Cache Oblivious Dynamic Dictionaries with Insert/Query Tradeoffs

Martin Jacobsen, 20073054

Master's Thesis, Computer Science
April 2018
Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

The field of cache-oblivious algorithms encapsulates multi level memory hierarchies for unknown memory- and block-sizes. This offers increased portability over cache-aware algorithms. The dictionary problem is a widespread subproblem to other algorithms and data structures, including the field of databases. Consequently a cache oblivious dictionary is of interest, and being able to set a tradeoff between update and query operations is a highly desirable property. The main contribution of this thesis is to implement and experimentally evaluate the first optimal cache oblivious dynamic dictionary with an update/query tradeoff [BDF+10]. We document in this thesis that for some constant $\varepsilon < \frac{1}{2}$ the structure is bounded by $\mathcal{O}(\frac{1}{\varepsilon}\log_B(\frac{N}{M})/B^{1-\varepsilon})$ and $\mathcal{O}(\frac{1}{\varepsilon}\log_B\frac{N}{M})$ I/O's for insertion and query respectively between two layers in the memory hierarchy.

Resumé

Feltet af cache-oblivious algoritmer enkapsulerer multi level hukommelses hierarkier for ukendte hukommelses- og blok-størrelser. Dette giver øget portabilitet over cache-aware algoritmer. Dictionary problemet er et udbredt under-problem til andre algoritmer og datastrukturer, inklusivt database feltet. Naturligt er en cache-oblivious dictionary dermed af interesse, og at kunne bestemme en balance mellem indættelse og søgning er en ønskværdig egenskab. Hoved bidraget af dette speciale er at implementere og experimentielt evaluere den første cache oblivious dynamiske dictionary med et tradeoff mellem opdateringer og søgninger kaldet xDict [BDF$^+$10]. Vi dokumenterer at for en konstant $\varepsilon < \frac{1}{2}$ bruger strukturen $\mathcal{O}(\frac{1}{\varepsilon}\log_B(\frac{N}{M})/B^{1-\varepsilon})$ og $\mathcal{O}(\frac{1}{\varepsilon}\log_B \frac{N}{M})$ I/O operationer for indsættelse og søgning respektivt mellem to lag i hukommelses hierarkiet.

Acknowledgments

First and foremost I would like to thank my advisor Gerth Stølting Brodal, for his advise and encouragement before and throughout the thesis. I first came to Gerth with several proposals for a thesis topic. Gerth listened and acknowledged each topic as a perfectly good thesis topic. Gerth then discarded them all collectively. Instead he suggested I take on a challenge, and for that I am grateful. This thesis has helped shape and develop me both as a computer scientist, and on a personal level.

Secondly I would like to thank Jesper Asbjørn Sindahl Nielsen, for taking me in when I had no adequate office to work in, for his warm welcome and help settling in, and for all his help and encouragement throughout. Jesper listened to every personal triumph and failure throughout the thesis.

I would like to thank Mathias Rav, for the discussions on computer science in general, and my thesis in particular, and for granting me access to and setting up the server Raleigh, on which I have performed the experiments of the thesis. Mathias time and again gave input and encouragement when I encountered difficulties in the thesis.

I would like to thank the lunch club of the algorithms and datastructures research group at Aarhus University, for the many interesting conversations and laughs we have had throughout the time of the thesis.

Finally I would like to thank my family, for their support and encouragement throughout the years, without which I would not have been able to complete my studies and write this thesis.

Martin Jacobsen,
Aarhus, April 5, 2018.

Contents

1 Introduction

The thesis is structured as follows. We first define the models of computation in Section 2, and list related work with regards to cache oblivous algorithms, particularly cache oblivious dictionaries, in Section 3, as well as the lower bounds for comparison based dictionaries. Section 4 outlines common definitions, methods and structures used throughout the rest of the thesis. Section 5 describes the theory of the three cache aware data structures implemented in the thesis, followed by a description of the main structure of the thesis in Section 6, the cache oblivious dynamic dictionary xDict. The structures listed in Sections 5 and 6 are experimentally evaluated and compared in Section 7. Section 8 concludes on the experiments performed in the previous section.

The main contribution of the thesis is the implementation and experimental evalutation (Section 7) of the xDict data structure (Section 6). We show that the the xDict is bounded by $\mathcal{O}(\frac{1}{\alpha B^{1/(1+\alpha)}} \log_B(\frac{N}{M}))$ and $\mathcal{O}(\frac{1}{\alpha} \log_B \frac{N}{M})$ I/Os for insertion and query respectively, and the tradeoff parameter $0 < \alpha \leq 1$. For some constant $\varepsilon < \frac{1}{2}$ these bounds become $\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B(\frac{N}{M}))$ and $\mathcal{O}(\frac{1}{\varepsilon} \log_B \frac{N}{M})$ I/O's. The experiments evaluating the bounds were performed for two levels of the memory hierarchy, between internal memory and the external layer of a harddisk drive.

2
Models of Computation

The Random Access Machine (RAM) model [CR72] serves as the standard model of computation for internal memory. It consists of a processor containing a constant number of registers and can access a single layer of infinite memory, see Figure 2.1. Both the registers and memory can contain indirect addresses (pointers). The measure of performance is the number of instructions performed by the CPU, including data transfers between memory and registers. However, in the real world internal memory is not infinite, and, as our society has become ever more data driven, data sets often exceeds the limits of internal memory, increasingly forcing algorithms and data structures out onto external memory. The difference in accessing memory located externally on a hard disk drive over accessing internal memory is approximately a factor million slower. The RAM model can not encapsulate this bottleneck, which has driven the development of other models of computation, namely the I/O and Cache Oblivious models.

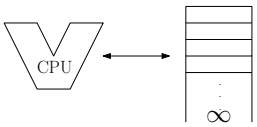


Figure 2.1: The RAM model

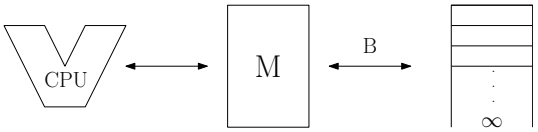


Figure 2.2: The I/O model

2.1 The I/O-Model

The I/O-model created by Aggarwal and Vitter [ASV88] models the movement of blocks of data between two layers of memory as a measure of performance, see Figure 2.2. This movement is named an I/O operation, and dependent on direction is denoted a read or write operation. The model defines the following parameters.

- N = # Elements in total
- M = # Elements that can fit in internal memory
- B = # Elements in a block

- P = # Blocks that can be transferred concurrently

We make the assumption that $N > M \geq 2B$. For the purpose of this thesis we set $P = 1$ and ignore this parameter.

Familiar RAM model upper bounds have equivalent I/O-model upper bounds, see Table 2.1 for a selection.

Operation	RAM	I/O	Reference I/O
Scan	n	$\frac{N}{B}$	
Search	$\log_2 n$	$\log_B N$	[BM72]
Sorting	$n \log_2 n$	$\frac{N}{B} \log_{M/B}(\frac{N}{B})$	[ASV88]

Table 2.1: Upper bounds in the RAM and I/O models. The RAM bound is in number of instructions and the I/O bound is in number of I/O operations.

2.2 The Cache Oblivious Model

The I/O-model is restricted to modeling the transfer of blocks between two layers of memory. In modern computers we have several layers in the shape of the CPU, cache, RAM, and external memory, see Figure 2.3. Frigo et al. [FLPR99] introduced the cache oblivious model where an algorithm or data structure is oblivious to the parameters M and B of any layer of memory. It follows that a cache oblivious algorithm or data structure, which is optimal between two layers of memory, is optimal between all layers of the memory hierarchy, under the assumptions of the ideal cache model. The ideal cache model assumes an automatic optimal cache replacement strategy with full associativity. Frigo et al. addresses these assumptions in Section 6 of [FLPR99], arguing the ideal cache model can be simulated by weaker models.

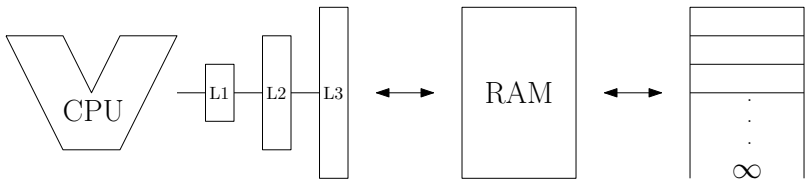


Figure 2.3: A modern computer memory hierarchy.

Algorithms in the cache oblivious model make use of the tall cache assumption $M = \Omega(B^2)$, or alternatively $M = \Omega(B^{1+\varepsilon})$ for some parameter $\varepsilon > 0$, which influences the bounds of the algorithm.

Brodal and Fagerberg [BF03b] proved that cache oblivious sorting requires the tall cache assumption, and that there does not exist an optimal cache oblivious algorithm for permuting. Brodal et al. [BFM05] proved a tight bound for adaptive sorting in the I/O and cache oblivious model of $\Theta(\frac{N}{B}(1 + \log_{M/B}(1 + \frac{Inv}{N})))$ I/O's, where Inv is the number of inversions in the input. Bender et al. [BBF+11] proved that cache oblivious searching is bounded from above and below by $\log_2(e) \log_B(N) = \Theta(\log_B(N))$ I/O's. Additional lower bounds relevant to this thesis is outlined in Section 3.1.

3
Related Work

In this section we outline related work in the cache oblivious field, especially on cache oblivious dictionaries. In Section 3.1 we describe the lower bounds on external dictionaries outlined in [BF03a], which forms the basis for the thesis.

Study of cache oblivious algorithms was initiated by Frigo et al. [FLPR99] who presented cache oblivious algorithms for matrix transposition, Fast Fourier Transformation, and sorting. Prokop [Pro99] described a cache oblivious static binary search tree, the structure of which would become known as the van Emde Boas layout. Brodal et al. [BFJ02] turned Prokops static structure into a dynamic search tree. Bender et al. [BDFC00] presented a cache oblivious dynamic B-Tree. Cache oblivious priority queues were developed by Arge et al. [ABD$^+$02] and Brodal and Fagerberg [BF02b]. Agarwal et al. [AADHM03] described cache oblivious data structures for orthogonal range searching.

For the comparison based external dictionary problem the aforementioned static binary tree of [Pro99] supports queries in $\mathcal{O}(\log_B(N))$ I/O's, in the cache oblivious model, and the dynamic trees of [BDFC00, BDIW02, BFJ02] support updates in $\mathcal{O}(\log_B(N))$, $\mathcal{O}(\log_B(N) + \log^2(\frac{N}{B}))$ and $\mathcal{O}\left(\frac{\log^2(N)}{\varepsilon \cdot B}\right)$ I/O's respectively, while maintaining the $\mathcal{O}(\log_B(N))$ I/O query bound. Bender et al. [BFCF$^+$07] described the Shuttle Tree which supports updates in $\mathcal{O}\left(\frac{\log_B(N)}{B^{\Theta(1/(\log\log B)^2)}} + \frac{\log^2(N)}{B}\right)$ I/O's while maintaining the $\mathcal{O}(\log_B(N))$ I/O query bound. An implicit cache oblivious dictionary supporting updates and query in $\mathcal{O}(\log_B(N))$ I/O's was presented by Franceschini and Grossi [FG03]. Brodal et al. [BKRT10, BKR12] built on Franceschinis and Grossis work to create an implicit cache oblivious predecessor dictionary which supports updates in $\mathcal{O}(\log_B(N))$ I/O's and search in $\mathcal{O}(\log_B \min(\ell_{p(e)}, \ell_e, \ell_{s(e)}))$ I/O's, where ℓ_e is the number of distinct elements searched for since element e was last searched for, and $p(e)$ and $s(e)$ is the predecessor and successor operations.

Brodal and Fagerberg [BF06] described a static cache oblivious string dictionary supporting prefix queries in $\mathcal{O}(\log_B(N) + \frac{|P|}{B})$ I/O's, where P is the query string.

Iacono and Pătraşcu [IP12] presented a cache aware dynamic dictionary

which departed from the indivisibility paradigm to achieve amortized $\mathcal{O}(\frac{\lambda}{B})$ I/O's update cost and $\mathcal{O}(\log_\lambda(N))$ I/O query cost, with high probability, for $\max(\log\log N, \log_{M/B}(\frac{N}{B})) \leq \lambda \leq B$. For $\lambda = B^\varepsilon$ this is a $\frac{1}{B^{1-\varepsilon}\log_B(N)}$ factor faster than a comparison based B-Tree, while maintaining its optimal query bound. The structure was inspired by the xDict [BDF+10], the main subject of this thesis.

Data Structure	Query	Update
Static Prokop [Pro99]	$\mathcal{O}(\log_B(N))$	N/A
Dynamic B-Tree [BDFC00]	$\mathcal{O}(\log_B(N))$	$\mathcal{O}(\log_B(N))$
Flat Implicit Tree [FG03]	$\mathcal{O}(\log_B(N))$	$\mathcal{O}(\log_B(N))$
Shuttle Tree [BFCF+07]	$\mathcal{O}(\log_B(N))$	$\mathcal{O}\left(\frac{\log_B(N)}{B^{\Theta(1/(\log\log B)^2)}} + \frac{\log^2(N)}{B}\right)$
Lower Bound [BF03a]	$\mathcal{O}(\frac{1}{\varepsilon}\log_B(\frac{N}{M}))$	$\Longrightarrow \Omega(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B(\frac{N}{M}))$
xDict [BDF+10]	$\mathcal{O}(\frac{1}{\varepsilon}\log_B(\frac{N}{M}))$	$\mathcal{O}(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B(\frac{N}{M}))$

Table 3.1: Cache Oblivious Dictionaries. Inspired by Table 1 in [BDF+10]

3.1 Lower Bounds

Brodal and Fagerberg [BF03a] determined lower bounds for external comparison based dictionaries, summarized in the following theorem.

Theorem 3.1. *If N insertions performs at most $\delta \cdot \frac{N}{B}$ I/O's, then*

1. *There exists a query requiring at least $\log_{B+1}(\frac{N}{M}) - \mathcal{O}(1)$ I/O's.*
2. *There exists a query requiring at least $N/(M \cdot (\frac{M}{B})^{\mathcal{O}(\delta)})$ I/O's for $N > M$.*
3. *There exists a query requiring $\Omega\left(\log_{\delta\cdot\log^2(N)}(\frac{N}{M})\right)$ I/O's, if $\delta \leq B/\log^3(N)$ and $N > M^2$.*

4
Preliminaries

In this section I outline several common data structures and techniques used in, or forming the basis for, the main data structures of the thesis.

4.1 Amortization

Tarjan [Tar85] defined the term "amortization" in a computational complexity context as "to average the running times of operations in a sequence over the sequence."

When we reason about running times summing the worst case upper bounds of the sequence of operations can be too pessimistic, as some operations on the data structure might benefit future operations or apply a future expense to an operation. Conversely simply averaging the cost of all functions over all elements might mis-estimate these benefits or expenses. We can utilize amortized analysis technique to reason about the average running times taking into account the shared cost of operations. Tarjan outlines two different yet equivalent views of amortization: That of the banker and that of the physicist.

The bankers view models our machines to run on coins. We assign a fixed amount of coins to spend on each operation in the sequence. We then either deposit to or withdraw from an account to pay for the execution of each operation. We are allowed to temporarily indebt ourselves, but at the end of the sequence of operations we must be debt free, in order to show that the initial allocation of coins was sufficient.

The physicists view handles amortized analysis using potential functions. The state of our data structure D is evaluated by a potential function $\Phi(D)$ into a real number we call its potential. The amortized time a of an operation then becomes $t + \Phi(D) - \Phi(D')$ where t denotes the real time of the operation, D the data structures state before the operation and D' the data structures state after the operation. The total real time of a sequence of n operations becomes:

$$\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} (a_i + \Phi_{i-1} - \Phi_i) = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i$$

4.2 Dictionary

In this thesis we base the definition of a dictionary on the definition given in [AH74]. They define a set of operations, namely insertion, deletion and member query, and name any structure that supports these operations on a set of data a dictionary. We extend the definition by naming a structure that supports insertion and member query a dynamic dictionary, with the contrasting static dictionary being a data structure that merely supports member queries. In general a data structure that supports the base operations, as well as an additional operation, on a set of data, is named after the additional operation, e.g. a predecessor dictionary supports the base operations and the predecessor operation. Furthermore we define a comparison based dictionary to be a dictionary that orders elements by comparison, as opposed to e.g. hashing. An element inserted into a dictionary consists of tuples, in this thesis restricted to {key,value} or {key,value,time}, with time being the relative order of insertion.

4.3 B-Tree

A B-Tree [BM72] is a comparison based search tree optimized for external memory. It is a specialized version of the (a,b)-tree [MH82] with a leaf parameter k. Formally we define a B-Tree as follows.

Definition 4.1. If $\mathcal{T}$ is a B-Tree with branching parameter b and leaf parameter k then

- Except for the root all nodes in $\mathcal{T}$ has degree between $\frac{1}{4}b$ and b.
- The root has degree between 2 and b.
- Each node stores keys that partitions the children of the node according to the range of the elements stored below them.
- All leaves are on the same level and contain between $\frac{1}{4}k$ and k elements.

Normally when we refer to a B-Tree in an external context it is implied that $k = \Theta(B)$, with the implication that insertion and deletion, collectively called updates, can be performed in $\mathcal{O}(\log_B(\frac{N}{B}))$ I/O's for $b = \Theta(B)$, disregarding balancing operations.

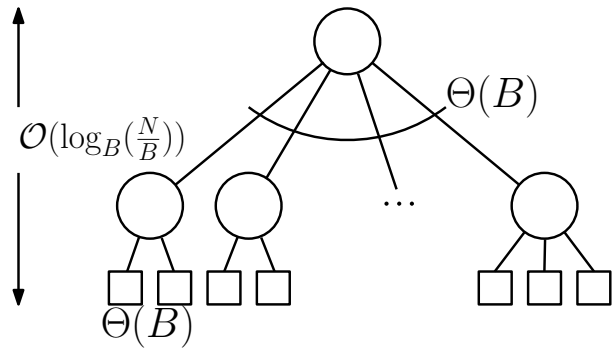


Figure 4.1: A B-Tree with $b = \Theta(B)$ and $k = \Theta(B)$

We split a node by adding a newly created node to the parent. The new node receives half the children of the original node. Splitting a leaf is handled similarly. Adding a node to the parent can cause the splitting to propagate up the height of the tree.

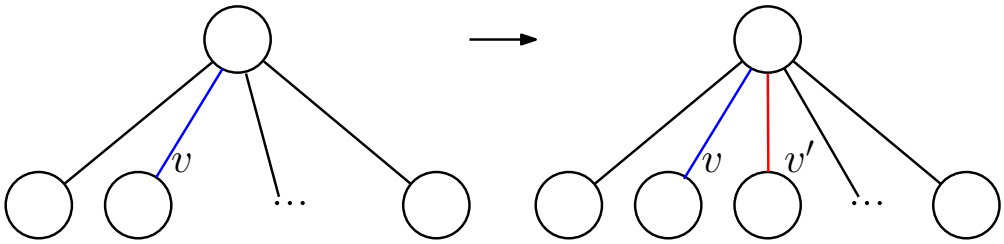


Figure 4.2: Splitting node v into node v and v'

We fuse a node by merging it with its smallest neighbor, possibly followed by a split. Fusing a leaf is handled similarly. The parent is then updated accordingly. Fusing can propagate up the height of the tree.

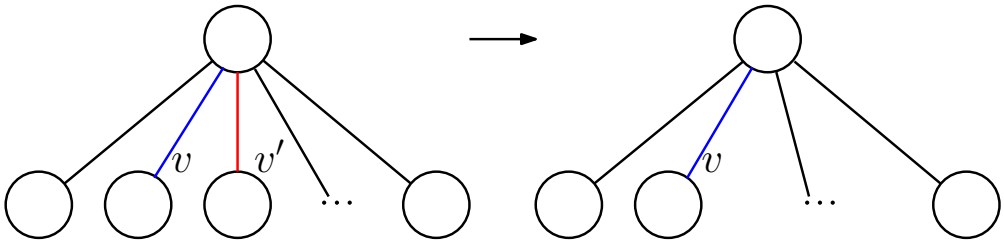


Figure 4.3: Fusing node v and v' into node v

For an (a,b)-tree with $b \geq 2a$ the amortized cost of rebalancing operations is $\mathcal{O}(\frac{1}{a})$ for each time we add or remove a leaf, see Theorem 1 [MH82]. Since a B-Tree is a specialized version of an (a,b)-tree it inherits this property, with the cost of internal rebalancing operations becoming $\mathcal{O}(\frac{1}{b})$. Furthermore it takes amortized $\mathcal{O}(k)$ updates on a leaf to produce a split or fuse of a leaf. The amortized cost of rebalancing is thus $\mathcal{O}(\frac{1}{b \cdot k})$ pr. update, and is dominated by the cost of searching for the leaf to apply the update to, see below. Therefore the amortized cost of an update in a B-Tree is $\mathcal{O}(\log_B(N))$ I/O's.

Online queries in a B-Tree with $b = \Theta(B)$ and $k = \Theta(B)$ can be performed by a top down search in the tree in $\mathcal{O}(\log_b(\frac{N}{k})) = \mathcal{O}(\log_B(N)$ I/O's.

4.4 Buffer Tree

The Buffer Tree [Arg95] augments a B-Tree with buffers, introducing a "laziness" to updates where we sacrifice handling updates immediately in return for being able to handle multiple updates pr. I/O, allowing us to amortize the cost of pushing a single update down the height of the tree. Formally a Buffer Tree is defined as follows.

Definition 4.2. A tree $\mathcal{T}$ is a Buffer Tree if

- $\mathcal{T}$ is a B-Tree with branching parameter $\frac{M}{B}$ and leaf parameter B.
- Each node is extended with a buffer of size M.

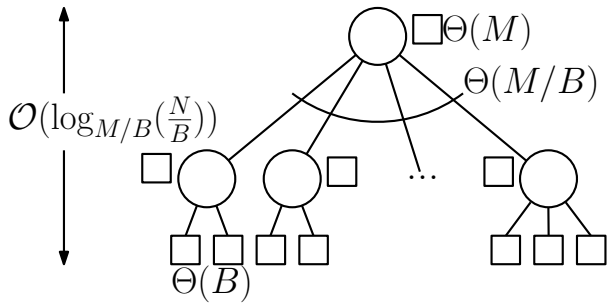


Figure 4.4: A Buffer Tree

When we introduce a new update to the tree we timestamp it and add it to a collection of $\Theta(B)$ updates in internal memory. Once we have gathered a full block of updates we append it to the roots buffer.

Flushing an internal node occurs when its buffer overflows, i.e. the number of elements exceeds M. The unsorted elements are loaded into memory and sorted while we resolve updates with duplicate keys using the timestamps. Then the updates are distributed to the node's children by appending to their buffers. Appending a non-full block to a child requires $\mathcal{O}(1)$ I/O's and can happen at most $\mathcal{O}(\frac{M}{B})$ times, resulting in a total cost of $\mathcal{O}(\frac{M}{B})$ for the entire flush. After completing the flush any buffer overflows of its children is handled recursively, with the exception of leaf nodes which are handled after we complete flushing all relevant internal nodes. Thus when we handle the flushing of a leaf node the buffers on the path to the root are empty.

Recursively flushing buffers might result in a buffer exceeding $\mathcal{O}(M)$ as one node can potentially receive all the updates from its ancestors buffers. We can handle a buffer of size $X > M$ in $\mathcal{O}(X/B)$ I/O's, noting that at most $\mathcal{O}(M)$ updates in the buffer can be unsorted, since we append a buffer in sorted order. We load in these unsorted updates, sort them while resolving duplicates, and merge them with the remaining updates. Distributing the updates to the children of the node is no different than above, and the process uses $\mathcal{O}(\frac{X}{B})$ I/O's.

Flushing a leaf node is handled differently than an internal node. We sort a buffer of size $X \geq M$ using $\mathcal{O}(X/B)$ I/O's as above. We then gather the elements from the nodes leaves in a single scan, resulting in a set of new updates and a set of existing elements, both sorted. We then write out new leaves while merging the two sets, resolving any duplicates, using $\mathcal{O}(\frac{X}{B})$ I/O's.

Splitting a node becomes necessary if at any point during the creation of new leaves the node exceeds its maximum degree. We perform the split like a normal B-Tree, noting that the buffers on the root to leaf path is empty, continuing the creation of new leaves after the split is completed.

Fusing a node becomes necessary if the creation of new leaves results in the node falling short of its minimum degree. We pad the node with dummy-leaves and then repeatedly remove one dummy-leaf at a time, handling any resulting fuses as we would in a normal B-Tree, with the exception that we have to flush the neighbors of a node before performing a fuse. This is necessary as a fuse can be followed by a split, which requires an empty buffer. Flushing a nodes neighbors can result in additional recursive fuses. The dummy-leafs prevent these recursive fuses from interfering with each other. We continue the process until no dummy-leaves remains.

The choice of branching and leaf parameters ensures that the amortized number of rebalancing operations is $\mathcal{O}(\frac{1}{M})$ pr. update, as argued in Section 4.3. Prior to fusing we flush the buffers of two children, which carries a cost of $\mathcal{O}(\frac{M}{B})$ I/O's. The cost of rebalancing is therefore dominated by the cost of flushing an element down the height of the tree and into a leaf, namely amortized $\mathcal{O}(\frac{1}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/O's pr. update.

To perform an online query in a Buffer Tree we scan each buffer on a root to leaf path before scanning the leaf requiring $\mathcal{O}(\frac{M}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/O's.

4.5 Cache Oblivious Techniques

To obtain the same bounds in the cache oblivious model as in the I/O model, the technique of recursion, where you divide the sub-problem or -structure into recursively smaller problems or structures, is generally employed. Exemplified best by the almost pervasive van Emde Boas layout [Pro99], in this section we explore a more complicated example, comparing a non-recursive algorithm, external merge sort, optimal in the I/O model, to that of the cache oblivious and recursive Funnelsort, more specifically a variant named Lazy Funnelsort. The lower bound for sorting in both the I/O and Cache Oblivious models is $\Omega(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$, see Table 2.1. The same techniques employed in this section will be utilized in Section 6, where the main structure of the thesis is described and analyzed.

4.5.1 External Merge Sort

External merge sort [ASV88] splits the input into blocks of size $\mathcal{O}(M)$ sorted elements and merge $\mathcal{O}(\frac{M}{B})$ lists at a time, keeping $\mathcal{O}(B)$ elements from each list in memory during the merge, until only one sorted list remains. This creates a merge tree of height $\mathcal{O}(\log_{M/B}(\frac{N}{M}))$ where at each level all the elements are scanned at a cost of $\mathcal{O}(\frac{N}{B})$ pr. level.

4.5.2 Lazy Funnelsort

Funnelsort [FLPR99] is a cache oblivious generalization of external merge sort. Funnelsort recursively splits the input into $N^{1/3}$ arrays of size $N^{2/3}$ after which the arrays are merged using a $N^{1/3}$-way merger. A k-way merger takes as input k sorted arrays and outputs k^3 elements in sorted order. A k-way merger is recursively constructed by merging the output of $\sqrt{k}$-way mergers. Invoking a

k-way merger will recursively fill up all the output buffers in the merger. Lazy Funnelsort [BF02a] changes the k-way merger of Funnelsort into a binary tree with an output buffer of size k^d, for $d > 1$. Each edge in the binary tree is a buffer, whose size is defined recursively: Split the height of the k-way merger in two, i.e $\log(k)/2$, then it consists of one top tree and $\sqrt{k}$ bottom trees of size $\sqrt{k}$. The buffers connecting the trees is given size $k^{d/2}$. These buffers are only refilled when needed, hence the lazy term in the name. Lazy Funnelsort recursively creates $N^{1/d}$ arrays of size $N^{1-1/d}$ and merges them using a $N^{1/d}$-way merger.

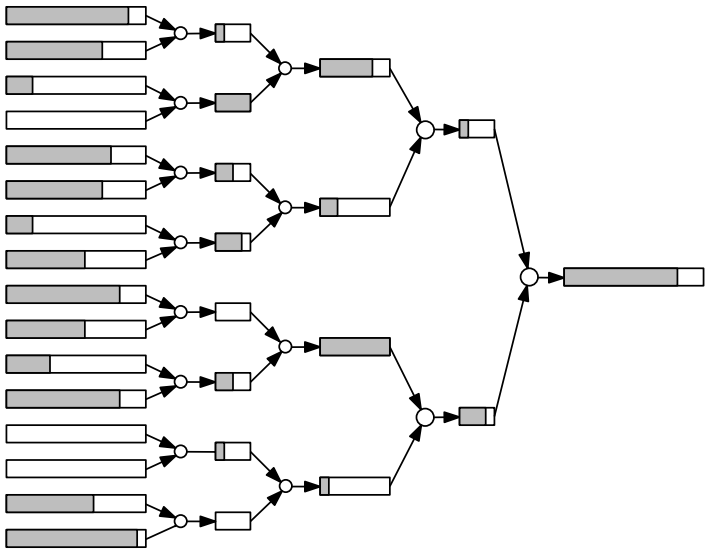


Figure 4.5: A 16-way merger in Lazy Funnelsort. Grey areas denote elements in buffers. Inspired by Figure 4 in [BFV08]

To analyze the k-way merger we find the size of the structure, and determine when it fits inside memory, i.e. any further recursion on the structure becomes free. Finding this base tree (case) is universal in analyzing any cache oblivious structure.

The k-way merger is structured as a van Emde Boas layout. Splitting the k-way merger in the middle of its height we have one top tree and $\sqrt{k}$ bottom trees. The buffers connecting this level consists of $\sqrt{k}$ buffers of size $k^{d/2}$. The recursion thus becomes $S(k) = \sqrt{k} \cdot k^{d/2} + (\sqrt{k}+1) \cdot S(\sqrt{k})$, which by the Master Theorem [CLRS09] case 3 is dominated by the size of the buffers and has the solution $\mathcal{O}(k^{(d+1)/2})$. Thus for $k^{(d+1)/2} \leq M/2c \implies k \leq (M/2c)^{2/(d+1)}$ this base tree can be loaded entirely into memory. With a tall cache assumption of $B^{(d+1)/(d-1)} \leq M/2c \implies B \leq (M/2c)^{(d-1)/(d+1)}$ we can load in one block from each of the inputs to the base tree because $k \cdot B \leq (M/2c)^{2/(d+1)} \cdot (M/2c)^{(d-1)/(d+1)} = (M/2c)^{(d+1)/(d+1)} = M/2c$. The k-way merger whose subtrees consists of such base trees has size $(k^2)^{(d+1)/2} = k^{d+1} > M/2c$. The buffers separating base trees within this merger we name large buffers. We will analyze the cost of merging around these base trees and the large buffers separating them.

Consider what happens when we invoke the merger on a base tree of size k. The cost of reading in the base tree becomes the cost scanning in the base tree and reading a single block from each of the k input buffers, incurring a cost of $\mathcal{O}(\frac{k^{(d+1)/2}}{B} + k)$. The parent merger of the base tree is larger than memory and this implies $k^{d+1} > M/2c \implies k > (M/2c)^{1/(d+1)} \implies k^{d-1} > (M/2c)^{(d-1)/(d+1)} \geq B$. This in turn implies $k = \frac{k^d}{k^{d-1}} \leq \frac{k^d}{B}$, and thus the cost of loading in the base tree can be charged to the output at $\mathcal{O}(\frac{1}{B})$I/O pr. element. If the operation on the base tree causes the input to run out we have to load in the structures beneath. This may push the base tree out of memory, and require a reload after we complete the operations on the input structures. If this happens then we will have output k^d elements already, and we simply charge the cost to these at $\mathcal{O}(\frac{1}{B})$I/O pr. element.

Let $F = (M/2c)^{1/(d+1)}$ be the minimum number of leaves in a base tree in a k-way merger, then we will insert each element in at most $\log_F(k) = \mathcal{O}(d \log_M(k)) = \mathcal{O}(\log_M(k^d))$ large buffers, for which we charge $\mathcal{O}(\frac{1}{B})$I/O pr. element. Thus the cost for outputting k^d elements is $\mathcal{O}(\frac{k^d}{B} \log_M(k^d)))$.

Recall that Lazy Funnelsort recursively merges $N^{1/d}$ arrays of size $N^{1-1/d}$. If we view this as a recursion tree then one element follows a single path up the height of the tree. This gives a pr. element cost of
$\mathcal{O}\left(\frac{1}{B} \sum_{i=0}^{\infty} \log_M N^{(1-1/d)^i}\right) = \mathcal{O}\left(\frac{1}{B} \log_M(N) \sum_{i=0}^{\infty} (1 - (1/d)^i\right) = \mathcal{O}\left(\frac{d}{B} \log_M(N)\right)$.
Note that $\mathcal{O}(\frac{d \cdot N}{B} \log_M(N)) = \mathcal{O}(\frac{N}{B} \log_{M/B}(\frac{N}{B}))$ under the tall cache assumption.

The authors utilize Lazy Funnelsort to solve several geometric problems. While Lazy Funnelsort and the geometric problems is certainly of independent interest, for this thesis it is the techniques used in the construction and analysis above which is of relevance. Comparing Lazy Funnelsort to external merge sort we notice how the basic idea of divide and conquer is used to solve the same problem in both models, but the cache oblivious structure needs recursion to adapt to the unknown M and B. The van Emde Boas Layout (although noted in [BFV08] to be unnecessary) and general recursive structure is universal to cache oblivious structures, and the analysis, where we find a base case at which the recursive structure either fits in M or B, using a tall cache assumption, is universal to the analysis of said cache oblivious structures.

5
Cache Aware Data Structures

In this section we describe the theory of the three cache aware dictionaries implemented in the thesis. Practical considerations are described in subsections to each structure. The data structures were first described by Brodal and Fagerberg [BF03a] to provide upper bounds proving the lower bounds of Theorem 3.1 asymptotically tight. The data structures all use $\mathcal{O}(\frac{N}{B})$ blocks of space.

5.1 Modified B-Tree

A Modified B-Tree is a B-Tree $b, k = B$, with b being the branching parameter and k the leaf parameter, where we keep the topmost $\mathcal{O}(\frac{M}{B})$ nodes in internal memory.

Theorem 5.1. *Updating a Modified B-Tree requires amortized* $\mathcal{O}(\log_B \frac{N}{M})$ *I/O's pr. update.*

Proof. We inherit from a normal B-Tree the property that rebalancing is amortized to less than the cost of traversing the height of the tree, see Section 4.3. The external height of the tree is $\mathcal{O}(\log_B \frac{N}{M})$ by keeping the topmost nodes in internal memory. □

Theorem 5.2. *An online query in a Modified B-Tree requires* $\mathcal{O}(\log_B \frac{N}{M})$ *I/O's.*

Proof. The proof is the same as for Theorem 5.1. □

5.1.1 Implementation

Internalizing and externalizing nodes is expensive and we can not afford to perform these functions with every update by alternating between insertion and deletion. We select a minimum and a maximum number of nodes to keep in internal memory. The maximum number is $max = \mathcal{O}(\frac{M}{B})$ and the minimum number is $min = \frac{max}{B}$. That is we have a range between min and max of one level of the tree. It follows that upon reaching max number of internal nodes we externalize $\mathcal{O}(max - min)$ nodes from the bottom internal level of the tree, and that a further $\mathcal{O}(max - min)$ splits in the internal part of the tree is required before we perform the externalization again. The same logic applies to fuses

with regards to internalization of nodes, but not to an intermixing of splits and fuses.

Splits and fuses are normally performed bottom up in a B-Tree. However, handling only single updates at a time we can anticipate and perform splits and fuses as we traverse the tree top down. If an update could result in a future split below a node, and the node is at capacity, we split it, and likewise fuse any node that can not accommodate a fuse below it.

Using external merge sort [ASV88] we can construct a Modified B-Tree of size N bottom up in $\mathcal{O}(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ I/O's, which is a $\mathcal{O}(B\log_B(\frac{M}{B}))$ factor faster than the naive approach of inserting N elements into an initially empty Modified B-Tree.

5.2 Buffered B-Tree

The Buffered B-Tree is a B-Tree with branching parameter $b = \Theta(\delta/\log N)$ for $\log N < \delta \leq B\log N$ and leaf parameter $k = B$. The tree will have height $\mathcal{O}(\log_{\delta/\log N}(\frac{N}{B}))$. Similar to a Buffer Tree each node is augmented with a buffer of size B, and similar to a Modified B-Tree the topmost $\mathcal{O}(\frac{M}{B})$ nodes, and their buffers, can be kept in internal memory.

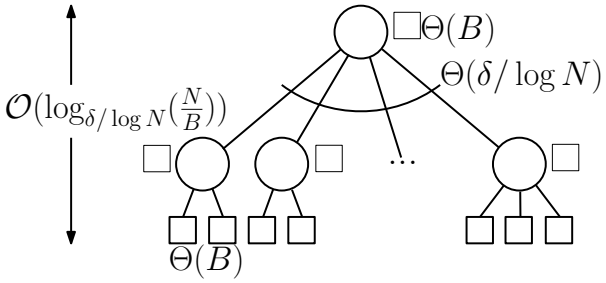


Figure 5.1: A Buffered B-Tree

When a buffer overflows there exists a child such that $\Omega((B\log N)/\delta)$ elements can be moved to the child's buffer. We push elements to all children meeting this threshold. The remaining elements stays in the buffer. Rebalancing operations necessary to maintain the tree structure after a buffer overflow is performed as we would in a Buffer Tree, with the addition that buffers is not guaranteed to be empty on the root to leaf path. However, since a buffer is of size B it is no more costly to handle each buffer than the corresponding node.

Theorem 5.3. *Updating a Buffered B-Tree requires amortized $\mathcal{O}(\delta \cdot \frac{N}{B})$ I/O's for N updates.*

Proof. N updates will use at most $\mathcal{O}(N/((B\log N)/\delta))$ I/O's at each of the at most $\mathcal{O}(\log N)$ levels of the tree, resulting in an amortized cost of $\mathcal{O}(\log(N) \cdot \delta \cdot N/(B\log N)) = \mathcal{O}(\delta \cdot \frac{N}{B})$ I/O's.

We inherit from B-Trees that we cause amortized $\mathcal{O}(\frac{\log N}{\delta \cdot B}) = \mathcal{O}(\frac{1}{B})$ rebalancing operations at a cost of $\mathcal{O}(1)$ I/O pr. update, see Section 4.3, which is dominated by the cost of handling the buffers. □

Theorem 5.4. *An online query in a Buffered B-Tree requires $\mathcal{O}(\log_\delta(\frac{N}{M}))$ I/O's given $\delta \geq \log^{1+\epsilon} N$, for any constant $\varepsilon > 0$.*

Proof. We search down the height of the tree, scanning each buffer on the root to leaf path. Keeping the topmost $\mathcal{O}(\frac{M}{B})$ nodes in internal memory reduces the external height to $\mathcal{O}(\log_{\delta/\log N}(\frac{N}{M})) = \mathcal{O}(\log_\delta(\frac{N}{M}))$ given $\delta \geq \log^{1+\varepsilon} N$, for any constant $\varepsilon > 0$. □

Note that for $\delta = B^\varepsilon \log N$ the tree has degree $\mathcal{O}(B^\varepsilon)$. The height becomes $\mathcal{O}(\log_{B^\varepsilon} \frac{N}{B}) = \mathcal{O}(\log_B \frac{N}{B})/(\log_B B^\varepsilon) = \mathcal{O}(\frac{1}{\varepsilon} \log_B \frac{N}{B})$ and the external height becomes $\mathcal{O}(\frac{1}{\varepsilon} \log_B \frac{N}{M})$. Inserting B updates causes at most $\mathcal{O}(B/((B \log N)/B^\varepsilon \log N))$ $= \mathcal{O}(B^\varepsilon B \log N/(B \log N)) = \mathcal{O}(B^\varepsilon)$ buffer overflows at each level of the tree resulting in a cost of $\mathcal{O}(\frac{B^\varepsilon}{\varepsilon} \log_B(\frac{N}{M}))$ I/O's. Online queries costs $\mathcal{O}(\frac{1}{\varepsilon} \log_B \frac{N}{M}))$ I/O's. We thereby gain faster updates compared to a normal B-Tree but maintain a query time within a constant factor.

5.2.1 Implementation

Handling a leaf node buffer overflow, similar to a Buffer Tree, is theoretically done by sorting the buffer and gathering the elements from the leaves. We then merge the two sets while writing out new leaves, using dummy elements if necessary. Rebalancing is performed when we exceed the capacity of the node by one, or fall short of the minimum size. However, in practice we simply update leaves directly with splits and fuses. This can theoretically result in the node exceeding its size if all elements on a root to leaf node path was pushed to this particular leaf node. In practice however the node is unlikely to grow more than a constant size too big, and it is much faster to let the leaves split and fuse than to write out new leaves one at a time. Furthermore this approach allows us to avoid creating dummy elements.

When a query travels down the height of the tree we can carry with us any elements in the buffer at height i to the relevant child at height $i+1$. This costs us only a constant extra I/O operations. After $\mathcal{O}(N/B)$ queries we expect to have visited each of the leaves and emptied all buffers. This will speed up the following queries by a constant.

Note that if we use a stable internal sorting algorithm then for two elements with equal keys the rightmost element takes precedence, regardless of the type of element (insertion/deletion). For merges the elements being pushed down in the tree always take precedence over elements already present, regardless of the type of element. Thus we can remove timestamps, introduced in Section 4.4, from the elements in the structure.

Similar to a Modified B-Tree we can build a Buffered B-Tree of size N in $\mathcal{O}(\frac{N}{B} \log_{\frac{M}{B}}(\frac{N}{B}))$ I/O's using external merge sort.

5.3 Truncated Buffer Tree

A Truncated Buffer Tree is a normal Buffer Tree, see Section 4.4, with a single twist. The tree is given a maximum depth δ at which the sub-trees rooted at the given depth is truncated into buckets. When the buffer of a node containing such a bucket overflows it is added to the bucket as a sorted list. Each bucket will have size $\mathcal{O}(N/(\frac{M}{B})^{\delta})$.

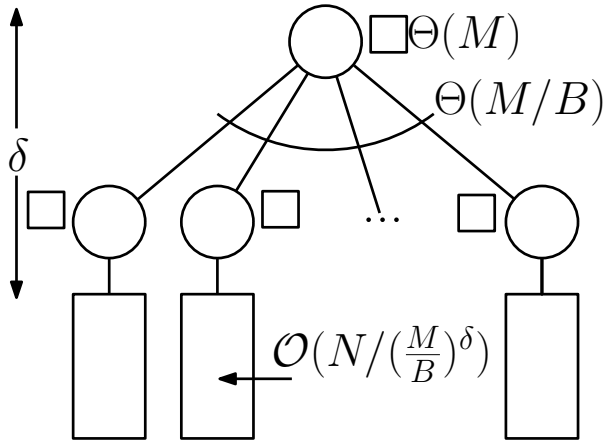


Figure 5.2: A Truncated Buffer Tree

To search the buckets we employ fractional cascading [CG86a, CG86b]. When the sorted list L_x is added to the bucket we build a new list F_x containing the elements of L_x and every second element of F_{x-1}. By induction the size of $F_x \leq 2M$. Storing two pointers, one to an elements relative position in L_x and one to the elements relative position in F_{x-1}, we can search through the lists below L_x spending $\mathcal{O}(1)$ for each list.

When a bucket overflows we find the median element in linear time [Sib99] and split the elements of the bucket around it. The fractional cascading structures is then rebuilt in linear time.

Theorem 5.5. *Inserting into a Truncated Buffer Tree requires amortized $\mathcal{O}(\delta \cdot \frac{N}{B})$ I/O's for N updates.*

Proof. We spend amortized $\mathcal{O}(\frac{1}{B})$ on each level of the tree pushing one update down to a bucket. We insert into a bucket in linear time of the buffer size, and if necessary split in linear time of the bucket size. Rebalancing operations can propagate upwards in the tree, which was shown in Section 4.4 for a Buffer Tree to be no more than the cost of pushing updates into the leafs, and even less for a Truncated Buffer Tree with its expanded leaf parameter. □

Theorem 5.6. *An online query in a Truncated Buffer Tree requires $\mathcal{O}(N/(M(\frac{M}{B})^{\Omega(\delta)}))$ I/O's for some constant $c > 1$ and $N \geq M \cdot (\frac{M}{B})^{c \cdot \delta}$.*

Proof. We scan the buffers at each level of the tree spending $\mathcal{O}(\delta \cdot \frac{M}{B})$ I/O's. We then search a bucket by scanning the top fractional cascading list, and search through the remaining lists spending $\mathcal{O}(1)$ I/O pr. list. This gives a total cost

of $\mathcal{O}(\delta \cdot \frac{M}{B} + N/(M(\frac{M}{B})^{\Omega(\delta)}))$. For a large enough N scanning the buckets will intuitively dominate searching the buffers. We can express this as the following set of equations.

$$\begin{aligned} N/(M(\tfrac{M}{B})^{c_1\delta}) &\geq \delta \cdot (\tfrac{M}{B}) && c_1 > 0 \\ N &\geq \delta \cdot (\tfrac{M}{B}) \cdot M \cdot (\tfrac{M}{B})^{c_1\delta} \end{aligned}$$

However,

$$M \cdot (\tfrac{M}{B})^{c_2\delta} \geq \delta \cdot (\tfrac{M}{B}) \cdot M \cdot (\tfrac{M}{B})^{c_1\delta} \qquad \text{for } c_2 \geq c_1 + 1$$

Thus for some $c > 1$ and $N \geq M \cdot (\frac{M}{B})^{c\delta}$ searching the buckets will dominate the cost of the query. □

Deletions can generally not be handled by a Truncated Buffer Tree in the same time as insertions, given that searching for the element to be deleted is too costly. Nor can we afford to sort the insertions and deletions in the given time. Therefore we can not use the technique of global rebuilding, where we mark the elements for deletion and rebuild the tree every time we double the number of updates.

5.3.1 Implementation

For all practical N, M and B the height of the tree can never exceed three. As we flush buffers on a root to leaf path all elements from the buffers on this path may end up in a single buffer, causing it to exceed size M. Upon flushing a leaf node buffer we do not split the buffer into potentially several lists of size M, knowing the buffer has size $\mathcal{O}(M)$.

Splitting a bucket is theoretically done by finding a median element to split the bucket around. This can be done in linear time [Sib99]. However, using the assumption that $M \geq \frac{N}{M}$ for practical N and M we can read the median of each list in the bucket into memory. We then select the median of medians. Half the medians will be smaller than the median of medians, and a quarter of the total elements in the bucket will be guaranteed to be smaller than these medians. The same argument can be made for medians and elements greater than the median of medians. We can therefore guarantee a 25/75 split using the median of medians.

A query can afford to scan the entire top fractional list of a bucket, but performing a binary search on the list is potentially faster, depending on the size of the list. Note that searching backwards in the list means the harddisk physically has to revolve the disk nearly a complete turn. In practice this doubles the cost of looking up a memory block located further back in the list. Therefore the list has to have a large enough size for the binary search to offset this additional cost. For $M = 8$MB and $B = 128$KB we save on average 50% I/O's searching the top list with binary search over doing a linear scan.

Similar to the implementation of Buffered B-Trees, see Section 5.2.1, we can remove the timestamps under the assumption of a stable internal sorting algorithm.

6
xDict

In this section we describe a cache oblivious dynamic dictionary with update/query tradeoff known as the xDict [BDF+10]. First we describe the underlying structure of the x-box in Section 6.1. Next we describe how we query an x-box in Section 6.2, and how an x-box supports batched insertions in Section 6.3. Using layers of x-boxes doubling exponentially in size we build an xDict in Section 6.4, supporting insertions in $\mathcal{O}(\frac{1}{\varepsilon}\log_B(\frac{N}{M})/B^{1-\varepsilon})$ I/O's pr. element, and queries in $\mathcal{O}(\frac{1}{\varepsilon}\log_B \frac{N}{M})$ I/O's, for some $\varepsilon > 0$. Implementation details are described in Section 6.5.

6.1 The x-box

The x-box is a recursive structure defined for a given parameter x and a global constant $1 \geq \alpha > 0$, with α determining the tradeoff between insertion and query for the xDict, see Section 6.4. The x-box uses $\mathcal{O}(x^{1+\alpha})$ addressing space to store up to $\frac{1}{2}x^{1+\alpha}$ real elements. We SEARCH an x-box in $\mathcal{O}((1+\alpha)\log_B x)$ I/O's and BATCH-INSERT $\frac{1}{2}x$ elements into an x-box using $\mathcal{O}((1+\alpha)\log_B(x)/B^{1/(1+\alpha)})$ I/O's pr. element. Thus lower values of α results in faster operations on a single x-box.

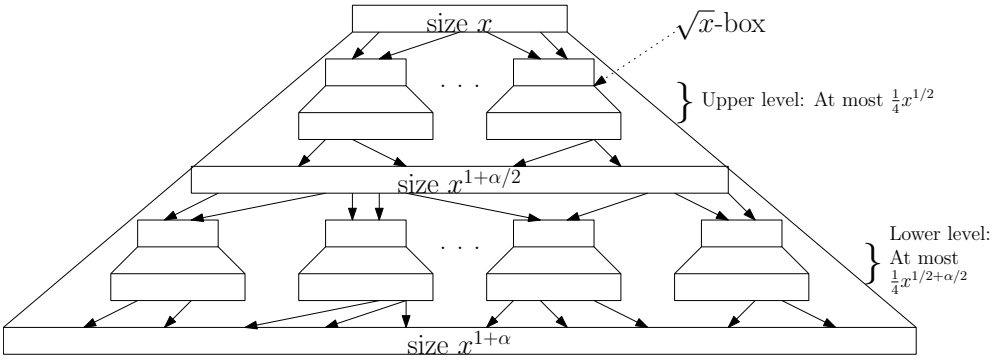


Figure 6.1: The layout of an x-box. The arrows indicate lookahead pointers evenly spaced in the target buffer, but not necessarily in the source buffer. The Figure is inspired by Figure 1 in [BDF+10].

The x-box structure contains three sorted buffers and two recursive layers of $\sqrt{x}$-boxes[1] named *subboxes*, see Figure 6.1. The three buffers are the *input buffer* of size x, the *middle buffer* of size $x^{1+\alpha/2}$ and the *output buffer* of size $x^{1+\alpha}$. The two layers of subboxes are referred to as the *upper level*, which consists of at most $\frac{1}{4}x^{1/2}$ subboxes, and the *lower level* which consists of at most $\frac{1}{4}x^{1/2+\alpha/2}$ subboxes, see Table 6.1. Thus the total number of subboxes is $\leq \frac{1}{2}x^{1/2+\alpha/2}$. For each level of subboxes we store a boolean array indicating whether a subbox is in use. An x-box, with associated subboxes, is laid out recursively in memory: First the input buffer, then the upper level boolean array, followed by the upper level subboxes, the middle buffer, the lower level boolean array, the lower level subboxes, and finally the output buffer. Notice that the placement of the subboxes in either level is not ordered.

The base $\mathcal{O}(1)$-box consists of a single array which makes up the input and the output buffer. There is no middle buffer and no subboxes.

The number of subboxes at the upper level contains a combined input buffer size of $\frac{1}{4}x$, a constant factor of the enclosing x-box's input buffer size, and similarly the combined size of the output buffers of the upper level subboxes is $\frac{1}{4}x^{1+\alpha/2}$, a constant factor of the enclosing x-box's middle buffer size. The number of subboxes at the lower level results in equivalent combined buffer sizes matching the middle and output buffer of the enclosing x-box to within a constant factor.

Buffer	Size per buffer	Number of buffers	Total size
Input Buffer	x	1	x
Input Buffer	$\sqrt{x}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x$
Middle Buffer	$(\sqrt{x})^{1+\alpha/2}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x^{1+\alpha/4}$
Output Buffer	$(\sqrt{x})^{1+\alpha}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x^{1+\alpha/2}$
Middle Buffer	$x^{1+\alpha/2}$	1	$x^{1+\alpha/2}$
Input Buffer	$\sqrt{x}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+\alpha/2}$
Middle Buffer	$(\sqrt{x})^{1+\alpha/2}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+3\alpha/4}$
Output Buffer	$(\sqrt{x})^{1+\alpha}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+\alpha}$
Output Buffer	$x^{1+\alpha}$	1	$x^{1+\alpha}$

Table 6.1: The size of buffers in an x-box. The table lists the size of the three main buffers in the x-box as well as the buffers of the first recursive $\sqrt{x}$-boxes. The table is inspired by Table 2 from [BDF+10].

The three buffers of the x-box contains elements in sorted order. The placement of an element in one buffer implies no relation to elements in the other buffers. However, each level of subboxes partition the keyspace of the enclosing buffers. The partition of the input buffer is implemented by pointers to each subbox, stored with the elements in the input buffer, indicating the minimum key of the subbox. These pointers are appropriately named *subbox pointers*. Likewise subbox pointers are stored in the middle buffer for the lower level subboxes. The

[1] In the rest of the section when referring explicitly to the size of a subbox we will use the notation $\sqrt{x}$ instead of $x^{1/2}$ for clarity.

subboxes are not placed in order of the subbox pointers, their order is solely denoted by the subbox pointers. There is no relation between the partition imposed by the two levels of subboxes. An element contained in the x-box is located in one of the three main buffers, or in a specific subbox at either the upper or lower level.

Inspired by fractional cascading [CG86a, CG86b] *lookahead pointers* are sampled upwards in the buffer hierarchy. For example the input buffer will contain pointers to a constant fraction of the elements in the input buffers of the upper level subboxes. For theoretical purposes we set this constant equal to sampling every 16th element. Similarly to subbox pointers the lookahead pointers are stored with the elements in the relevant buffers. See Figure 6.1 where the lookahead pointers are indicated by arrows.

Each main buffer thus contains a set of elements intermixed with subbox and lookahead pointers. The pointers are not necessarily evenly distributed amongst the set of elements. To enable constant time lookup of pointers we associate with each element a pointer to the nearest subbox or lookahead pointer, backwards and forwards, in the buffer the element is located in.

Storing the additional subbox and lookahead pointers in the buffers reduces the space for storing real elements. We therefore consider an x-box full when it contains $\frac{1}{2}x^{1+\alpha}$ real elements.

The structure, specifically the choice of three main buffers and two recursive layers of subboxes with the specific fanout, may seem nonintuitive, but the structure follows naturally once three original conditions are imposed. Firstly, we want the input to be x and the output to be $x^{1+\alpha}$. Secondly we want one or more recursive structures of size $\sqrt{x}$. Lastly we require for each subbox a subbox pointer, as well as a constant (in the size of the underlying buffer) number of lookahead pointers to perform a query. Without a middle buffer either the combined input layer of the subboxes would not match to within a constant size of the input buffer, or the combined output layer of the subboxes would not match to within a constant size of the output buffer, meaning we would not be able to sample a constant amount of lookahead pointers. This also specifices the fanout of the two level of subboxes to be $\mathcal{O}(x^{1/2})$ and $\mathcal{O}(x^{1/2+\alpha/2})$ respectively. Furthermore set $\alpha \approx 0$ and the constant for the fanout of the subboxes follows immediately from the need for the middle buffer to be able to contain the real elements in the input buffer (after inserting $\frac{1}{2}x$ new elements), contain the real elements in the upper level, and contain the real elements and subbox pointers in the middle buffer. This scenario occurs when we merge the elements in the input buffer and upper level subboxes into the middle buffer, during a Batch-Insert, see Section 6.3.

Having outlined the structure of an x-box we can now determine its space usage including that of the recursive structures.

Lemma 6.1. *The total space usage of an x-box is at most $cx^{1+\alpha}$ for some constant $c > 0$.*

Proof. We prove the lemma by induction. Each x-box consists of tree main buffers of size $c'(x+x^{1+\alpha/2}+x^{1+\alpha}) \leq 3c'x^{1+\alpha}$, where c' is a constant representing the additional pointers. The boolean arrays associated with the two levels

of subboxes use at most $\frac{1}{2}x^{1/2+\alpha/2} \leq c'x^{1+\alpha}$ space. The total space usage, disregarding the subboxes, is therefore $\leq 4c'x^{1+\alpha}$. By the induction assumption the subboxes use at most $c(\sqrt{x})^{1+\alpha} \cdot \frac{1}{2}x^{1/2+\alpha/2} = \frac{c}{2}x^{1+\alpha}$ space. For $c \geq 8c'$ this gives us a total space usage of at most $cx^{1+\alpha}$. □

Thus α describes a relation between x and B, determining the base case, where an x-box fits inside a single block of memory, to be $x^{1+\alpha} = \mathcal{O}(B)$ or conversely $x = \mathcal{O}(B^{1/(1+\alpha)})$. We will use this base case in the following sections to analyze the x-box.

6.2 Search in an x-box

For an x-box D we wish to support the operation SEARCH(D,s,k), which returns a pointer to an element with key k in D, or if no such element exists in D returns the predecessor of k in D's output buffer. We assume we are given a pointer to the nearest lookahead pointer s preceding k in D's input buffer.

We perform the SEARCH by scanning the input buffer of D from s until we find k or s', where s' is the first element such that $s' > k$. By the definition of lookahead pointers we scan at most $\mathcal{O}(1)$ elements. If we found k we can return, otherwise we follow the nearest lookahead or subbox pointer preceding $s' - 1$. The recursive call on the upper level subbox will return either k or a pointer into D's middle buffer. We continue the same process for the middle buffer, scanning a constant amount of elements, recursively call SEARCH on a subbox, and scanning a constant number of elements in the output buffer.

Lemma 6.2. *For $x > B$ a* SEARCH *in an x-box costs $\mathcal{O}((1+\alpha)\log_B x)$ I/O's.*

Proof. We spend $\mathcal{O}(1)$ I/O's scanning the tree main buffers, and perform two recursive calls. The cost of a SEARCH becomes the recurrence $S(x) = 2S(\sqrt{x}) + \mathcal{O}(1)$. Solving the recursion $S(x) = 2S(\sqrt{x}) + \mathcal{O}(1)$ requires us to solve the simpler recursion $S(x) = S(\sqrt{x}) + \mathcal{O}(1)$, which we do by modifying the equation as follows.

$$\begin{aligned} y = \log x &\implies S(2^y) = S(2^{y/2}) + \mathcal{O}(1) \\ T(y) = S(2^y) &\implies T(y) = T(y/2) + \mathcal{O}(1) \end{aligned}$$

By the master theorem [CLRS09] case 2 the recursion $T(y) = T(y/2) + \mathcal{O}(1)$ has the solution $T(y) = \mathcal{O}(\log y)$. It follows that $S(x) = S(2^y) = T(y) = \mathcal{O}(\log y) = \mathcal{O}(\log\log x)$.

Once $x^{1+\alpha} = \mathcal{O}(B)$, or equivalently $x = \mathcal{O}(B^{1/(1+\alpha)})$, all further recursive calls cause no extra I/O's. We can express this base case as $S(\mathcal{O}(B^{1/(1+\alpha)})) = 0$. The height of the recursion before reaching the base case consequently becomes $\mathcal{O}(\log\log x - \log\log B^{1/(1+\alpha)})$.

We can now express the cost of the original recursion $S(x) = 2S(\sqrt{x}) + \mathcal{O}(1)$, with its two recursive calls, as the following set of equations.

$$\begin{aligned}
\mathcal{O}\left(2^{\log\log x - \log\log B^{1/(1+\alpha)}}\right) &= \mathcal{O}\left(\frac{2^{\log\log x}}{2^{\log\log B^{1/(1+\alpha)}}}\right) \\
&= \mathcal{O}\left(\frac{\log x}{\log B^{1/(1+\alpha)}}\right) \\
&= \mathcal{O}\left(\frac{\log x}{(1/(1+\alpha))\log B}\right) \\
&= \mathcal{O}\left(\frac{(1+\alpha)\log x}{\log B}\right) \\
&= \mathcal{O}\left((1+\alpha)\log_B x\right)
\end{aligned}$$

□

6.3 Batch-Insert into an x-box

For an x-box D we wish to support the operation Batch-Insert(D,e_1,e_2,...,$e_{\Theta(x)}$), which inserts an array of $\Theta(x)$ sorted elements into D while maintaining lookahead pointers. As previously described we consider an x-box containing $\frac{1}{2}x^{1+\alpha}$ real elements full.

To support Batch-Insert we create two helper functions named Flush and Sample-Up. Flush will be responsible for flushing the real elements of the x-box and related subboxes down into the output buffer, and Sample-Up will be responsible for maintaining the lookahead pointers.

6.3.1 Flush

After we complete Flush(D) the k real elements of D will be located in the $\Theta(k)$ first slots of D's output buffer. No elements will reside in D's input buffer, middle buffer or subboxes, including the removal of all lookahead pointers except in D's output buffer. The output buffer will contain at most $\frac{1}{2}x^{1+\alpha}$ lookahead pointers into the enclosing x-box, see Section 6.3.2.

To Flush D we first Flush all D's subboxes. The result is the placement of each element in one of five places: The three buffers, or the output buffers of the upper and lower level subboxes. With the partitioning of the keyspace imposed by each level of subboxes we can consider the elements as located in five sorted lists. Using a five way merge we can place all elements in the output buffer. Reading from five different lists of combined size X can be done in $\mathcal{O}(\frac{X}{B})$ I/O's, noting that even if the scanning of one list pushes the other lists out of memory we will have read enough elements to pay for reading into memory the position in the remaining four lists. However, the lists contained in the subboxes are fragmented, needing a more detailed analysis.

Lemma 6.3. *For $x^{1+\alpha} > B$ a* Flush *on an x-box costs $\mathcal{O}(x^{1+\alpha}/B)$ I/O's.*

Proof. We can perform a Flush using a five way merge on D, with an extra cost added by the fragmented lists of the subboxes at each level. Furthermore

we need to recursively FLUSH all the subboxes. Thus we can describe FLUSH by the recurrence $F(x) = \mathcal{O}(x^{1+\alpha}/B) + \mathcal{O}(x^{1/2+\alpha/2}) + \frac{1}{2}x^{1/2+\alpha/2}F(\sqrt{x})$, where the first term is the merge, the second term a random access to each subbox, and the third term a recursive call on each subbox.

The second term disappears once we can load the entire x-box into memory, or equivalently when $x^{1+\alpha} = \mathcal{O}(M)$. Using the tall cache assumption $M = \Omega(B^2)$, of the cache oblivious model, see Section 2.2, the second term only occurs when $x^{1+\alpha} = \Omega(B^2) \rightarrow x^{1/2+\alpha/2} = \Omega(B)$. Using this expression we can reduce the second term to $x^{1/2+\alpha/2} = x^{1+\alpha}/x^{1/2+\alpha/2} = x^{1+\alpha}/\Omega(B) = \mathcal{O}(x^{1+\alpha}/B)$. This is equal to the first term and the recurrence $F(x)$ can be reduced to $F(x) \leq c_1 x^{1+\alpha}/B + \frac{1}{2}x^{1/2+\alpha/2}F(\sqrt{x})$ for some constant $c_1 > 0$.

Using induction we complete the Lemma as follows. As a base case when $y^{1+\alpha}$ fits in memory the cost becomes $F(y) = cy^{1+\alpha}/B$. Using the inductive hypothesis that $F(y) \leq cy^{1+\alpha}/B$ for some constant c and $y < x$, the inductive step becomes $F(x) \leq c_1 x^{1+\alpha}/B + \frac{1}{2}x^{1/2+\alpha/2}(cx^{1/2+\alpha/2}/B) = c_1 x^{1+\alpha}/B + \frac{1}{2}cx^{1+\alpha}/B$. For $c \geq 2c_1$ we now have that $F(x) = \mathcal{O}(x^{1+\alpha}/B)$. □

6.3.2 Sample-Up

The function SAMPLE-UP(D) will restore the lookahead pointers of x-box D. We assume SAMPLE-UP is called on an x-box with all $k < x^{1+\alpha}$ elements located in the output buffer. This includes lookahead pointers to the enclosing x-box. Sampling a lookahead pointer for every 16th element we create $(k/16)/(x^{1/2+\alpha/2}/2) = k/8x^{1/2+\alpha/2} < x^{1+\alpha}/(8x^{1/2+\alpha/2}) = x^{1/2+\alpha/2}/8$ new subboxes in the lower level, and assign to each of these subboxes $x^{1/2+\alpha/2}/2$ lookahead pointers. This fills up half the output buffer of half the subboxes in the lower level. We then recursively call SAMPLE-UP on the newly created subboxes, and sample lookahead pointers from their input buffers to the middle buffer. The same process is repeated for the upper level subboxes. Note that even if we assume the input buffers of the lower level subboxes are filled up by the recursive SAMPLE-UP's we will sample at most $\frac{1}{16\cdot 8}x^{1/2+\alpha/2}\sqrt{x} = \frac{1}{128}x^{1+\alpha/2}$ elements into the middle buffer. This will in turn create at most $\frac{1}{128}x^{1+\alpha/2}/(x^{1/2+\alpha/2}/2) = \frac{1}{64}x^{1/2}$ new upper level subboxes.

Lemma 6.4. *For $x^{1+\alpha} > B$* SAMPLE-UP *on an x-box costs $\mathcal{O}(x^{1+\alpha}/B)$ I/O's.*

Proof. The recursion for SAMPLE-UP as described above becomes $SU(x) \leq \mathcal{O}(x^{1+\alpha}/B) + \frac{1}{4}x^{1/2+\alpha/2}SU(\sqrt{x})$, noting that the upper level will create at most the same amount of subboxes as the lower level. The recursion is nearly identical to the reduced version of $F(x)$ which we solved to $\mathcal{O}(x^{1+\alpha}/B)$ I/O's in Lemma 6.3. □

6.3.3 Batch-Insert

Having described the FLUSH and SAMPLE-UP helper functions we can now describe the BATCH-INSERT function upon x-box D.

BATCH-INSERT takes as input a sorted array of $\Theta(x)$ elements. For theoretical purposes we set this to be $\frac{1}{2}x$ elements. We merge this array into the

input buffer and increment the counter of total elements in D. While merging we remove the lookahead pointers, to be restored later.

Using the subbox pointers we implicitly partition the input buffer. For any partition of the input buffer containing at least $\frac{1}{2}x^{1/2}$ elements, we repeatedly remove $\frac{1}{2}x^{1/2}$ elements from the partition, and recursively Batch-Insert them into the appropriate subbox, until the partition contains less than $\frac{1}{2}x^{1/2}$ elements.

During the process of repeated Batch-Insert calls, into a specific subbox D', if performing the next Batch-Insert would cause D' to exceed $\frac{1}{2}(\sqrt{x})^{1+\alpha}$ $= \frac{1}{2}x^{1/2+\alpha/2}$ real elements we "split" D'. A "split" on D' consists of a Flush(D') followed by the creation of a new subbox D''. Using a linear number of I/O's, in the size of D', we move half the elements from the output buffer of D' into the output buffer of D''. We then perform a Sample-Up on both subboxes to restore the lookahead pointers, and update the relevant counters. Notice that we place the new subbox in the first available place, not in order according to the range of elements they store. We find the first available space for a subbox in $\mathcal{O}(\frac{1}{4}x^{1/2}/B)$ or $\mathcal{O}(\frac{1}{4}x^{1/2+\alpha/2}/B)$ I/O's respectively. We cover this cost by charging $\mathcal{O}(\frac{1}{B})$ to all the elements inserted into the respective level of subboxes. Then we charge one random access for each split, to mark the space occupied, see the cost of random access below.

After completing all Batch-Insert calls into the top level subboxes the input buffer of D contains at most $\frac{1}{2}x^{1/2} \cdot \frac{1}{4}x^{1/2} = \frac{1}{8}x$ elements, as otherwise there would exist a partition large enough to be Batch-Insert'ed into a subbox.

We now resample lookahead pointers from the input buffers of the at most $\frac{1}{4}x^{1/2}$ subboxes resulting in a total of at most $\frac{1}{16}x^{1/2} \cdot \frac{1}{4}x^{1/2}$ lookahead pointers. When combined with the upper bound on the number of real elements in D's input buffer we see that the buffer is less than half full, i.e. there is space for another insertion into D.

When during the process of a split we allocate the last subbox in the upper level we abort the insertion and flush the upper level subboxes. We then merge the input buffer and the fragmented list in the output buffers of the upper level into the middle buffer. Then we start a new process of insertion, but now from the middle buffer into the lower level subboxes. This process can in turn fill up the lower level, upon which we repeat the procedure from the upper level and place all the elements into the output buffer. Dependent on which buffer the insertion completes at we either call Sample-Up on D or perform a partial Sample-Up from the middle buffer to the input buffer, using recursive Sample-Up's on the relevant levels of subboxes.

The total number of lookahead pointers in the upper level subbox's output buffers are at most $\frac{1}{16}x^{1+\alpha/2}$. When the elements from the input buffer and upper level is moved into the middle buffer these lookahead pointers are the only elements in the upper level, and distributed across at most $\frac{1}{16}x^{1+\alpha/2}/x^{1/2+\alpha/2}/2$ $= \frac{1}{8}x^{1/2}$ subboxes, which is half the maximum amount of subboxes in the upper level. As a consequence there must be at least $\frac{1}{8}x^{1/2}$ new splits before we move elements into the middle buffer, and since a split only occurs when each subbox contains $\frac{1}{2}(\sqrt{x})^{1+\alpha}$ real elements, each of the resulting subboxes will contain at

least $\frac{1}{4}(\sqrt{x})^{1+\alpha}$ real elements. Thus we must insert at least $\frac{1}{4}(\sqrt{x})^{1+\alpha} \cdot \frac{1}{8}x^{1/2} = \Omega(x^{1+\alpha/2})$ elements between moving elements into the middle buffer.

Similarly the total number of lookahead pointers in the lower level subboxs' output buffers are most $\frac{1}{16}x^{1+\alpha}$. When all the elements are contained in the output buffer we create at most $\frac{1}{16}x^{1+\alpha}/x^{1/2+\alpha/2}/2 = \frac{1}{8}x^{1/2+\alpha/2}$ subboxes in the lower level. Therefore we require at least $\frac{1}{4}(\sqrt{x})^{1+\alpha} \cdot \frac{1}{8}x^{1/2+\alpha/2} = \Omega(x^{1+\alpha})$ insertions into D between movements into the output buffer.

Theorem 6.5. *A* Batch-Insert *into an x-box with $x > B$ costs an amortized $\mathcal{O}((1+\alpha)\log_B(x)/B^{1/(1+\alpha)})$ I/O's pr. element.*

Proof. For each element we have several sources of cost for insertion. First we merge the element into an input array costing $\mathcal{O}(1/B)$ I/O's pr. element. Then we recursively insert each element into a subbox in the top level, during which we pay random accesses to load the first block of each subbox. We check this block to ensure we will not exceed capacity on insertion, and the cost is incurred under the assumption that we do not cache the block. Then we sample the subboxes, the cost of which is dominated by the cost to scan the elements and that of the random accesses. Each element will have to pay a cost for a potential split of the subbox it is inserted into. This cost can be amortized against the $\Omega((\sqrt{x})^{1+\alpha})$ elements required to split a subbox. We have to pay for moving elements from the output buffers of the top level into the middle buffer. This cost can be amortized against the $\Omega(x^{1+\alpha/2})$ elements in the output buffers. There are similar costs for the movement of element from the middle buffer, through the lower level subboxes, and into the output buffer.

The costs outlined are known, except the random accesses to subboxes. If all the upper level subboxes fit into memory, then the cost of random accesses becomes the minimum of performing the random accesses or simply loading in the entire upper level into memory. A random access to the upper level is denoted UpperRA(x). If the upper level does not fit into memory we have UpperRA(x) $= \mathcal{O}(\frac{1}{4}x^{1/2})$, and the cost becomes UpperRA(x) $= \mathcal{O}(\min\{\frac{1}{4}x^{1/2}, x^{1+\alpha/2}/B\})$ if the upper level fits in memory. This cost can be amortized against the $\Theta(x)$ elements inserted. We can analyze the two cases using the tall cache assumption $M = \Omega(B^2)$ of the cache oblivious model, see Section 2.2.

1. Consider the case when the upper level does not fit into memory, that is $x^{1+\alpha/2} = \Omega(M) = \Omega(B^2)$. It follows that $\sqrt[(1+\alpha/2)]{x^{1+\alpha/2}} = x = \Omega(\sqrt[(1+\alpha/2)]{B^2}) = \Omega(B^{2/(1+\alpha/2)})$, and consequently $x^{1/2} = \Omega(B^{2/(2+\alpha)})$. The cost of the amortized random accesses becomes UpperRA(x)$/x = \mathcal{O}(\frac{1}{4}x^{1/2})/x = \mathcal{O}(1/x^{1/2}) = \mathcal{O}(1/B^{2/(2+\alpha)})$.

2. Consider the case when the upper level fits into memory, that is $x^{1+\alpha/2} = \mathcal{O}(M) = \mathcal{O}(B^2)$. We now have two subcases, one case where we can only load in the upper level, that is $x > B^{2/(1+\alpha)}$, and one case where we can load the entire x-box into memory, meaning $x \leq B^{2/(1+\alpha)}$. In the first subcase we charge a constant number of I/Os pr. upper level subbox, and it follows that UpperRA(x)$/x = \mathcal{O}(x^{1/2}/x) = \mathcal{O}(1/\sqrt{x}) = \mathcal{O}(1/B^{1/(1+\alpha)})$.

The second subcase we pay for reading in all the subboxes of the upper level and the cost becomes $\text{UpperRA}(x)/x = \mathcal{O}(x^{1+\alpha/2}/(B \cdot x)) = \mathcal{O}(x^{\alpha/2}/B) = \mathcal{O}((B^{2/(1+\alpha)})^{\alpha/2}/B) = \mathcal{O}((B^{\alpha/(1+\alpha)})/B) = \mathcal{O}((B^{\alpha/(1+\alpha)}) /B^{(1+\alpha)/(1+\alpha)}) = \mathcal{O}(B^{(\frac{\alpha}{1+\alpha}-\frac{1+\alpha}{1+\alpha})}) = \mathcal{O}(B^{-1/(1+\alpha)}) = \mathcal{O}(1/B^{1/(1+\alpha)})$.

Because $\mathcal{O}(1/B^{2/(2+\alpha)}) < \mathcal{O}(1/B^{1+\alpha})$ the cost becomes $\text{UpperRA}(x)/x = \mathcal{O}(1/B^{1+\alpha})$.

Using a similar analysis for the lower level we get a cost of $\text{LowerRA}(x) = \mathcal{O}(\frac{1}{4}x^{1/2+\alpha/2})$ if the lower level does not fit into memory, and $\text{LowerRA}(x) = \mathcal{O}(\min \frac{1}{4}x^{1/2+\alpha/2}, x^{1+\alpha}/B)$ if the lower level fits into memory. This can be amortized against the $\Theta(x^{1+\alpha/2})$ elements moved.

1. Consider the case when the lower level does not fit into memory, that is $x^{1+\alpha} = \Omega(M) = \Omega(B^2)$. It follows that $\sqrt[(1+\alpha)]{x^{1+\alpha}} = x = \Omega(\sqrt[(1+\alpha)]{B^2}) = \Omega(B^{2/(1+\alpha)})$, and consequently $x^{1/2} = \Omega(B^{(2/(1+\alpha))^{1/2}} = \Omega(B^{1/(1+\alpha)})$. The cost of the amortized random accesses becomes $\text{LowerRA}(x)/x^{1+\alpha/2} = \mathcal{O}(\frac{1}{4}x^{1/2+\alpha/2})/x^{1+\alpha/2} = \mathcal{O}(1/x^{1/2}) = \mathcal{O}(1/B^{1/(1+\alpha)})$.

2. Consider the case when the lower level fits into memory, that is $x^{1+\alpha} = \mathcal{O}(M) = \mathcal{O}(B^2)$. As was the case with the upper level we have the same two subcases of being able to load in the lower level or the entire x-box. In the first subcase we charge a constant number of I/Os pr. lower level subbox, and it follows that $\text{LowerRA}(x)/x^{1+\alpha/2} = \mathcal{O}(x^{1/2+\alpha/2}/x^{1+\alpha/2}) = \mathcal{O}(1/x^{1/2}) = \mathcal{O}(1/B^{1/(1+\alpha)})$. The second subcase we pay for reading in all the subboxes of the lower level and the cost becomes $\text{LowerRA}(x)/x^{1+\alpha/2} = \mathcal{O}(x^{1+\alpha}/(B \cdot x^{1+\alpha/2})) = \mathcal{O}(x^{\alpha/2}/B) = \mathcal{O}(1/B^{1/(1+\alpha)})$.

We now have all the pieces to calculate the total cost of Batch-Insert. To summarize we have to pay for scanning the input buffer, random accesses to upper level subboxes, recursively calling Batch-Insert on upper level subboxes, splitting subboxes when necessary (which is dominated by the cost of Flush on each of the splitting subboxes), calling Flush on all upper level subboxes to place their elements in their output buffers, moving elements from the upper level subboxes to the middle buffer, and repeating the entire process for the movement from the middle buffer through the lower level to the output buffer. All of the operations can be amortized against the number of elements involved, or minimum number of elements between operations. This gives us the following equation.

$$
\begin{aligned}
I(x) &= \mathcal{O}\Big(\frac{x/B}{x}\Big) + \mathcal{O}\Big(\frac{\text{UpperRA}(x)}{x}\Big) + I(\sqrt{x}) \\
&\quad + \mathcal{O}\Big(\frac{F(\sqrt{x})}{x^{1/2+\alpha/2}}\Big) + \mathcal{O}\Big(\frac{\frac{1}{4}x^{1/2}F(\sqrt{x})}{x^{1+\alpha/2}}\Big) \\
&\quad + \mathcal{O}\Big(\frac{x^{1+\alpha/2}/B}{x^{1+\alpha/2}}\Big) + \mathcal{O}\Big(\frac{\text{LowerRA}(x)}{x^{1+\alpha/2}}\Big) + I(\sqrt{x}) \\
&\quad + \mathcal{O}\Big(\frac{F(\sqrt{x})}{x^{1/2+\alpha/2}}\Big) + \mathcal{O}\Big(\frac{\frac{1}{4}x^{1/2+\alpha/2}F(\sqrt{x})}{x^{1+\alpha}}\Big) \\
&\quad + \mathcal{O}\Big(\frac{x^{1+\alpha}/B}{x^{1+\alpha}}\Big) \\
&= \mathcal{O}(1/B) + \mathcal{O}\Big(\frac{\text{UpperRA}(x)}{x}\Big) + \mathcal{O}\Big(\frac{\text{LowerRA}(x)}{x^{1+\alpha/2}}\Big) \\
&\quad + \mathcal{O}\Big(\frac{F(\sqrt{x})}{x^{1/2+\alpha/2}}\Big) + 2I(\sqrt{x}) \\
&= \mathcal{O}(1/B) + \mathcal{O}(1/B^{1/(1+\alpha)}) \\
&\quad + \mathcal{O}\Big(\frac{x^{1/2+\alpha/2}/B}{x^{1/2+\alpha/2}}\Big) + 2I(\sqrt{x}) \\
&= \mathcal{O}(1/B^{1/(1+\alpha)}) + 2I(\sqrt{x})
\end{aligned}
$$

As a base case for the recursion we incur no further cost when the box fits inside a single block, that is $x = \mathcal{O}(B^{1/(1+\alpha)})$. The recursion $2I(\sqrt{x})$ was shown in Lemma 6.3 to have a height of $\mathcal{O}(\log\log x)$ and the total cost becomes $\mathcal{O}\Big(\frac{2^{\log\log x - \log\log B^{1/(1+\alpha)}}}{B^{1/(1+\alpha)}}\Big) = \mathcal{O}\Big(\frac{2^{\log\log x}}{B^{1/(1+\alpha)} \cdot 2^{\log\log B^{1/(1+\alpha)}}}\Big) = \mathcal{O}(\frac{(1+\alpha)\log x}{B^{1/(1+\alpha)}\log B})$ $= \mathcal{O}((1+\alpha)\log_B(x)/B^{1/(1+\alpha)})$.

□

6.4 Building a Dictionary out of x-boxes

Stacking x-boxes exponentially doubling in size to create an xDict we can support insertion and queries using the functions described above. The x-boxes are linked together by sampling the lookahead pointers from the ith box's input buffer into the $(i-1)$st box's output buffer. The size of the ith x-box becomes $x = 2^{(1+\alpha)^i}$, and the height of the xDict becomes $\log_{1+\alpha}\log_2 N$.

To query an xDict we perform a SEARCH on each x-box starting from the topmost box. A call to SEARCH on the ith box will provide the element or a pointer into the $(i+1)$st box, enabling a SEARCH on this box.

Theorem 6.6. *The xDict supports online queries in $\mathcal{O}(\frac{1}{\alpha}\log_B \frac{N}{M})$ I/O's.*

Proof. The xDict is a series of x-boxes of height $\log_{1+\alpha}\log_2 N$. From Lemma 6.2 the cost of SEARCH on a single x-box is $\mathcal{O}((1+\alpha)\log_B x)$ I/O's. We sum over the cost of calling SEARCH on each box in the xDict structure.

$$\sum_{i=0}^{\log_{1+\alpha}\log_2 N} \mathcal{O}((1+\alpha)\log_B(2^{(1+\alpha)^i}) = \mathcal{O}\left(\frac{1+\alpha}{\log_2 B}\sum_{i=0}^{\log_{1+\alpha}\log_2 N}(1+\alpha)^i\right)$$

We know the three following properties.

1. $(1+\alpha)^{(\log_{1+\alpha}\log_2 N)-j} = \frac{\log_2 N}{(1+\alpha)^j} \implies \sum_{j=0}^{\log_{1+\alpha}\log_2 N}(1+\alpha)^j \leq \log_2 N \sum_{j=0}^{\infty}\frac{1}{(1+\alpha)^j}$
2. The geometric series $\sum_{i=0}^{\infty}\frac{1}{(1+\alpha)^i}$ converges to $\frac{1+\alpha}{\alpha}$
3. For $0 < \alpha \leq 1$ we have $(1+\alpha)^2 = \mathcal{O}(1)$

This leads us to the equation.

$$\frac{1+\alpha}{\log_2 B}\sum_{i=0}^{\log_{1+\alpha}\log_2 N}(1+\alpha)^i \leq \frac{(1+\alpha)\log_2 N}{\log_2 B}\sum_{i=0}^{\infty}\frac{1}{(1+\alpha)^i} = \frac{(1+\alpha)^2}{\alpha}\log_B N = \mathcal{O}(\tfrac{1}{\alpha}\log_B N)$$

Finally we note that an x-box of size $x^{1+\alpha} = \mathcal{O}(M) \implies x = \mathcal{O}(M^{1/(1+\alpha)})$ will fit in memory. In fact all x-boxes above the largest x-box that fulfills this space requirement will fit in memory, as the size increases superexponentially. Thus $\mathcal{O}(M)$ memory can hold $\mathcal{O}(M)$ elements. Assuming these elements are buffered in memory we finish the proof noting that the first $\mathcal{O}(\frac{1}{\alpha}\log_B M^{1/(1+\alpha)}) = \mathcal{O}(\frac{1}{\alpha}\log_B M)$ memory transfers come free, reducing the cost to $\mathcal{O}(\frac{1}{\alpha}\log_B \frac{N}{M})$.

□

To insert an element into the xDict we insert it into the topmost box $i = 0$ of size $x = \mathcal{O}(1)$. This box is size $\mathcal{O}(1)$ and thus supports single insertions. When the ith box reaches its full capacity of $\frac{1}{2}2^{(1+\alpha)^{i+1}}$ we FLUSH it, placing its real elements in its output buffer. These elements we BATCH-INSERT into the $(i+1)$st box. We continue in this manner, pushing elements down into the xDict, until we reach the jth box, where the jth box is the first box that can accommodate the inserted elements without reaching its capacity. Note that the recursive FLUSH call will leave all boxes above the jth box empty. We rebuild the lookahead pointers by calling SAMPLE-UP on the $(j-1)$st box and upwards in the structure.

Theorem 6.7. *The xDict supports single element insertions in amortized $\mathcal{O}(\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)})$ I/O's.*

Proof. From Theorem 6.5 the cost of BATCH-INSERT is amortized $\mathcal{O}((1+\alpha)\log_B(x)/B^{1/(1+\alpha)})$ I/O's pr. element. We sum over the cost of calling BATCH-INSERT on each box in the xDict structure. The analysis is identical to Theorem 6.6, except we multiply all costs by $\mathcal{O}(1/B^{1/(1+\alpha)})$. The cost of BATCH-INSERT clearly dominates the cost of the following SAMPLE-UP's required to restore lookahead pointers to the xDict. □

To match the lower bounds on insertion and query from Theorem 3.1 we need to perform one final step. Setting $\alpha = \frac{\varepsilon}{1-\varepsilon}$ for $0 < \varepsilon < \frac{1}{2}$ we obtain that $\frac{1}{\alpha} = \frac{1-\varepsilon}{\varepsilon} = \mathcal{O}(\frac{1}{\varepsilon})$ and $B^{1/(1+\alpha)} = B^{1-\varepsilon}$. Since $\alpha < 1$ we can apply Theorem 6.6 and 6.7 to obtain a query bound of $\mathcal{O}(\frac{1}{\varepsilon}\log_B \frac{N}{M})$ I/O's, and an insert bound of $\mathcal{O}(\frac{1}{\varepsilon}\log_B(\frac{N}{M})/B^{1-\varepsilon})$ I/O's, which matches the lower bounds to within a constant factor.

Recall that the tradeoff between Batch-Insert and Search on the x-box did not exist (lower alpha is faster for both). As shown above there exists a tradeoff between insertion and query for the xDict. The difference lies in α directly effecting the height of the xDict, and thus the number of Search calls on individual x-boxes we perform, as well as the size of the x-box at height i in the xDict. For lower values of α the i'th x-box's size decreases, but the maximum height increases. The size of the largest x-box in the xDict, and as a consequence the depth of that x-box's recursion, increases as α decreases. Thus the total number of layers across the entire xDict, and consequently the cost of Search, increases as α decreases. Intuitively one would expect the cost of Batch-Insert to similarly increase with the number of layers in the xDict. Due to random accesses this is not the case. Intuitively we amortize the cost of random accesses over more elements for lower values of α.

Deletion in the xDict is performed using global rebuilding. Upon a delete we insert an anti element in the xDict, and every time the number of deletions makes up half the number of updates we flush the xDict, gathering all elements in the output buffer of the last x-box, and rebuild the structure.

An xDict containing N elements possibly contains an N-box using $\mathcal{O}(N^{1+\alpha})$ address space, dominating the space of the entire xDict. The N-box will store N elements in the upper level, allowing us to compress the structure to N space by removing the lower level and output buffer, as each subbox will use memory linear in the number of elements stored in the subbox. Thus the entire xDict use optimal $\mathcal{O}(N)$ space.

6.5 Implementation

The xDict was implemented using mmap to map a sparse file against an array of longs. This allows us to ignore the parameters M and B, as pr. the Cache Oblivious model, instead relying on the operating system to handle any I/O calls. Each x-box is laid out recursively, with a cutoff point at $x = 16$ where an x-box is represented in its base case of a single array of size $x^{1+\alpha}$. Note that at $x = 16$ we have exactly one upper level subbox. Instead of relying on a boolean array to indicate the use of a subbox we use an array of pointers to facilitate fast subbox lookups, and to keep the array in sorted order with regards to the partition imposed by the subboxes. Notice that a split will result in the addition of a new subbox pointer to partition the above buffer. If the sole placement of the minimum key of the subbox is to be kept in this subbox pointer we must immediately insert it into the buffer following a split. Rather than this costly approach we store alongside the array of pointers to subboxes a set of minimum keys, restoring subbox pointers after we complete all of the insertions. Subboxes

are deleted by setting its pointer to zero.

We use a single sparse file to contain the xDict. Sparse files only allocates blocks we write to. In our concrete example the space of the output buffer of the largest x-box in the xDict might be empty. In this case it will take up only one block of space, since we write -1 to the buffer to indicate its empty. We do this for every buffer. Thus the input buffer of an x-box will only be allocated in the file if we pushed elements through it, which is minimum $\frac{1}{2}x$. The same holds for the size of the remaining buffers. When a subbox splits it contains $\mathcal{O}((\sqrt{x})^{1+\alpha})$ real elements and the space is $\mathcal{O}(x^{1/2+\alpha/2})$. This will compress the entire xDict structure to $\mathcal{O}(N)$ space.

7
Experimental Evaluation

In this section we experimentally evaluate and compare the data structures implemented in the thesis. First we evaluate the cache aware structures, and then we evaluate the cache oblivious xDictionary.

We implemented insertion and query for all structures, as well as deletion for Modified B-Tree, Buffered B-Tree and Buffer Tree, the latter of which acted as a prototype for Truncated Buffer Tree. Truncated Buffer Tree, as noted in Section 5.3 does not support deletion, and consequently we did not perform tests for deletion on the structures. Explicit internalization and externalization was implemented for Modified B-Tree and Buffered B-Tree, and is used in the tests. The structures were implemented in C++, and in total the structures and prototypes consists of approximately 23.700 lines of code.

The tests were all performed on the same machine, for details we refer the reader to Appendix B.1. The machine was set to only use a limited amount of RAM, and we accordingly set the internal memory size $M = 8$MB and the block size $B = 128$KB. The actual amount of free memory on the test machine was 16MB, i.e. $\Theta(M)$. The datastructures use M and B in units of elements, which will vary from data structure to data structure, with some using only a key and value for each element, others using key, value and time. For the cache aware structures these variables where all integers, while the xDictionary uses longs. The range of N starts at a value at least fifty times larger than M, forcing the structures out into external memory. For these experiments N was limited to 800 million elements, in order to complete the experiments within the limits of the thesis.

Each experiment was run ten times, unless otherwise noted, and the average value used. This was in order to filter out noise from background processes running on the test machine, as well as limiting the impact of randomness. While a dictionary is intended to have unique keys, for testing purposes the structures where modified to handle elements with identical keys, after the initial implementation. The universe size of the keys where set to size $2 \cdot N$, which according to the theory of Balls and Bins [RS98] gives an expected maximum collision of at most $\frac{\log N}{\log \log N}(1 - o(1))$ elements. The chosen universe size is a balance between the desire for unique elements and the desire for a random query to find an element.

We have three different options for measuring the I/O performance of our structures.

1. Calls we explicitly make in the program.
2. Calls the OS makes as it bundles our explicit calls.
3. Sectors read/written to disk.

Each of these options have their pros and cons. Option one accurately measures the program flow, but does not take caching into account, and will thus not reflect the actual work performed, nor will it be of any use for the cache oblivious structure where we make no explicit I/O calls. Option two weighs calls to larger blocks the same as calls to smaller blocks, and will thus not accurately reflect the work carried out for a long scan. Option three will accurately measure the work of a long scan, but will not accurately reflect the seek time to access many widely distributed small blocks. We chose option three as the measure of I/O performance, under the assumption that seek time is at most a factor two of read/write. For details on how the data was collected we refer the reader to Appendix B.2.

7.1 Modified B-Tree

In order to experimentally evaluate a Modified B-Tree we build a tree bottom up by first sorting the elements using external merge sort. The nodes will be given a fanout between $\frac{1}{2}B$ and $2B$, except for the rightmost path of the tree which may be left deform, that is with a fanout between 1 and $2B$.

Theorem 5.1 stipulates that insertions on a Modified B-Tree costs $\mathcal{O}(\log_B(\frac{N}{M})) = c \cdot \log_B(\frac{N}{M})$. We divide the time and number of I/O's with the expected cost to see if they converge to a constant. Note that the measurements is over one million insertions, which follows an initial one million insertions. In the buffered structures we fill up buffers with the initial insertions and thus pre-load the structure. To make the measurements on the Modified B-Tree consistent with the buffered structure we therefore also pre-load the tree with insertions before measuring.

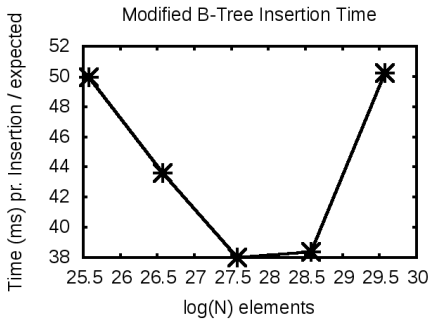


Figure 7.1: Time pr. insertion divided by expected $\log_B(\frac{N}{M})$.

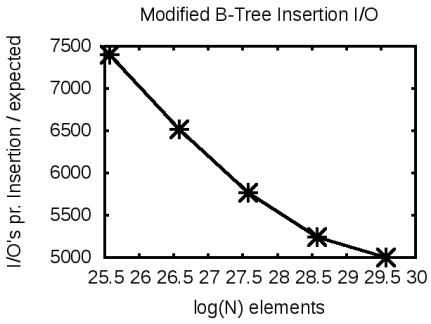


Figure 7.2: I/O's pr. insertion divided by expected $\log_B(\frac{N}{M})$ I/O's.

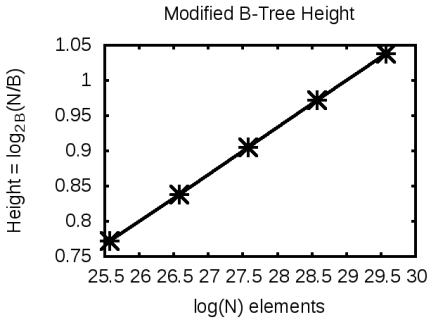


Figure 7.3: Continuous height of a Modified B-Tree. Real height is the ceiling of the continuous value.

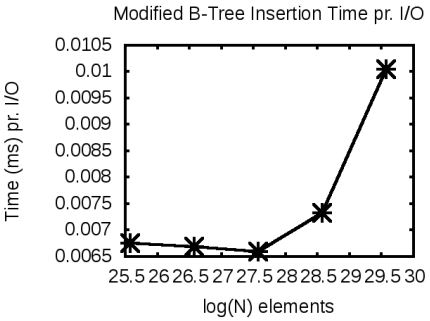


Figure 7.4: Time pr. I/O operation.

Figure 7.1 shows that insertion time does not converge to a constant, and Figure 7.2 shows the same result for the number of I/O operations. The two graphs do not follow the same curve, contrary to the assumption that the structure is I/O bound.

From Figure 7.1 and 7.3 it is clear that the increase in insertion time follows an increase in the external height of the tree.

The observation that Figures 7.1 and 7.2 do not follow the same curve, and that the time pr. insertion increases more than we expect in practice, led me to graph the time pr. I/O operation in Figure 7.4. The graph shows that as the height increases so does the cost of each I/O operation. Notice that for $x = 28.5$ the structure achieved an increase in height for one out of the ten runs, hence the slight increase in time pr. I/O operation. Figure A.1 and A.2 does not show the same increase in time pr. I/O for queries. The main difference between insertion and query is that upon inserting we create, read and write to files, whereas a query merely reads the present files. The filesystem ext4 indexes the files with an Extent Tree [ext], a special B-Tree, of height $\mathcal{O}(\log_{340}(\frac{N}{B}))$. We investigated the impact of this tree on the performance of creating and writing to files.

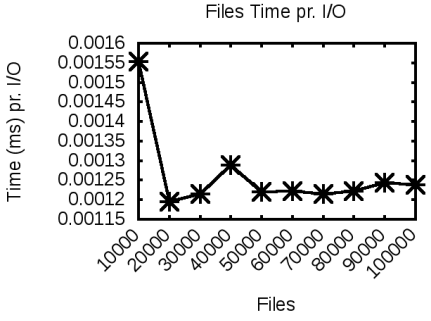


Figure 7.5: Time pr. I/O writing out files.

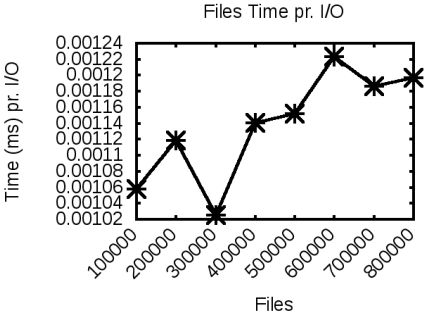


Figure 7.6: Time pr. I/O writing out files.

For the given range of N we expect less than one hundred thousand files, which is graphed in Figure 7.5. The time pr. I/O operation remains constant. The graph in Figure 7.6 seems to be increasing, but note that it is merely increasing to the same constant as in Figure 7.5. We thus conclude that the number of files, and the Extent Tree used to index them, does not have an impact upon the time pr. I/O operation.

Further inspection of the structure revealed that an optimization, where upon insertion only nodes whose keys had changed would be written to disk, had created a memory leak in the structure. The code responsible for cleaning up the memory used to read in the node was placed in the function that wrote the node back to disk. We changed the code to remove the memory leak and reran the tests for a single run on each value of N.

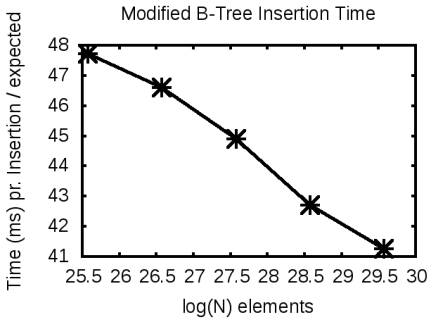


Figure 7.7: Time pr. insertion divided by expected $\log_B(\frac{N}{M})$.

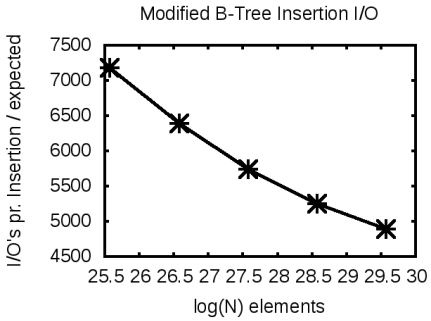


Figure 7.8: I/O's pr. insertion divided by expected $\log_B(\frac{N}{M})$ I/O's.

We observe in Figures 7.7 and 7.8 that insertion time now follows the number of I/O operations more closely, but the insertion cost does not converge to a constant.

We note that we only explicitly keep the root of the tree in internal memory, since our parameters for M and B is such that $M < B^2$. Therefore we divide the measured performance with expected $\mathcal{O}(\log_B(\frac{N}{B})) = \mathcal{O}(\log_B(N))$, to investigate if the failure to converge to a constant for expected $\mathcal{O}(\log_B(\frac{N}{M}))$ is due to not utilizing the internal memory as a cache.

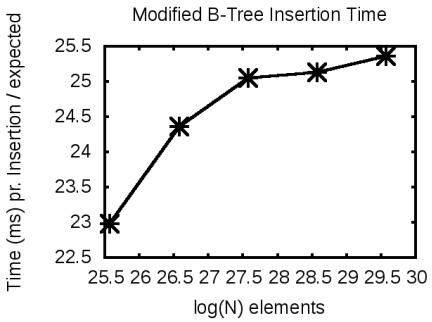


Figure 7.9: Time pr. insertion divided by expected $\log_B(N)$.

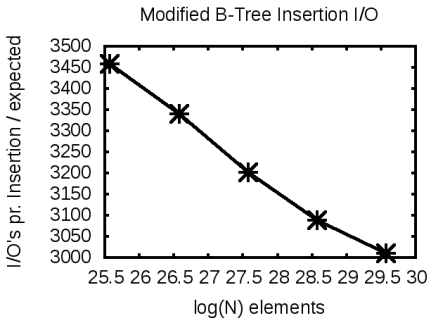


Figure 7.10: I/O's pr. insertion divided by expected $\log_B(N)$ I/O's.

Figures 7.9 and 7.10 shows that neither time nor number of I/O operations converge to the new expected value. We observe that the curve in Figure 7.7 is declining and the curve in Figure 7.9 is inclining, and that consequently the real time cost of insertion lies in between the two expected functions. Figure 7.8 and 7.10 are both declining, and consequently the real cost for I/O does not lie in between the two expected costs, meaning the structure is not I/O bound, contrary to theory.

Theorem 5.2 stipulates that an online query on a Modified B-Tree costs $\mathcal{O}(\log_B(\frac{N}{M})) = c \cdot \log_B(\frac{N}{M})$. We divide the measurements with the expected cost to determine if they converge to a constant. Due to running these tests after fixing the memory leak these tests were only run once.

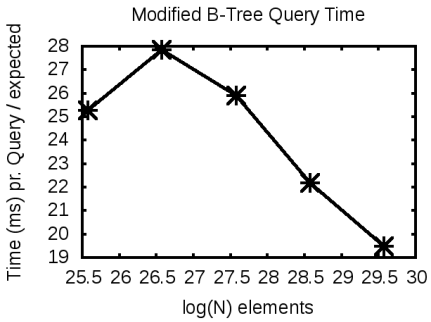


Figure 7.11: Time pr. Query divided with expected $\log_B(\frac{N}{M})$

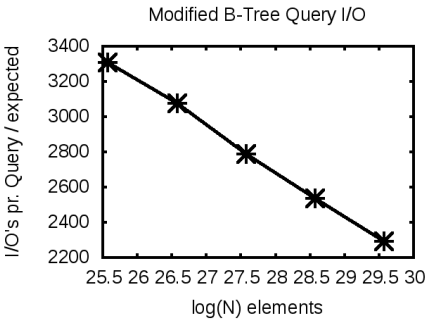


Figure 7.12: I/O's pr. query divided with expected $\log_B(\frac{N}{M})$

Figures 7.7 and 7.12 shows that the query cost does not converge to a constant. As we did with insertion we investigate if query cost follows expected $\mathcal{O}(\log_B(N))$ more closely.

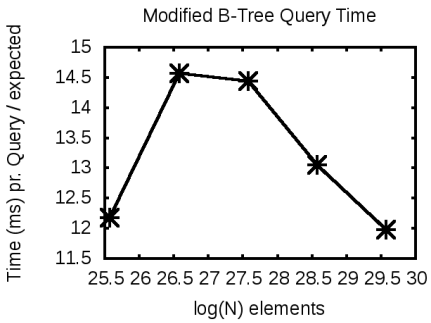


Figure 7.13: Time pr. Query divided with expected $\log_B(\frac{N}{B})$

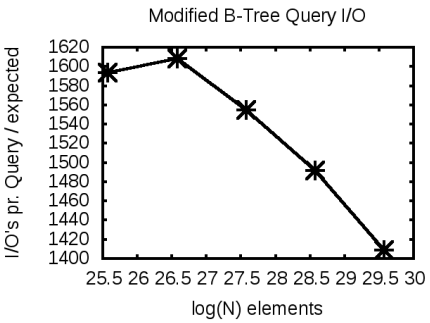


Figure 7.14: I/O's pr. query divided with expected $\log_B(\frac{N}{B})$

Figures 7.13 and 7.14 shows that the query cost does not converge to expected $\mathcal{O}(\log_B(N))$ either.

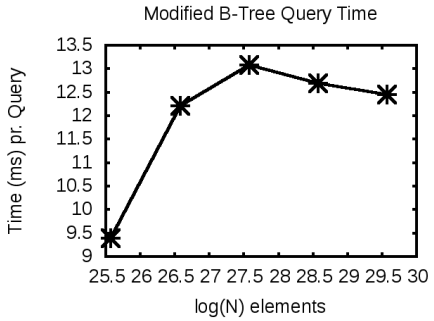


Figure 7.15: Query time pr. element.

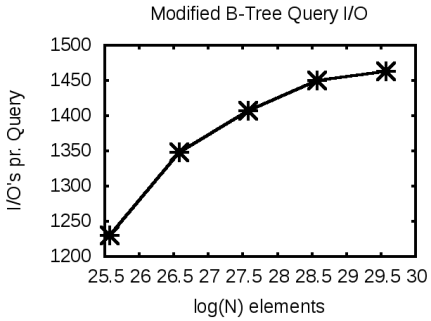


Figure 7.16: Query I/O pr. element.

Figures 7.15 and 7.16 shows that when we look at the raw data the query time remains more or less constant, while the number of I/O operations increases. This suggests that the query is not I/O bound, contrary to theory.

7.2 Buffered B-Tree

To evaluate the Buffered B-Tree we first measure the tradeoff between insertion and query determined by the value δ, for $\log N < \delta \leq B \log N$, which for this structure determines the fanout of the nodes. We set $N = 800$ million elements and vary δ over the range for five evenly distributed values. We construct a tree of size N minus one million bottom up using external merge sort to order the elements. For each node we select a random fanout between $\frac{\delta}{4 \log N}$ and $\frac{\delta}{\log N}$, except for the rightmost path of the tree, where the fanout is between 1 and $\frac{\delta}{\log N}$. We then insert an additional one million elements to populate the buffers in the tree and achieve the desired size of N elements. This forces the insertions out into external memory and also caches random parts of the tree in memory before measurements begin.

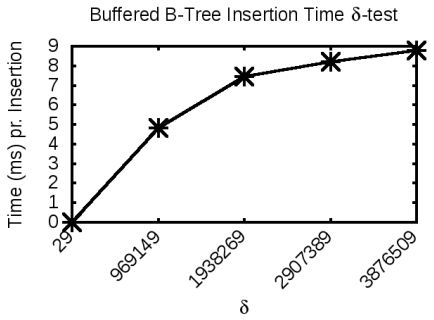


Figure 7.17: Insertion time pr. element.

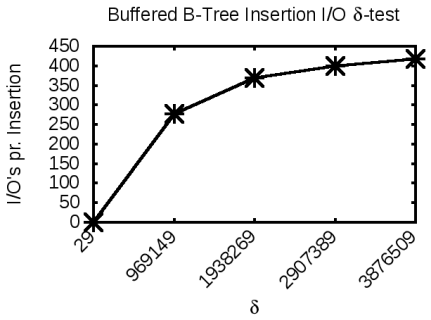


Figure 7.18: Insertion I/O pr. element.

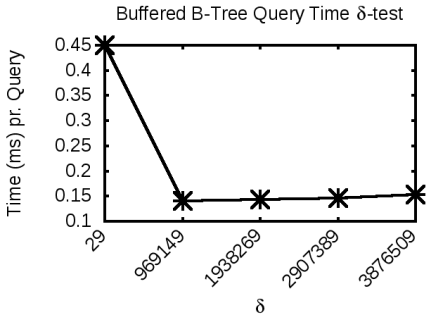


Figure 7.19: Query time pr. element.

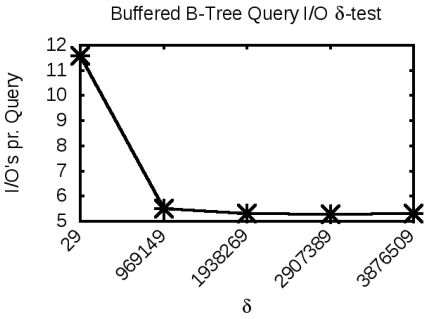


Figure 7.20: Query I/O pr. element.

Figure 7.17 to 7.20 shows that the tree flattens out for any but the smallest value of δ. Note an implementation detail for which all fanouts below two is changed to a fanout between two and eight. For $\delta = \log N = 29$, which would give a fanout of one, this special rule is applied, i.e. $\delta > \log N$. For such a low fanout we keep several layers of nodes in internal memory, reducing the external height. The external height of the tree for the remaining values of δ is two, with only the root being kept in internal memory. To properly evaluate δ we would have to measure the structure for much larger values of N. We therefore proceed with a fanout between two and eight, as it is the only interesting value within the scope of our experiments.

Theorem 5.3 stipulates that the insertion cost for a Buffered B-Tree is $\mathcal{O}(\delta \cdot \frac{N}{B}) = c \cdot \delta \cdot \frac{N}{B}$. We divide the measured time and number of I/O operations with the expected cost to determine if they converge to a constant.

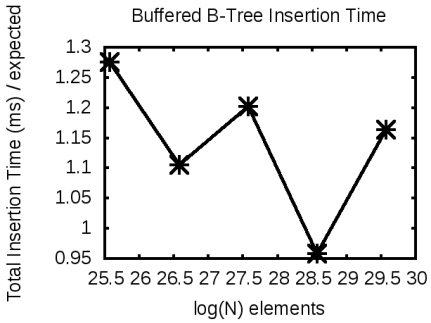


Figure 7.21: Total time for one million insertions divided by expected $\delta \cdot \frac{1mil}{B}$.

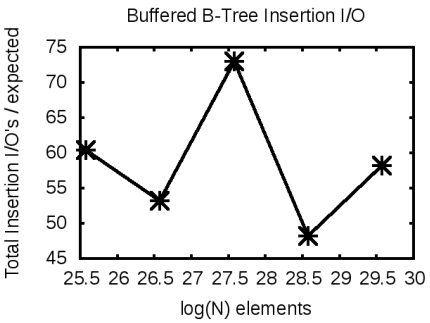


Figure 7.22: Total I/O's for one million insertions divided by expected $\delta \cdot \frac{1mil}{B}$ I/O's.

Figures 7.21 and 7.22 shows that while the insertion time and number of I/O's fluctuate they are not dependent on the height of the tree. This is consistent with theory. The fluctuation can be caused by the randomness of the construction and subsequent insertions. Further testing is needed to conclusively remove noise and prove that the insertion cost converges to a constant.

Theorem 5.4 stipulates that the query cost of a Buffered B-Tree is $\mathcal{O}(\log_{\delta/\log N}(\frac{N}{M}))$, or with our specific parameters $c \cdot \log_8(\frac{N}{M})$. We divide the measurements with the expected time to determine if the query cost converges to a constant.

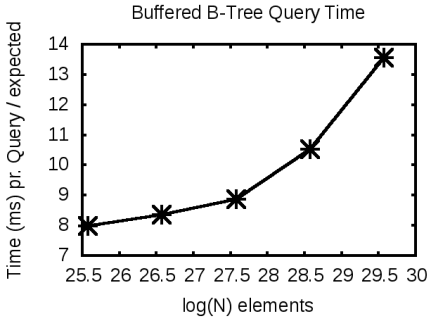


Figure 7.23: Time pr. query divided by expected $\log_8(\frac{N}{M})$.

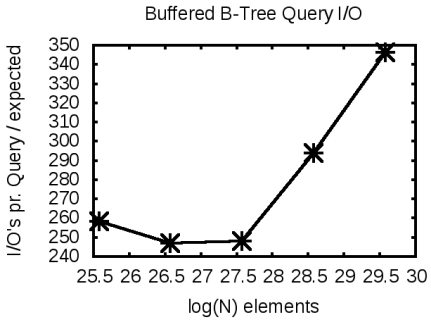


Figure 7.24: I/O's pr. query divided by expected $\log_8(\frac{N}{M})$ I/O's.

We observe in Figures 7.23 and 7.24 that the query does not converge to a constant.

During insertion we placed two million elements in the buffers of the tree. A subsequent query for any of these elements will return faster than a query for a non existing element. If the inserted elements can not fill out the external buffers then lower values for N will have proportionally more elements residing in the buffers, and will consequently query faster. The minimum number of elements to fill out the buffers of the tree for the lowest value of N is 5.6 million elements. The non-proportionally full buffers creates a lot of noise in the measurements.

The query does not converge to N, $log_8(\frac{N}{B})$, $log_8^2(\frac{N}{B})$ or $\sqrt{N}$. The conclusion is that there is too much noise from the buffers on the relatively few queries to determine the real cost of queries.

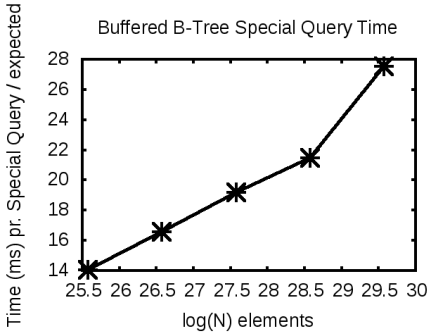


Figure 7.25: Time pr. special query divided by expected $\log_8(\frac{N}{M})$.

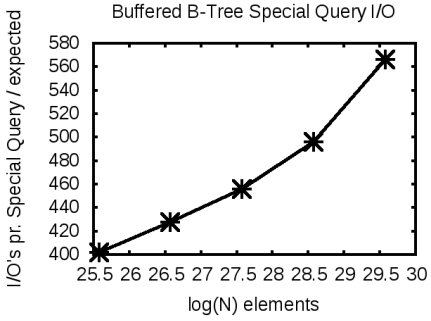


Figure 7.26: I/O's pr. special query divided by $\log_8(\frac{N}{M})$ I/O's.

The special query described in Section 5.2.1 where we "soft"-flush the buffers as

we traverse down the tree, to speed up future queries, does not converge to a constant. Nor do we see a decrease in query time when comparing Figures 7.23 and 7.25. Figures 7.24 and 7.26 confirms that the "soft"-flush does not decrease the I/O's pr. query either. The curve of the two graphs suggests that the special query is I/O bound. Performing only 10.000 queries is a sufficient amount of queries to flush the entire tree for low N (#leafs $= \frac{N}{B} = 4578$ for $N = 50$mil), and too few queries to flush the entire tree for high N (#leafs $= \frac{N}{B} = 73247$ for $N = 800$mil). We conclude that we perform too few queries to subsequently take advantage of the speedup in query, as both time and number of I/O's increases with the special query compared to the regular query for all N.

7.3 Truncated Buffer Tree

In order to experimentally evaluate Truncated Buffer Tree we build a tree by direct insertions and measure the tradeoff between insertion and query determined by the value of δ. For this structure δ is the height of the tree. We set $N = 800$ million elements and vary δ from one to four. In the following subsection $\delta = 1$ is a tree consisting of a single buffered node with a single bucket beneath it.

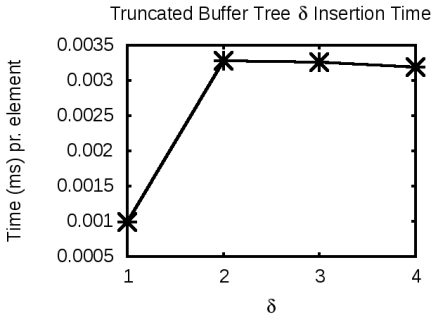


Figure 7.27: Insertion time pr. element.

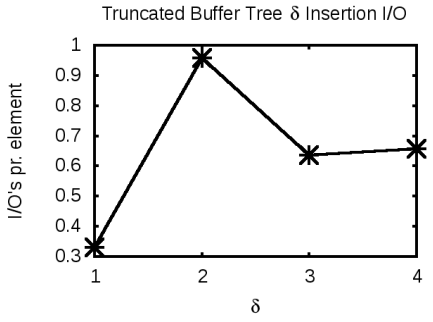


Figure 7.28: Insertion I/O's pr. element.

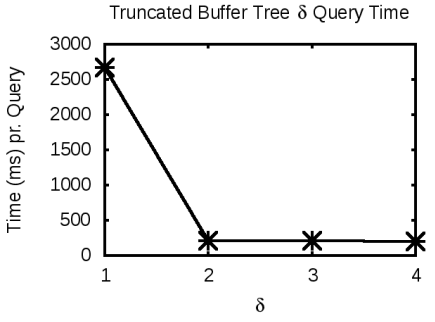


Figure 7.29: Query time pr. element

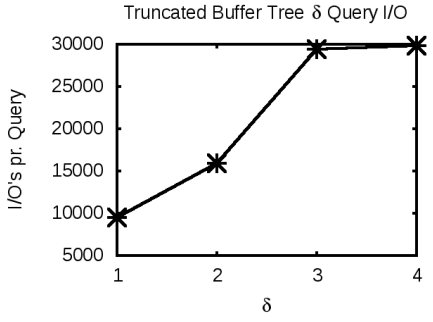


Figure 7.30: Query I/O pr. element

Figures 7.27 to 7.30 shows that as δ increases the cost of insertion goes up and the cost of query goes down. This is consistent with theory. Note that there is no difference between $\delta = 3$ and $\delta = 4$, as the tree never achieves the required height.

There seems to be no direct correlation between time and number of I/O's for both insertion and query, suggesting the structure is not I/O bound. Several factors might contribute to this result. When $\delta = 1$, the top of the single bucket will be cached for queries. Furthermore the M elements contained in each buffer results in a faster query for these elements. Higher δ results in more buffers and thus more "cheap" elements. This will obfuscate the measurements of queries. For insertions lower values of δ results in larger buckets and fewer splits, but a split will carry a larger cost. The way we count I/O operations we consider a larger split equal to several smaller splits, as long as the same amount of elements are involved, but this fails to take into account the increased seek time of the smaller splits.

Excluding $\delta = 1$, because it is merely fractional cascading with a buffer, we pick $\delta = 4$ to proceed with, being of interest as it will in practice be expected to behave similar to a Buffered B-Tree, with a different leaf structure.

Theorem 5.5 stipulates that insertion into a Truncated Buffer Tree costs $\mathcal{O}(\delta \cdot \frac{N}{B}) = c \cdot \delta \cdot \frac{N}{B}$. We divide the measured time with the expected time to determine if it converges to a constant.

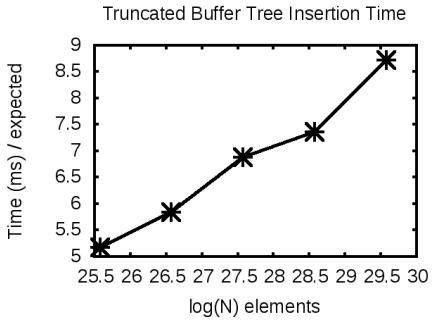


Figure 7.31: Insertion time divided by expected $\delta \cdot \frac{N}{B}$.

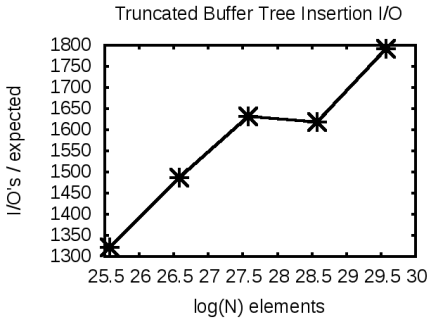


Figure 7.32: Insertion I/O's divided by expected $\delta \cdot \frac{N}{B}$.

From Figure 7.31 and 7.32 we conclude that the cost of insertion does not converge to a constant. The two graphs do not follow the same curve to a degree where we can conclude insertion to be I/O bound.

The structure with the given value for δ will in practice behave like a Buffer Tree, i.e. we do not cut insertions off before reaching size $\mathcal{O}(M)$ of a bucket. We therefore divide the data with the expected insertion time of $\mathcal{O}(\frac{N}{B} \log_{M/B}(\frac{N}{B}))$ from a Buffer Tree.

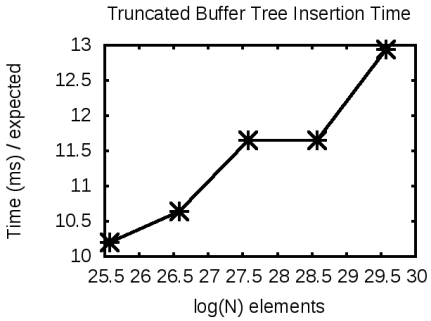


Figure 7.33: Insertion time divided by expected $\frac{N}{B}\log_{M/B}(\frac{N}{B})$.

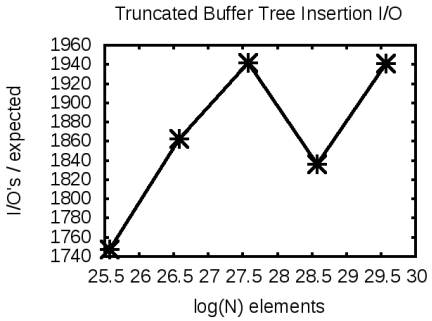


Figure 7.34: Insertion I/O's divided by expected $\frac{N}{B}\log_{M/B}(\frac{N}{B})$.

Figure 7.34 is possibly converging to a constant for the number of I/O operations, but we need more runs on larger sets of data to remove the noise and make any conclusions. Figure 7.33 does not seem to converge to a constant.

Theorem 5.6 stipulates the query cost of Truncated Buffer Tree to be $\mathcal{O}(N/(M(\frac{M}{B})^{\Omega(\delta)}))) = c \cdot N/(M(\frac{M}{B})^{\delta-1})$. Notice that we use $\delta - 1$ since we want $\delta = 1$ to give a bucket size of N. The maximum bucket size is $N/(M(\frac{M}{B})^{\delta-1}) < 1$. The bucket size is obviously rounded up. We divide the measured cost with the expected cost to determine if it converges to a constant.

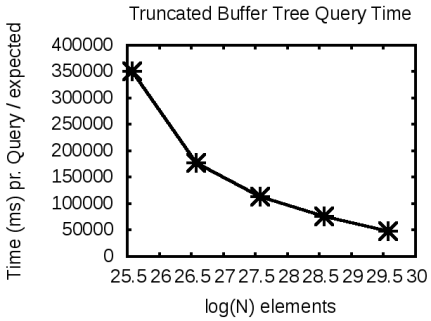


Figure 7.35: Time pr. query divided by expected $N/(M(\frac{M}{B})^3)$.

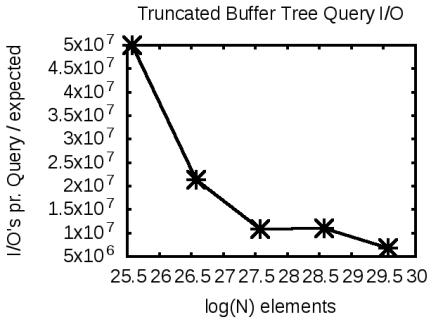


Figure 7.36: I/O's pr. query divided by expected $N/(M(\frac{M}{B})^3)$.

Figures 7.35 and 7.36 does not converge to a constant This is not surprising, as we showed in Theorem 5.6 that we need $N \geq M \cdot (\frac{M}{B})^{c\delta}$, for $c > 1$, in order for the buckets to dominate the scanning of the buffers in the tree. For our specific parameters, and $c = 1$, we require $N \geq 183,251,937,963$ elements before the buckets dominates the buffers.

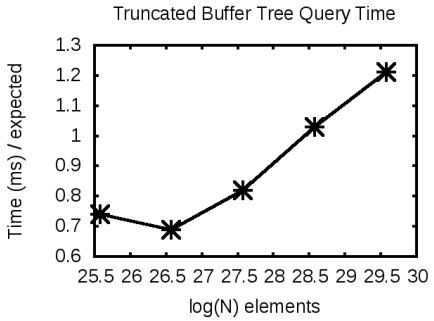


Figure 7.37: Time pr. query divided by expected $\frac{M}{B}\log_{M/B}(\frac{N}{B})$.

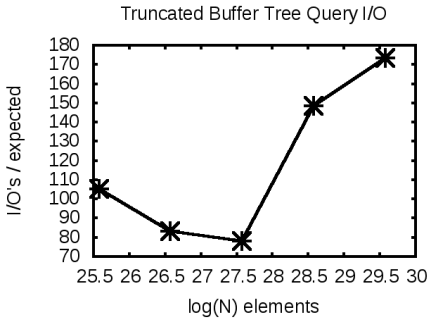


Figure 7.38: I/O's pr. divided by expected $\frac{M}{B}\log_{M/B}(\frac{N}{B})$ I/O's.

We investigate whether the query cost converges to a constant for scanning the buffers in the tree. Figures 7.37 and 7.38 shows the that query cost does not converge to a constant.

Like in Buffered B-Trees any elements contained in the buffers will return quickly. This creates noise for the query measurements. Unlike Buffered B-Trees all the buffers contain elements, and the number of elements contained in the buffers is proportional to the size of the tree, due to construction being direct insertions on the structure. The low number of queries combined with the elements in the buffers still creates too much noise to determine the true cost of the queries. However, we observe that Figure 7.35 is declining and Figure 7.37 is inclining, thus the true cost of query lies in between the two expectations. The same observation holds for Figures 7.36 and 7.38.

7.4 xDict

In order to experimentally evaluate the xDict data structure we first measure the tradeoff between insertion and query determined by the value of $0 < \alpha \leq 1$. We build up the xDict by direct insertion. We perform this initial test at only $N = 10$ million elements to verify the structures correctness.

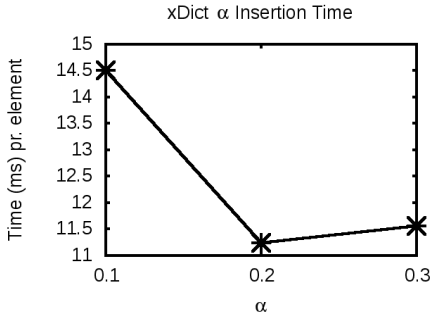


Figure 7.39: Insertion time pr. element.

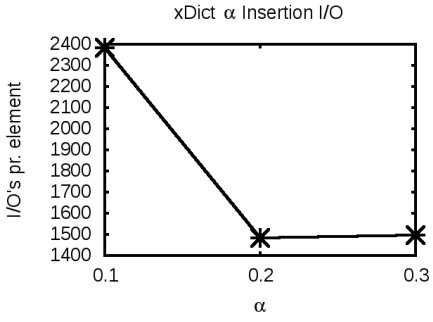


Figure 7.40: Insertion I/O's pr. element.

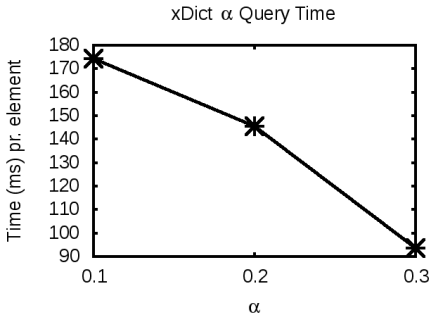


Figure 7.41: Query time pr. element.

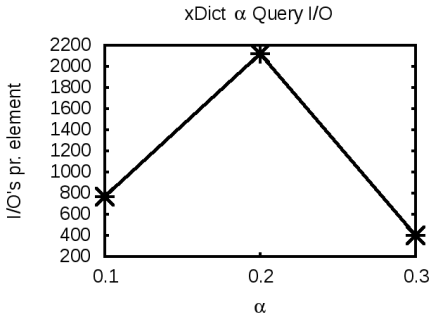


Figure 7.42: Query I/O's pr. element

Notice that Figure 7.39 to 7.42 cuts α of at 0.3. The file system limits the virtual address space to 16TB, which we exceed for $N = 10$ million and $\alpha = 0.4$. We observe that the insertion cost flattens out in Figure 7.39 and 7.40. This is due to the structure quickly constructing an x-box capable of containing all of the insertions. For $\alpha = 0.1$ the xDict pushes the elements through 33 x-boxes before it settles the elements in a single x-box. For $\alpha = 0.2$ the height is 18, and $\alpha = 0.3$ the height is 12. Consistent with theory we observe in Figure 7.41 that the query time goes down as α goes up. Figure 7.42 shows that the query I/O's does not correspond to the query time. A possible explanation is the low value of N we conduct the test at, meaning large parts of the structure is cached.

Mindful of the restriction imposed on the virtual address space by the file system we proceed the testing of the xDict data structure with $\alpha = 0.2$, since this value offers the best tradeoff between insertion time and virtual space usage. For $N = 800$ million elements we will expect a virtual address space of roughly 136GB, all constants included.

This is where the experiment ran into trouble. Memory mapping will utilize all of the available memory. When a background task, such as the kernel will continuously run, requires memory, the logical step is to wait for the memory mapping to release memory, by writing to the mapped file. This logical solution is the outcome the majority of the time, but the system might falsely determine itself out of memory and run a check on each process instead. This check will observe the memory mapping utilizing most of the non-kernel memory, as well as our doubly exponential virtual space, which equates to a high kill priority, and accordingly kill the process. Stack traces of the system at the point in time where the process was killed confirmed it was the kernel request for memory, most often a fork, leading to the OOM killer procedure being run. To quote [mem] directly "Unfortunately, it is possible that the system is not out memory and simply needs to wait for IO to complete or for pages to be swapped to backing storage." We attempted to rectify the issue by giving the system 16MB additional free memory, which allowed us to complete tests with $N = 100$ million elements. To reach $N = 200$ million elements, and thus a longer running time in which it could experience the issue, the server needed 256MB of memory, with 93MB available to the process. Still, upon checking whether the tests

completed via. SSH we would kill the process with the memory request needed to handle the login. Four attempts at $N = 400$ million elements thus failed, each being given a day to complete, and we abandoned this line of experiments.

The experiments so far have been exploratory experiments where we increase N exponentially. We switched to linear experiments where we document the data structure across a linear range of N. This type of experiment is better at documenting the behavior of a structure, at the price of a narrow range of coverage. We conducted two types of experiments: With the server restricted to 128MB (16MB free) of memory we linearly explored the range of $N = 1$ million elements to $N = 50$ million elements, in steps of one million elements, and with the server restricted to 256MB (93MB free) of memory we linearly explored the range of $N = 10$ million elements to $N = 200$ million elements, in steps of ten million elements.

Theorem 6.7 stipulates the amortized insertion cost of the xDict to be $\mathcal{O}(\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)})$. We divide by the expected cost to determine if it converges to a constant.

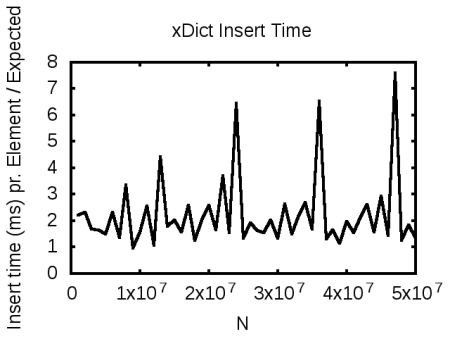


Figure 7.43: Insertion time pr. element divided by expected $\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)}$ on an xDict with 128MB system memory.

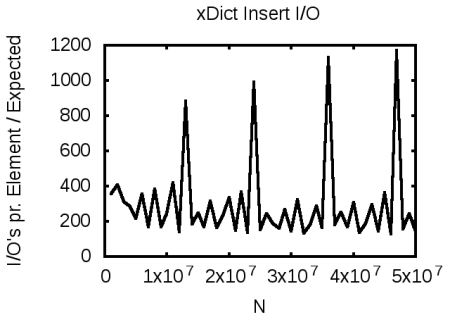


Figure 7.44: Insertion I/O's pr. element divided by expected $\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)}$ on an xDict with 128MB system memory.

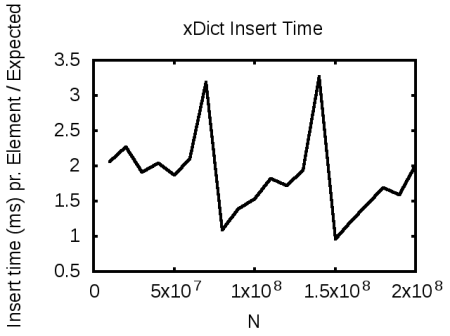


Figure 7.45: Insertion time pr. element divided by expected $\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)}$ on an xDict with 256MB system memory.

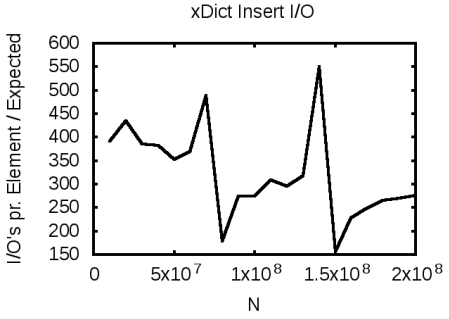


Figure 7.46: Insertion I/O's pr. element divided by expected $\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)}$ on an xDict with 256MB system memory.

Figures 7.43 to 7.46 shows that the structure behaves exactly as we would expect. The insertion cost moves in waves, the peaks of which are when we create a new x-box in the xDict. The lows that follows such peaks is due to the top x-boxes of the xDict being empty, thus allowing for cheap insertions. The noise in between the peaks are the elements being pushed down the height of the xDict in steps. Notice that our granularity of measurements does not capture the exact time of peaks and lows, creating a variance in the lows particularly.

The insertion time does not follow the insertion I/O perfectly. When we insert elements into the top of the xDict they will be free in terms of I/O operations, but will still incur a cost in time. Generally we can see the time and I/O cost peak at the same intervals, and the structure is thus I/O bound.

Theorem 6.6 stipulates the query cost of the xDict to be $\mathcal{O}(\frac{1}{\alpha}\log_B(\frac{N}{M}))$. We divide by the expected cost to determine if it converges to a constant.

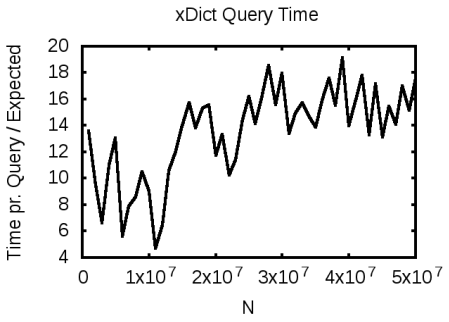


Figure 7.47: Query time pr. element divided by expected $\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)}$ on an xDict with 128MB system memory.

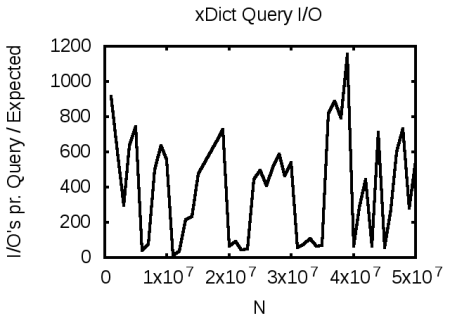


Figure 7.48: Query I/O's pr. element divided by expected $\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)}$ on an xDict with 128MB system memory.

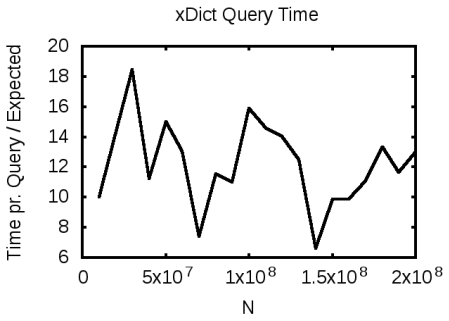


Figure 7.49: Query time pr. element divided by expected $\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)}$ on an xDict with 256MB system memory.

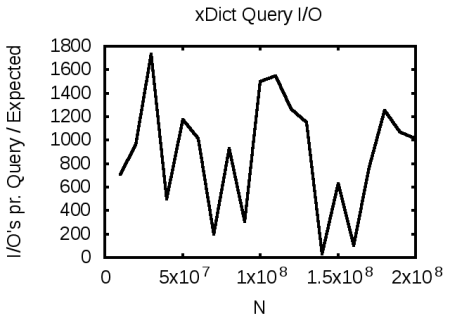


Figure 7.50: Query I/O's pr. element divided by expected $\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)}$ on an xDict with 256MB system memory.

Figures 7.47 to 7.50 shows a seemingly random cost of queries. However, we can reason that the query cost follows the underlying structure of the xDict

closely. When the elements in the xDict is pushed down into the last x-box the query will effectively follow a single path down to the last x-box. The path might diverge slightly from the single path, but in practice only to a constant. This path we cache in memory, and thus we arrive, for free I/O wise, at the last x-box, with a pointer into it that allows for an almost direct lookup into the correct subbox. Any additional elements inserted into the topmost part of the xDict we also query for free, I/O wise. Thus we, generally speaking, have two cases. One case where the elements of the xDict are located at the top and bottom, allowing for cheap queries, and another case where the elements are distributed throughout the xDict, meaning we can not utilize the cached memory of one query in the next query to the same effect. The Figures 7.48 and 7.50 shows this almost binary behavior. The time measured in Figures 7.47 and 7.49 do not show the same behavior as clearly as the measurements of I/O, as traversing the height of the structure in cached memory still requires time.

The structure is I/O bound for the query, as the peaks of time and I/O occurs at the same points of measurement. This is more evident in Figures 7.49 and 7.50 than in Figures 7.47 and 7.48. The cause is the increased size of M for the former query. The increase in depth of the structure containing N =200 million elements over the structure containing $N = 50$ million elements is less than the increase in M, thus allowing for additional caching of query paths down the height of the structure.

7.5 Comparison of Data Structures

We compare the time for insertions and query on the three cache aware data structures directly.

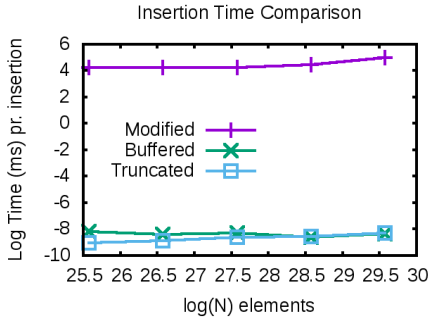


Figure 7.51: Direct comparison of insertion time for the cache aware structures.

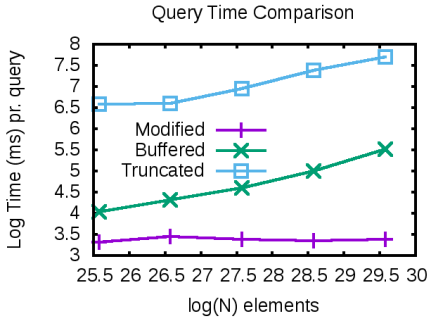


Figure 7.52: Direct comparison of query time for the cache aware structures.

Figure 7.51 shows, as expected, that insertions on the Modified B-Tree are more costly than insertion on the other structures, consistent with theory. The insertion time of the Buffered B-Tree is slower in theory than insertions on the Truncated Buffertree. However, in practice we inserted too few elements in the Buffered B-Tree for the true cost to be properly measured, as the inserted

elements never filled out the trees buffers.

Figure 7.52 shows the ordering of query time in practice follows the theoretical ordering between the three cache aware structures.

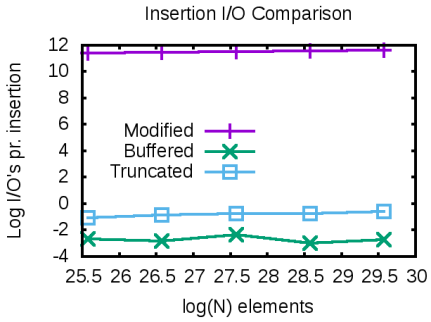


Figure 7.53: Direct comparison of insertion I/O for the cache aware structures.

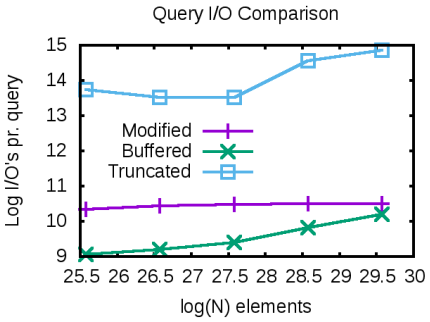


Figure 7.54: Direct comparison of query I/O for the cache aware structures.

Figure 7.53 shows the same trend as in Figure 7.51, that as expected insertions on the Modified B-Tree are more costly than insertion on the other structures, consistent with theory. Unlike Figure 7.51 we see in Figure 7.53 Truncated Buffer Tree being less costly than Buffered B-Tree. Consequently we spend more time pr. I/O performing calculations on the fetched data in a Truncated Buffer Tree than we do in a Buffered B-Tree.

In Figure 7.54 we observe that the relative query I/O cost amongst the three data structures is not consistent with theory. However, we also observe that the increasing tendency in Buffered B-Tree looks to correct this for larger N. Relative to Figure 7.52 we observe that Buffered B-Tree is inefficient with regards to time spent pr. I/O operation compared to Modified B-Tree. This suggests that the buffers of the Buffered B-Tree are expensive to process time wise, the buffers being the main difference between the two structures.

Next we compare the cache oblivious xDict to the cache aware Modified B-Tree. The test did not complete before the end of the thesis, and thus we only provide partial data.

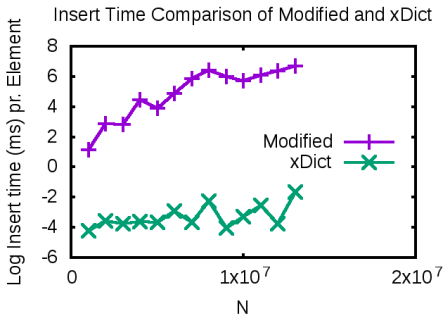


Figure 7.55: Direct comparison of insertion time.

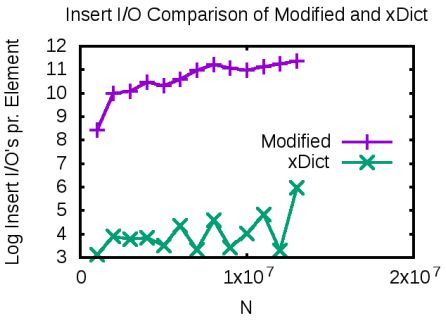


Figure 7.56: Direction comparison of insertion I/O.

Figures 7.55 and 7.56 shows the insertions on the xDict to be less costly than on the Modified B-Tree. The tradeoff parameter $\alpha = 0.2$ tuned the xDict for faster updates, and the result is as expected.

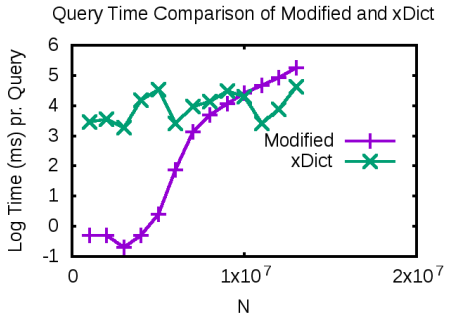


Figure 7.57: Direct comparison of query time.

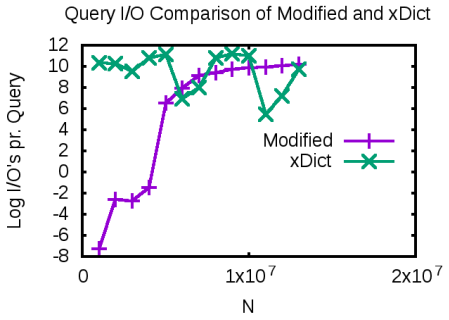


Figure 7.58: Direct comparison of query I/O.

Figure 7.57 shows a faster query time for the xDict over the Modified B-Tree. This result is contrary to theory. To explain the result we turn to Figure 7.58 which shows the I/O cost to be roughly equal once the Modified B-Tree goes into external memory. The cache oblivious structure remains external throughout the graphs, due to an increased element size. The cache oblivious structure is theoretically optimal for all layers of the memory hierarchy, not just the two layers we are observing. Thus it will perform less work pr. I/O in the lower levels of the memory hierarchy. This results in a faster query time.

Figure 7.57 allows us to draw direct comparisons to Figure 7.52. Granted the range of N does not overlap, however, we can make the conjecture that it is likely that xDict queries faster than all the cache aware data structures.

8
Conclusion

In this thesis we described the theory of the data structures from [BF03a] and [BDF+10] in Sections 5 and 6 respectively. We implemented the structures and experimentally evaluated and compared them in Section 7.

The main result of the thesis was documenting that the measured time and I/O operations for insertion and query on the xDict follows the expected cost described by the theory, which is $\mathcal{O}(\frac{1}{\alpha}\log_B(\frac{N}{M})/B^{1/(1+\alpha)})$ and $\mathcal{O}(\frac{1}{\alpha}\log_B\frac{N}{M})$ respectively. This is not a given, as the Cache Oblivious Model makes several assumptions that may not hold true in practice, see Section 2.2. We were not able to properly evaluate the tradeoff constant α for large N, as the underlying file system did not support the virtual space required.

Secondary results was the experimental evaluation of the Modified B-Tree, the Buffered B-Tree, and the Truncated Buffer Tree, and subsequent comparison between the cache aware structures, and the incomplete comparison between the cache oblivious xDict and the cache aware Modified B-Tree.

The direct comparison between the xDict and Modified B-Tree, although incomplete, showed that the cache oblivious structure is competitive. The cache oblivious structure performed less work (time) pr. I/O in our test, suggesting it is more efficient in the lower layers of the memory hierarchy.

The otherwise thorough testing of Modified B-Tree contained a memory leak, and subsequently the time schedule only left room for a single test. This single test showed insertion cost to converge to between $\mathcal{O}(\log_B(\frac{N}{M}))$ and $\mathcal{O}(\log_B(N))$ pr. insertion. The same test showed query cost to remain nearly constant. Further testing is needed to properly evaluate the Modified B-Tree.

The Buffered B-Tree evaluation showed that insertion cost converges to $\mathcal{O}(\delta \cdot \frac{1}{B})$ pr. insertion, as described in theory. We did however not perform enough insertions to fill out the buffers of the tree, which created noise on the query cost. Consequently the query cost did not converge to the expected time, nor did the implemented special query speed up our query time. For the tradeoff parameter δ the tradeoff showed insertion cost go up for larger values of δ, and the query cost need testing on larger values of N to be properly evaluated.

The Truncated Buffer Tree evaluation showed the need to test on larger values of N to properly evaluate the tradeoff parameter δ. We therefore proceeded with $\delta = 4$, to evaluate the structure as a Buffer Tree with extended leafs. The

insertion cost is possibly converging to the expected $\mathcal{O}(\frac{1}{B}\log_{M/B}(\frac{N}{B}))$ pr. insertion. Further testing is needed to conclusively remove noise. The evalutation of the query cost showed that the extended leaf size influenced query cost such that they did not converge to expected $\mathcal{O}(\frac{M}{B}\log_{M/B}(\frac{N}{B}))$ pr. query, nor did we reach a number of insertions sufficient for the fractional cascading in the leafs to dominate the cost of the query, which is consistent with theory as this requires $N \geq M \cdot (\frac{M}{B})^{c\delta}$ for $c > 1$, a value we never reached.

Comparing the three cache aware structures directly we noted that the relative insertion and query time for Modified B-Tree to the other structures behaved as the theory describes. The relative query cost between Buffered B-Tree and Truncated Buffer Tree was likewise consistent with theory, however, their insertion cost remained almost identical. This is due to too few insertions in the test setup for Buffered B-Tree. Further testing is thus needed to properly compare the insertion costs of the two structures directly.

Comparing the query I/O cost of the three cache aware structures the evaluation showed we did not reach a large enough value of N. The insertion I/O cost showed that contrary to theory the Buffered B-Tree required fewer I/O's than the Truncated Buffer Tree. This is due to an insufficient number of insertions in the test setup for the Buffered B-Tree.

8.1 Future Work

Outlined in the above conclusion is the need for additional testing on several structures to remove noise or expand the evaluation to larger values of N, particularly to properly evaluate the tradeoff parameters.

It would be of interest to measure the performance of the xDict structure on other levels in the memory hierarchy than the movement of data between internal memory and harddisk drives.

Similarly it would be of interest to confirm the conjecture that the wave like behavior of insertion of an xDict observed in Section 7 is caused by an increase in the number of x-boxes in the structure, and likewise confirm that the "binary" behavior of the query on an xDict is caused by the allocation of elements to the top and bottom of the xDict.

Bibliography

[AADHM03] Pankaj K. Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley. Cache-oblivious Data Structures for Orthogonal Range Searching. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, SCG '03, pages 237–245, New York, NY, USA, 2003. ACM. doi:10.1145/777792.777828.

[ABD+02] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and James I. Munro. Cache-oblivious Priority Queue and Graph Algorithm Applications. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 268–276, New York, NY, USA, 2002. ACM. doi:10.1145/509907.509950.

[AH74] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[Arg95] Lars Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms (Extended Abstract). In *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, WADS '95, pages 334–345, London, UK, 1995. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645930.672850.

[ASV88] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

[BBF+11] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Dongdong Ge, Simai He, Haodong Hu, John Iacono, and Alejandro López-Ortiz. The Cost of Cache-Oblivious Searching. *Algorithmica*, 61(2):463–505, Oct 2011. doi:10.1007/s00453-010-9394-0.

[BDF+10] Gerth S. Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and James I. Munro. Cache-oblivious Dynamic Dictionaries with Update/Query Tradeoffs. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1448–1456, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1873601.1873718.

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000. doi:10.1109/SFCS.2000.892128.

[BDIW02] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A Locality-preserving Cache-oblivious Dynamic Dictionary. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 29–38, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=545381.545385.

[BF02a] Gerth S. Brodal and Rolf Fagerberg. Cache Oblivious Distribution Sweeping. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 426–438, London, UK, UK, 2002. Springer-Verlag. doi:10.1007/3-540-45465-9_37.

[BF02b] Gerth S. Brodal and Rolf Fagerberg. Funnel Heap - A Cache Oblivious Priority Queue. In *Proceedings of the 13th International Symposium on Algorithms and Computation*, ISAAC '02, pages 219–228, London, UK, UK, 2002. Springer-Verlag. doi:10.1007/3-540-36136-7_20.

[BF03a] Gerth S. Brodal and Rolf Fagerberg. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. URL: http:/doi.acm.org/10.1145/644108.644201.

[BF03b] Gerth S. Brodal and Rolf Fagerberg. On the Limits of Cache-obliviousness. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 307–315, New York, NY, USA, 2003. ACM. doi:10.1145/780542.780589.

[BF06] Gerth S. Brodal and Rolf Fagerberg. Cache-oblivious String Dictionaries. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 581–590, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1109557.1109621.

[BFCF+07] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious Streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM. doi:10.1145/1248377.1248393.

[BFJ02] Gerth S. Brodal, Rolf Fagerberg, and Riko Jacob. Cache Oblivious Search Trees via Binary Trees of Small Height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 39–48, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. URL: https://dl.acm.org/citation.cfm?id=545386.

[BFM05] Gerth S. Brodal, Rolf Fagerberg, and Gabriel Moruz. Cache-Aware and Cache-Oblivious Adaptive Sorting. In *Automata, Languages and Programming*, pages 576–588, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11523468_47.

[BFV08] Gerth S. Brodal, Rolf Fagerberg, and Kristoffer Vinther. Engineering a Cache-oblivious Sorting Algorithm. *J. Exp. Algorithmics*, 12:2.2:1–2.2:23, June 2008. doi:10.1145/1227161.1227164.

[BKR12] Gerth S. Brodal and Casper Kejlberg-Rasmussen. Cache-Oblivious Implicit Predecessor Dictionaries with the Working-Set Property. In *29th International Symposium on Theoretical Aspects of Computer Science (STACS 2012)*, volume 14 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 112–123, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.STACS.2012.112.

[BKRT10] Gerth S. Brodal, Casper Kejlberg-Rasmussen, and Jakob Truelsen. A Cache-Oblivious Implicit Dictionary with the Working Set Property. In *Algorithms and Computation*, pages 37–48, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-17514-5_4.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972. doi:10.1007/BF00288683.

[CG86a] Bernard. Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986. doi:10.1007/BF01840440.

[CG86b] Bernard. Chazelle and Leonidas J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986. doi:10.1007/BF01840441.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[CR72] Stephen A. Cook and Robert A. Reckhow. Time-bounded Random Access Machines. In *Proceedings of the Fourth Annual ACM*

Symposium on Theory of Computing, STOC '72, pages 73–80, New York, NY, USA, 1972. ACM. doi:10.1145/800152.804898.

[ext] ext4 Extent Tree. URL: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Extent_Tree.

[FG03] Gianni Franceschini and Roberto Grossi. Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees. In *Algorithms and Data Structures*, pages 114–126, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-45078-8_11.

[FLPR99] 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*, FOCS '99, pages 285–288, Washington, DC, USA, 1999. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=795665.796479.

[IP12] John Iacono and Mihai Pătraşcu. Using Hashing to Solve the Dictionary Problem. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 570–582, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=2095116.2095164.

[mem] Memory Rules of the Linux Kernel. URL: https://www.kernel.org/doc/gorman/html/understand/understand016.html.

[MH82] Kurt Melhorn and Scott Huddleston. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982. doi:10.1007/BF00288968.

[pro] proc/diskstats on Linux. URL: https://www.kernel.org/doc/Documentation/ABI/testing/procfs-diskstats.

[Pro99] Harald Prokop. Cache-Oblivious Algorithms. Master's Thesis, Massachusetts Institute of Technology, 1999. URL: http://supertech.csail.mit.edu/papers/Prokop99.pdf.

[RS98] Martin Raab and Angelika Steger. "Balls into Bins" - A Simple and Tight Analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '98, pages 159–170, London, UK, UK, 1998. Springer-Verlag. doi:10.1007/3-540-49543-6_13.

[Sib99] Jop F. Sibeyn. External Selection. In *Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science*, STACS'99, pages 291–301, Berlin, Heidelberg, 1999. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1764891.1764929.

[Tar85] Robert E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985. doi:10.1137/0606031.

A
Additional Graphs

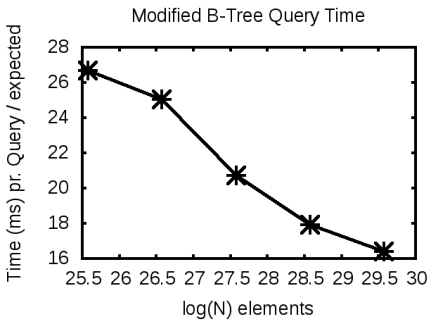


Figure A.1: Time pr. Query divided with expected $c \cdot \log_{2B}(\frac{N}{M})$, for Modified B-Trees before fixing the memory leak.

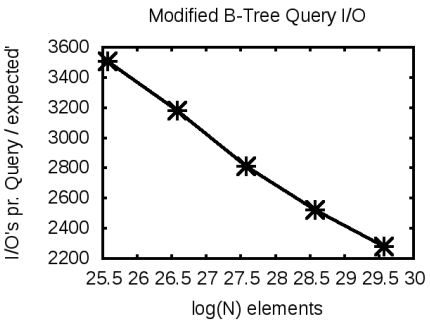


Figure A.2: I/O's pr. query divided with expected $c \cdot \log_{2B}(\frac{N}{M})$ I/O's, for Modified B-Trees before fixing the memory leak.

B
Technical Information

B.1 Test Machine

The test machine used in the thesis is named Raleigh and is located in the MADALGO server room.

The software installed on Raleigh is Linux Kernel 3.16.0-31 and Ubuntu 14.04.1. The relevant hardware is as follows. CPU - Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz - 4 physical and 8 logical cores. L1 32KB, L2 256KB, L3 8MB. The machine contains several hard disk drives. The harddisk drive utilized in the thesis has a minimum block size of 4096 bytes.

The machine was booted in recovery mode with the memory restricted, typically 128MB, of which 16MB is available to the user. There is no swap space available to the operating system, granting us more control of the memory utilized.

The programs where compiled on a separate x86_64 machine running Ubuntu 16.04.9, using GCC version 5.4.0 and CMAKE version 3.8. The libraries (standard only) where statically compiled with the program.

B.2 I/O Data Collection

I/O data is collected from the "file" */proc/diskstats*. The kernel conveniently allows access to disk I/O data through this file like API. The data is represented with a line for each disk containing the following data [pro]:

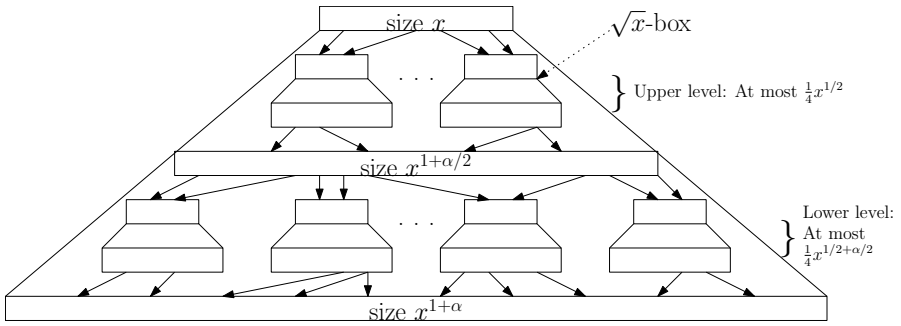
1. major number
2. minor mumber
3. device name
4. reads completed successfully
5. reads merged
6. **sectors read**

7. time spent reading (ms)
8. writes completed
9. writes merged
10. **sectors written**
11. time spent writing (ms)
12. I/Os currently in progress
13. time spent doing I/Os (ms)
14. weighted time spent doing I/Os (ms)

Highlighted is the parameters we use to measure I/O performance.

C
x-box Survival Guide

This appendix includes an overview of the x-box structure for easy reference.



Buffer	Size per buffer	Number of buffers	Total size
Input Buffer	x	1	x
Input Buffer	$\sqrt{x}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x$
Middle Buffer	$(\sqrt{x})^{1+\alpha/2}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x^{1+\alpha/4}$
Output Buffer	$(\sqrt{x})^{1+\alpha}$	$\frac{1}{4}x^{1/2}$	$\frac{1}{4}x^{1+\alpha/2}$
Middle Buffer	$x^{1+\alpha/2}$	1	$x^{1+\alpha/2}$
Input Buffer	$\sqrt{x}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+\alpha/2}$
Middle Buffer	$(\sqrt{x})^{1+\alpha/2}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+3\alpha/4}$
Output Buffer	$(\sqrt{x})^{1+\alpha}$	$\frac{1}{4}x^{1/2+\alpha/2}$	$\frac{1}{4}x^{1+\alpha}$
Output Buffer	$x^{1+\alpha}$	1	$x^{1+\alpha}$

Size of x-box buffers and first recursive layer.

Space: The total space usage of an x-box is at most $cx^{1+\alpha}$ for some constant $c > 0$. Consequently $x^{1+\alpha} = \mathcal{O}(B) \implies x = \mathcal{O}(B^{1/(1+\alpha)})$
Search: For $x > B$ a SEARCH in an x-box costs $\mathcal{O}((1+\alpha)\log_B x)$ I/O's.
Flush: For $x^{1+\alpha} > B$ a FLUSH on an x-box costs $\mathcal{O}(x^{1+\alpha}/B)$ I/O's.
Sample-Up: For $x^{1+\alpha} > B$ SAMPLE-UP on an x-box costs $\mathcal{O}(x^{1+\alpha}/B)$ I/O's.
Batch-Insert: A BATCH-INSERT into an x-box with $x > B$ costs an amortized $\mathcal{O}((1+\alpha)\log_B(x)/B^{1/(1+\alpha)})$ I/O's pr. element.