a simple tree layout. In a breadth-¯rst layout, a binary tree of *N* nodes is represented as an array. Each element of the array corresponds to a node. The values held in the array are values of nodes. The root node is located at the ¯rst poisition of the array. The arrayo child66 node breadth-¯rst layout simple to use and conserves space by not allocating pointers for children of nodes. Figure 4-1 shows an example of a binary tree with breadth-¯rst indices, along with van Emde Boas indices.

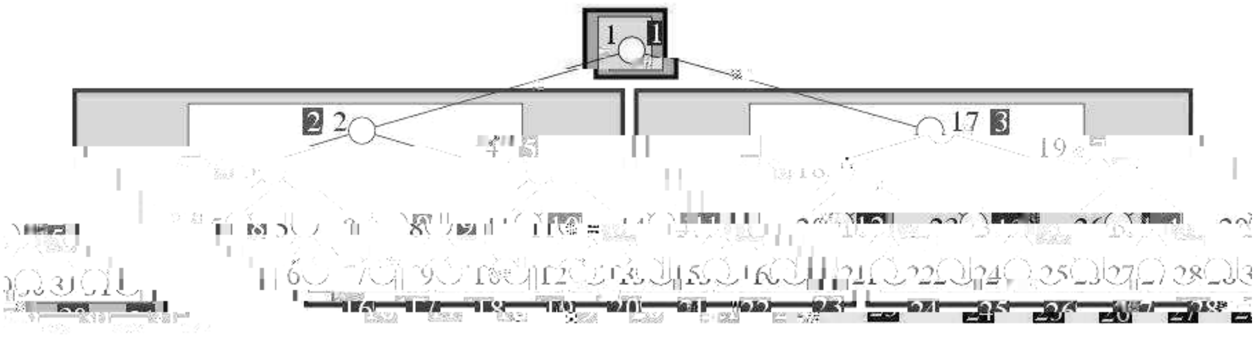


Figure 4-1

a binary search on a tree with a breadth-¯rst layout. The variables, `depth` and `height`, are

two bits gives us the root of B_i to be $000\ldots0001011 = 11$. Therefore, $x = 7 + (11 - 8) \ast$

To change the ¯rst k ¡ 2 bits to 0000:::0001 may be done easily. In the bit representation of n, we know the dth bit is a 1 and all higher order bits are 0. Thus, to evaluate n^0, we need to right-shift bits up to position d to position d^0. For example, for -3516(46;349(pnd)-TJ/F511.95Tf591.70TD[

Chapter 5

Packed Memory Structure

2. The size of the array T, is a power of 2. The array is divided into sections of size S such that S is the power of 2 arithmetically closest to $\lg^2 T$. Note that $(\sqrt{2}/2) \lg^2 T \le S \le \sqrt{}$

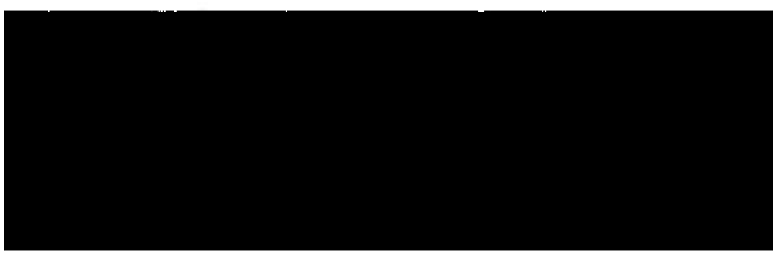
has several issues, along with several possible solutions, all of which are presented in the next section. We focus on updating the search tree. Suppose we have found and rebalanced the proper portion that will contain *i*. The subtree rooted at the node in the tree representing the rebalanced portion is no longer valid. We must update the subtree representing this portion. This can be done recursively. If the node is a leaf, return the maximum value of the represented section. Every internal node solves for two values from its children, the largest

rigt;e itgft,gonglong

eft79514[=fdatTtrebtre,igdex,g

pennbrp241Tf/lFtig9gh treevw427(ose)6345(w)26(427(ion.427(eerm(de)--2,will)-347(con)26(309.01TJ/F5

Figure 5-1: The initial state of the array before rebalancing begins. The array has four sections, each with a capacity of four elements.



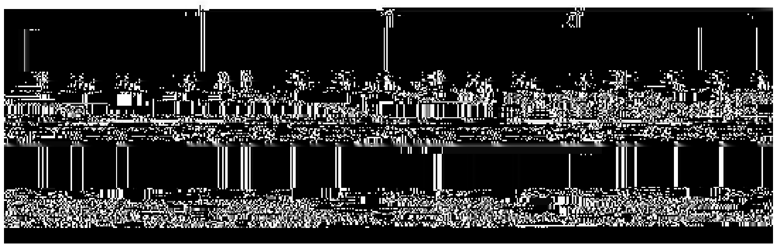


Figure 5-5: The state of the array after section 1 is crunched towards Section 2. Sections 3 and 4 remain to be crunched.

is lost. Similarly, if a and b are both left moving elements, ¯rst move a and then b. If a is a left moving element and b

elements are moved, because a and b a is a right

moving element and b

are moved, because $a < a^0 < b^0 < b$.

Chapter 6

Conclusion

be intolerable to users. Thus, one area of research is to deamortize the cost of insertion. That is, ¯nd a way to reduce the variance of the time to insert elements.

Bibliography

[1] Lars Arge and Je®rey Scott Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 560{569, Burlington, VT, October 1996.

[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Record of the 1970 ACM SICFIDET Workshop on Data Description and Access*, pages 107{141, Houston, Texas, November 1970.

[3]

Haodong Hu, John Iacono, and Alejandro L¶opez-Ortiz. The cost of cache-oblivious searching. In *Proceedings of the 44st Annual Symposium on Foundations of Computer Science*, pages 271{282, Cambridge, Massachusetts, October 2003.

[4] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scan-

[6] Michael A. Bender, Richard Cole, and Rajeev Raman. Exponential structures for ef-¯cient 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, M¶laga, 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 Gosil1.66TD[(ScB-)]TJ-4*

[14] Matteo Frigo, Charles E. Leiserso8, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285{297, New York, October 1999.

[15]

[22] V. Raman. Locality preserving dictionaries: theory and application to clustering in databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1999.

[23] Sleepycat Software. Berkeley db 4.0.14. http://www.sleepycat.com, 2002.

[24] Dan E. Willard. Maintaining dense sequential ¯les in a dynamic environment. In