Cache-Oblivious Dynamic Dictionaries with Update/Query Tradeoffs

Gerth Stølting Brodal[*†] Erik D. Demaine[‡†] Jeremy T. Fineman[‡§] John Iacono[¶]
Stefan Langerman[‖] J. Ian Munro[**]

Abstract

Several existing cache-oblivious dynamic dictionaries achieve $O(\log_B N)$ (or slightly better $O(\log_B \frac{N}{M})$) memory transfers per operation, where N is the number of items stored, M is the memory size, and B is the block size, which matches the classic B-tree data structure. One recent structure achieves the same query bound and a sometimes-better amortized update bound of $O\left(\frac{1}{B^{\Theta(1/(\log\log B)^2)}}\log_B N + \frac{1}{B}\log^2 N\right)$ memory transfers. This paper presents a new data structure, the *xDict*, implementing predecessor queries in $O(\frac{1}{\varepsilon}\log_B \frac{N}{M})$ worst-case memory transfers and insertions and deletions in $O\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B \frac{N}{M}\right)$ amortized memory transfers, for any constant ε with $0 < \varepsilon < 1$. For example, the xDict achieves subconstant amortized update cost when $N = M\,B^{o(B^{1-\varepsilon})}$, whereas the B-tree's $\Theta(\log_B \frac{N}{M})$ is subconstant only when $N = o(MB)$, and the previously obtained $\Theta\left(\frac{1}{B^{\Theta(1/(\log\log B)^2)}}\log_B N + \frac{1}{B}\log^2 N\right)$ is subconstant only when $N = o(2^{\sqrt{B}})$. The xDict attains the optimal tradeoff between insertions and queries, even in the broader external-memory model, for the range where inserts cost between $\Omega(\frac{1}{B}\lg^{1+\varepsilon} N)$ and $O(1/\lg^3 N)$ memory transfers.

1 Introduction

This paper presents a new data structure, the xDict, which is the asymptotically best data structure for the dynamic-dictionary problem in the cache-oblivious model.

Memory models. The *external-memory (or I/O) model* [1] is the original model of a two-level memory hierarchy. This model consists of an internal memory of size M and a disk storing all remaining data. The algorithm can transfer contiguous blocks of data of size B to or from disk at unit cost. The textbook data structure in this model is the B-tree [2], a dynamic dictionary that supports inserts, deletes, and predecessor queries in $O(\log_B N)$ memory transfers per operation.

The *cache-oblivious model* [9, 10] arose in particular from the need to model multi-level memory hierarchies. The premise is simple: analyze a data structure or algorithm just as in the external-memory model, but the data structure or algorithm is not explicitly parametrized by M or B. Thus the analysis holds for an arbitrary M and B, in particular all the M's and B's encountered at each level of the memory hierarchy. The algorithm could not and fortunately does not have to worry about the block replacement strategy because the optimal strategy can be simulated with constant overhead. This lack of parameterization has let algorithm designers develop elegant solutions to problems by finding the best ways to enforce data locality.

Comparison of cache-oblivious dictionaries. Refer to Table 1. In the cache-oblivious model, Prokop's static search structure [10] was the first to support predecessor searches in $O(\log_B N)$ memory transfers, but it does not support insertion or deletion. Cache-oblivious B-trees [3, 4, 7] achieve $O(\log_B N)$ memory transfers for insertion, deletion, and search. The shuttle tree [5] supports insertions and deletions in amortized $O\left(\frac{1}{B^{\Theta(1/(\log\log B)^2)}}\log_B N + \frac{1}{B}\log^2 N\right)$ memory transfers, which is an improvement over $\Theta(\log_B N)$ for $N = 2^{o(B/\log B)}$, while preserving the $O(\log_B N)$ query bound.[1] Our xDict reduces the insertion and deletion bounds further to $O\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B \frac{N}{M}\right)$, for any constant $0 < \varepsilon \le 1$, under the *tall-cache assumption* (common to many cache-oblivious algorithms) that $M = \Omega(B^2)$. For all of these data structures, the query bounds are worst case and the update bounds are amortized.

* Department of Computer Science, Aarhus University, IT-parken, Åbogade 34, DK-8200 Århus N, Denmark, gerth@cs.au.dk

† Supported in part by MADALGO — Center for Massive Data Algorithmics — a Center of the Danish National Research Foundation.

‡ MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA, edemaine@mit.edu, jfineman@cs.cmu.edu

§ Supported in part by NSF Grants CCF-0541209 and CCF-0621511, and Computing Innovation Fellows.

¶ Department of Computer Science and Engineering, Polytechnic Institute of New York University, 5 MetroTech Center, Brooklyn, NY 11201, USA, http://john.poly.edu

‖ Maître de recherches du F.R.S.-FNRS, Computer Science Department, Université Libre de Bruxelles, CP212, Boulevard du Triomphe, 1050 Bruxelles, Belgium, stefan.langerman@ulb.ac.be

** Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, imunro@uwaterloo.ca

[1] For these previous data structures, the $\log_B N$ terms may well reduce to $\log_B \frac{N}{M}$ terms, but only $\log_B N$ was explicitly proven.

Data Structure	Search	Insert/Delete
static search [10]	$O(\log_B N)$	not supported
B-trees [3, 4, 7]	$O(\log_B N)$	$O(\log_B N)$
shuttle tree [5]	$O(\log_B N)$	$O\left(\frac{1}{B^{\Theta(1/(\log\log B)^2)}}\log_B N + \frac{1}{B}\log^2 N\right)$
lower bound [6]	$O(\frac{1}{\varepsilon}\log_B \frac{N}{M})$	$\implies \Omega\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B \frac{N}{M}\right)$
xDict [this paper]	$O(\frac{1}{\varepsilon}\log_B \frac{N}{M})$	$O\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B \frac{N}{M}\right)$

Table 1: Summary of known cache-oblivious dictionaries. Our xDict uses the tall-cache assumption that $M = \Omega(B^2)$.

Lower bounds. The tradeoff between insertion and search costs was partly characterized in the external-memory model [6]. Because any algorithm in the cache-oblivious model is also an algorithm in the external-memory model using the same number of memory transfers, lower bounds for the external-memory model carry over to the cache-oblivious model. Brodal and Fagerberg [6] proved that any structure supporting insertions in $I = O\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B(N/M)\right)$ amortized memory transfers, when I is between $\Omega(\frac{1}{B}\lg^{1+\varepsilon} N)$ and $O(1/\lg^3 N)$ and when $N \geq M^2$, must use $\Omega(\frac{1}{\varepsilon}\log_B(N/M))$ worst-case memory transfers for search. They also produced a data structure achieving this tradeoff, but their structure is not cache-oblivious, using knowledge of the parameters B and M. The xDict structure achieves the same tradeoff in the cache-oblivious model, and is therefore optimal for this range of insertion cost I. Slightly outside this range, the optimal bounds are not known, even in the external-memory model.

2 Introducing the x-box

Our xDict dynamic-dictionary data structure is built in terms of another structure called the x-box. For any positive integer x, an *x-box* supports a batched version of the dynamic-dictionary problem (defined precisely later) in which elements are inserted in batches of $\Theta(x)$. Each x-box will be defined recursively in terms of y-boxes for $y < x$, and later we build the overall xDict data structure in terms of x-boxes for x increasing doubly exponentially. Every x-box uses a global parameter, a real number $\alpha > 0$, affecting the insertion cost, with lower values of α yielding cheaper insertions. This parameter is chosen globally and remains fixed throughout.

As shown in Figure 1, an x-box is composed of three buffers (arrays) and many $\sqrt{x}$-boxes, called *subboxes*. The three buffers of an x-box 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 $\sqrt{x}$-subboxes of an x-box are divided into two levels: the *upper level* consists of at most $\frac{1}{4}x^{1/2}$ subboxes, and the *lower level* consists of at most $\frac{1}{4}x^{1/2+\alpha/2}$ subboxes. Thus, in total, there are fewer than $\frac{1}{2}x^{1/2+\alpha/2}$ subboxes. See Table 2 for a table of buffer counts and sizes. As a base case, an $O(1)$-box consists of a single array that acts as both the input and output buffers, with no recursive subboxes or middle buffer.

Logically, the upper-level subboxes are children of the input buffer and parents of the middle buffer. Similarly, the lower-level subboxes are children of the middle buffer and parents of the output buffer. However, the buffers and subboxes do not necessarily form a tree structure. Specifically, for an x-box D, there are pointers from D's input buffer to the input buffers of its upper-level subboxes. Moreover, there may be many pointers from D's input buffers to a single subbox. There are also pointers from the output buffers of the upper-level subboxes to D's middle buffer. Again, there may be many pointers originating from a single subbox. Similarly, there are pointers from D's middle buffer to its lower-level subboxes' input buffers, and from the lower-level subboxes' output buffers to D's output buffers.

The number of subboxes in each level has been chosen to match the buffer sizes. Specifically, the total size of the input buffers of all subboxes in the upper level is at most $\frac{1}{4}x^{1/2} \cdot x^{1/2} = \frac{1}{4}x$, which is a constant factor of the size of the x-box's input buffer. Similarly, the total size of the upper-level subboxes' output buffers is at most $\frac{1}{4}x^{1/2} \cdot (x^{1/2})^{1+\alpha} = \frac{1}{4}x^{1+\alpha/2}$, which matches the size of the x-box's middle buffer. The total size of the lower-level subboxes' input and output buffers are at most $\frac{1}{4}x^{1/2+\alpha/2} \cdot x^{1/2} = \frac{1}{4}x^{1+\alpha/2}$ and $\frac{1}{4}x^{1/2+\alpha/2} \cdot (x^{1/2})^{1+\alpha} = \frac{1}{4}x^{1+\alpha}$, which match the sizes of the x-box's middle and output buffers, respectively, to within a constant factor.

An x-box D organizes elements as follows. Suppose that the keys of elements contained in D range from $[\kappa_{\min}, \kappa_{\max})$. The elements located in the input buffer occur in sorted order, as do the elements located in the middle buffer and the elements located in the output buffer. All three of these buffers may contain elements having any keys between $\kappa_{\min}$ and $\kappa_{\max}$. The upper-level subboxes, however, partition the key space. More precisely, suppose that there are r upper-level subboxes. Then there exist keys $\kappa_{\min} = \kappa_0 < \kappa_1 < \cdots < \kappa_r = \kappa_{\max}$ such that each subbox contains elements in a distinct range $[\kappa_i, \kappa_{i+1})$. Similarly, the lower-level subboxes partition the key space. There is, however, no relationship between the partition imposed by the upper-level subboxes and that of the lower-level subboxes; the subranges are entirely unrelated. What this setup

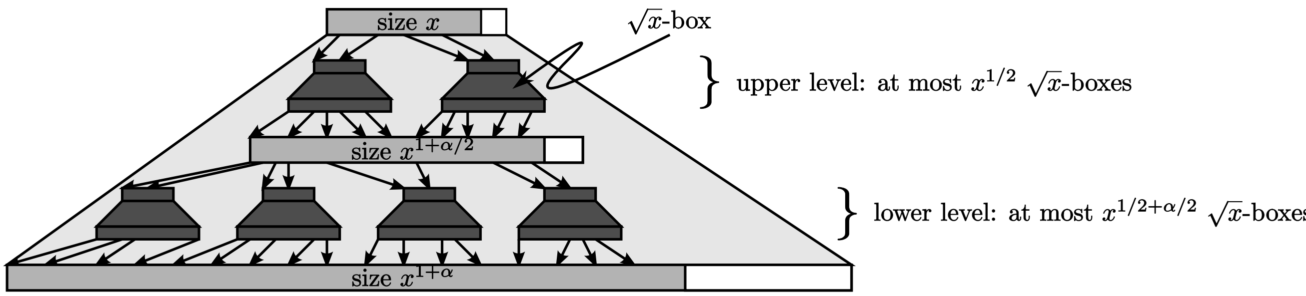

Figure 1: The recursive structure of an x-box. The arrows represent lookahead pointers. These lookahead pointers are evenly distributed in the target buffer, but not necessarily in the source buffer. Additional pointers (not shown) allow us to find the nearest lookahead pointer(s) in $O(1)$ memory transfers. The white space at the right of the buffers indicates empty space, allowing for future insertions.

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

Table 2: Sizes of buffers in an x-box. This table lists the sizes of the three buffers in an x-box and the sizes and number of buffers in its recursive $x^{1/2}$-boxes, expanding just one level of recursion.

means is that an element with a particular key may be located in the input buffer or the middle buffer or the output buffer or a particular upper-level subbox or a particular lower-level subbox. Our search procedure will thus look in all five of these locations to locate the element in question.

Before delving into more detail about the x-boxes, let us first give a rough sketch of insertions into the data structure. When an element is inserted into an x-box D, it is first inserted into D's input buffer. As the input buffer stores elements in sorted order, elements in the input buffer must move to the right to accommodate newly inserted elements. When D's input buffer becomes full (or nearly full), the elements are inserted recursively into D's upper-level subboxes. When an upper-level subbox becomes full enough, it is "split" into two subboxes, with one subbox taking the half of the elements with smaller keys and the other taking the elements with larger keys. When the maximum number of upper-level subboxes is reached, all elements are moved from the upper-level subboxes to D's middle buffer (which stores all elements in sorted order). When the middle buffer becomes full enough, elements are moved to the lower-level subboxes in a similar fashion. When the maximum number of lower-level subboxes is reached, all elements are moved from the lower-level subboxes to D's output buffer.

To aid in pushing elements down to the appropriate subboxes, we embed in the input (and middle) buffers a pointer to each of the upper-level (and lower-level) subboxes. These *subbox pointers* are embedded into the buffers by associating with them the minimum key stored in the corresponding subbox, and then storing them along with the buffer's elements in sorted order by key.

To facilitate searches, we employ the technique of *fractional cascading* [8], giving an x-box's input buffer (and middle buffer) a sample of the elements of the upper-level subboxes' (and lower-level subboxes') input buffers. Specifically, let U be an upper-level subbox of the x-box D. Then a constant fraction of the keys stored in U's input buffer are also stored in D's input buffer. This sampling is performed deterministically by placing every sixteenth element in U's input buffer into D's input buffer. The sampled element (in D's input buffer) has a pointer to the corresponding element in U's input buffer. This type of pointer also occurs in [5], where they are called "lookahead pointers." We adopt the same term here. This sampling also occurs on the output buffers. Specifically, the output buffers of the upper-level (and lower-level) subboxes contain a similar sample of the elements in D's middle buffer (and output buffer).[2]

The advantage of lookahead pointers is roughly as follows. Suppose we are looking for a key κ in some buffer A. Let s be the multiple of 16 (i.e., a sampled element) such that $A[s] \leq \kappa < A[s+16]$. Then our search procedure will scan slots s to $s+16$ in the buffer A. If κ is not located in any of these slots, then it is not in the buffer. Ideally, this scan would also provide us with a good starting point to search within the next buffer.

As the lookahead pointers may be irregularly distributed in D's input (or middle) buffer, a stretch of sixteen elements in D's input (or middle) buffer may not contain a lookahead pointer to an upper (or lower)-level subboxes. To remedy this problem, we associate with each element in D's input (or middle) buffer a pointer to the nearest lookahead or subbox pointers preceding and following it.

Techniques. While fractional cascading has been employed before [5], we are not aware of any previous cache-oblivious data structures that are both recursive and that use fractional cascading. Another subtler difference between the x-box and previous cache-oblivious structures involves the sizing of subboxes and our use of the middle buffer. If the x-box matched typical structures, then an x-box with an input buffer of size x and an output buffer of size $x^{1+\alpha}$ would have a middle buffer of size $x^{\sqrt{1+\alpha}}$, not the $x^{1+\alpha/2}$ that we use. That is to say, it is natural to size the buffers such that a size-x input buffer is followed by a size-x^δ middle buffer for some δ, and a size-$y = x^\delta$ middle buffer is followed by a size-$y^\delta = x^{\delta^2}$ output buffer. Our choice of sizes causes the data structure to be more topheavy than usual, a feature which facilitates obtaining our query/update tradeoff.

3 Sizing an x-box

An x-box stores the following fields, in order, in a fixed contiguous region of memory.

1. A counter of the number of real elements (not counting lookahead pointers) stored in the x-box.
2. The top buffer.
3. Array of booleans indicating which upper-level subboxes are being used.
4. Array of upper-level subboxes, in an arbitrary order.
5. The middle buffer.
6. Array of booleans indicating which lower-level subboxes are being used.
7. Array of lower-level subboxes, in an arbitrary order.
8. The bottom buffer.

In particular, the entire contents of each $\sqrt{x}$-subbox are stored within the x-box itself. In order for an x-box to occupy a fixed contiguous region of memory, we need a upper bound on the maximum possible space usage of a box.

LEMMA 3.1. *The total space usage of an x-box is at most $c\,x^{1+\alpha}$ for some constant $c > 0$.*

Proof. The proof is by induction. An x-box contains three buffers of total size $c'(x+x^{1+\alpha/2}+x^{1+\alpha}) \leq 3c'x^{1+\alpha}$, where c' is the constant necessary for each array entry (including information about lookahead pointers, etc.). The boolean arrays use a total of at most $\frac{1}{2}x^{1/2+\alpha/2} \leq c'x^{1+\alpha}$ space, giving us a running total of $4c'x^{1+\alpha}$ space. Finally, the subboxes by assumption use a total of at most $c(x^{1/2})^{1+\alpha} \cdot \frac{1}{2}x^{1/2+\alpha/2} = \frac{c}{2}x^{1+\alpha}$ space. Setting $c \geq 8c'$ yields a total space usage of at most $cx^{1+\alpha}$. □

4 Operating an x-box

An x-box D supports two operations:

1. BATCH-INSERT$(D, e_1, e_2, \ldots, e_{\Theta(x)})$: Insert $\Theta(x)$ keyed elements $e_1, e_2, \ldots, e_{\Theta(x)}$, given as a sorted array, into the x-box D. BATCH-INSERT maintains lookahead pointers as previously described.
2. SEARCH(D, s, κ): Return a pointer to an element with key κ in the x-box D if such an element exists. If no such element exists, return a pointer to κ's predecessor in D's output buffer, that is, the element located in D's output buffer with the largest key smaller than κ. We assume that we are given the nearest lookahead pointer s preceding κ pointing into D's input buffer: that is, s points to the sampled element (i.e., having an index that is a multiple of 16) in D's input buffer that has the largest key not exceeding κ.

We treat an x-box as *full* or *at capacity* when it contains $\frac{1}{2}x^{1+\alpha}$ real elements. A BATCH-INSERT is thus only allowed when the inserted elements would not cause the x-box to contain more than $\frac{1}{2}x^{1+\alpha}$ elements. Our algorithm does not continue to insert into any recursive x-box when this number of elements is exceeded. The constant can be tuned to waste less space, but we choose $\frac{1}{2}$ here to simplify the presentation. Recall that the output buffer of an x-box has size $x^{1+\alpha}$, which is twice the size necessary to accommodate all the real elements in the x-box. We allow the other

[2] Any sufficiently large sampling constant suffices here. We chose 16 as an example of one such constant for concreteness. The constant must be large enough that, when sampling D's output buffer, the resulting lookahead pointers occupy only a constant fraction of the lower-level subboxes' output buffers. To make the description more concise, the constant fraction we allow is $\frac{1}{4}$, but in fact any constant would work. As the lower-level subboxes' output buffers account for only $\frac{1}{4}$ of the space of D's output buffer, these two constants are multiplied to get that only $\frac{1}{16}$ of the elements in D's output buffer may be sampled.

$\frac{1}{2}x^{1+\alpha}$ space in the output buffer to store external lookahead pointers (i.e., lookahead pointers into a buffer in the containing x^2-box).

4.1 SEARCH. Searches are easiest. The operation SEARCH(D, s, κ) starts by scanning D's input buffer at slot s and continues until reaching an element with key κ or until reaching slot s' where the key at s' is larger than κ. By assumption on lookahead pointers, this scan considers $O(1)$ array slots. If the scan finds an element with key κ, then we are done. Otherwise, we consider the nearest lookahead or subbox pointer preceding slot $s'-1$. We follow this pointer and search recursively in the corresponding subbox. That recursive search returns a pointer in the subbox's output buffer. We again reference the nearest lookahead pointer and jump to a point in the middle buffer. This process continues, scanning a constant-size region in middle buffer, searching recursively in the lower-level subbox, and scanning a constant-size region in the output buffer.

LEMMA 4.1. *For $x > B$, a* SEARCH *in an x-box costs $O((1+\alpha)\log_B x)$ memory transfers.*

Proof. The search considers a constant-size region in the three buffers, for a total of $O(1)$ memory transfers, and performs two recursive searches. Thus, the cost of a search can be described by the recurrence $S(x) = 2S(\sqrt{x}) + O(1)$. Once $x^{1+\alpha} = O(B)$, or equivalently $x = O(B^{1/(1+\alpha)})$, an entire x-box fits in a block, and no further recursions incur any other cost. Thus, we have a base case of $S(O(B^{1/(1+\alpha)})) = 0$.

The recursion therefore proceeds with a nonzero cost for $\lg\lg x - \lg\lg O(B^{1/(1+\alpha)})$ levels, for a total of $2^{\lg\lg x - \lg\lg O(B^{1/(1+\alpha)})} = (1+\alpha)\lg x/\lg O(B) = O((1+\alpha)\log_B x)$ memory transfers. □

4.2 BATCH-INSERT overview. For clarity, we decompose BATCH-INSERT into several operations, including two new auxiliary operations:

1. FLUSH(D): After this operation, all k elements in the x-box D are located in the first $\Theta(k)$ slots of D's output buffer. These elements occur in sorted order. All other buffers and recursive subboxes are emptied, temporarily leaving the D without any internal lookahead pointers (to be fixed later by a call SAMPLE-UP). The $\Theta()$ arises because of the presence of lookahead pointers directed from the output buffer. The FLUSH operation is an auxiliary operation used by the BATCH-INSERT.

2. SAMPLE-UP(D): This operation may only be invoked on an x-box that is entirely empty except for its output buffer (as with one that has just been FLUSHed). The sampling process is employed from the bottom up, creating subboxes as necessary, and placing the appropriate lookahead pointers. The SAMPLE-UP operation is an auxiliary operations used by BATCH-INSERT.

4.3 FLUSH. To FLUSH an x-box, first flush all the subboxes. Now consider the result. Elements can live in only five possible places: the input buffer, the middle buffer, the output buffer, the upper-level subboxes' output buffers, and the lower-level subboxes' output buffers. The elements are in sorted order in all of the buffers. Moreover, as the upper-level (and lower-level) subboxes partition the key space, the collection of upper-level subboxes' output buffers form a fragmented sorted list of elements. Thus, after flushing all subboxes, moving all elements to the output buffer requires just a 5-way merge into the output buffer. A constant-way merge can be performed in a linear number of memory transfers in general, but here we have to deal with the fact that upper-level and lower-level subboxes represent fragmented lists, requiring random accesses to jump from the end of one output buffer to the beginning of another.

When merging all the elements into the output buffer, we merge only real elements, not lookahead pointers (except for the lookahead pointers that already occur in the output buffer). This step breaks the sampling structure of the data structure, which we will later resolve with the SAMPLE-UP procedure.

When the flush completes, the input and middle buffers are entirely empty, and all subboxes are deleted.[3]

LEMMA 4.2. *For $x^{1+\alpha} > B$, a* FLUSH *in an x-box costs $O(x^{1+\alpha}/B)$ memory transfers.*

Proof. We can describe the flush by the recurrence

$$F(x) = O(x^{1+\alpha}/B) + O(x^{1/2+\alpha/2}) + \frac{1}{2}x^{1/2+\alpha/2}F(\sqrt{x}) ,$$

where the first term arises due to scanning all the buffers (i.e., the 5-way merge), the second term arises from the random accesses both to load the first block of any of the subboxes and to jump when scanning the concatenated list of output buffers, and the third term arises due to the recursive flushing. When $x^{1+\alpha} = O(M)$, the second term disappears as the entire x-box fits in memory, and we thus need only pay for loading each block once. When applying a tall-cache assumption that $M = \Omega(B^2)$, it follows that the second term only occurs when $x^{1/2+\alpha/2} = \Omega(B)$, and hence when $x^{1/2+\alpha/2} = x^{1+\alpha}/x^{1/2+\alpha/2} = x^{1+\alpha}/\Omega(B) = O(x^{1+\alpha}/B)$. We thus have a total cost of

$$F(x) \le c_1 x^{1+\alpha}/B + \frac{1}{2}x^{1/2+\alpha/2}F(\sqrt{x}) ,$$

[3] In fact, the subboxes use a fixed memory footprint, so they are simply marked as deleted.

where c_1 is a constant hidden by the order notation.

We next prove that $F(x) \leq cx^{1+\alpha}/B$ by induction. As a base case, when $y^{1+\alpha}$ fits in memory, the cost is $F(y) = cy^{1+\alpha}/B$ as already noted, to load each block into memory once. Applying the inductive hypothesis that $F(y) \leq cy^{1+\alpha}/B$ for some sufficiently large constant c and $y < x$, we have $F(x) \leq c_1x^{1+\alpha}/B + \frac{1}{2}x^{1/2+\alpha/2}(cx^{1/2+\alpha/2}/B) = c_1x^{1+\alpha}/B + \frac{1}{2}cx^{1+\alpha}/B$. Setting $c > 2c_1$ completes the proof. □

4.4 SAMPLE-UP. When invoking a SAMPLE-UP, we assume that the only elements in the x-box live in the output buffer. In this state, there are no lookahead pointers to facilitate searches. The SAMPLE-UP recomputes the lookahead pointers, allowing future searches to be performed efficiently.

The SAMPLE-UP operates as follows. Suppose that the output buffer contains $k < x^{1+\alpha}$ elements. Then we create $(k/16)/(x^{1/2+\alpha/2}/2) = k/8x^{1/2+\alpha/2} \leq x^{1/2+\alpha/2}/8$ new lower-level subboxes. Recall that this is half the number of available lower-level subboxes. We then assign to each of these subboxes $x^{1/2+\alpha/2}/2$ of contiguous sampled pointers (filling half of their respective output buffers), and recursively call SAMPLE-UP in the subboxes. Then, we sample from the lower-level subboxes' input buffers to the middle buffer, and we sample the middle-buffer to upper-level subboxes in a similar fashion. Finally, we sample the upper-level subboxes up to the input buffer.

LEMMA 4.3. *A* SAMPLE-UP *in an x-box, for $x^{1+\alpha} > B$, costs $O(x^{1+\alpha}/B)$.*

Proof. The proof is virtually identical to the proof for FLUSH. The recurrence is the same (in fact, it is better here because we can guarantee that the subboxes are, in fact, contiguous).
□

4.5 BATCH-INSERT. The BATCH-INSERT operation takes as input a sorted array of elements to insert. In particular, when inserting into an x-box D, a BATCH-INSERT inserts $\Theta(x)$ elements. For conciseness, let us say that the constant hidden by the theta notation is $1/2$. First, merge the inserted elements into D's input buffer, and increment the counter of elements contained in D by $(1/2)x$. For simplicity, also remove the lookahead pointers during this step. We will readd them later.

Then, (implicitly) partition the input buffer according to the ranges owned by each of the upper-level subboxes. For any partition containing at least $(1/2)\sqrt{x}$ elements, repeatedly remove $(1/2)\sqrt{x}$ elements from D's input buffer and insert them recursively into the appropriate subbox until the partition contains less than $(1/2)\sqrt{x}$ elements. If performing a recursive insert would cause the number of (real) elements in the subbox to exceed $(1/2)\sqrt{x}^{1+\alpha}$, first "split" the subbox. A "split" entails first FLUSHing the subbox, creating a new subbox, moving the larger half of the elements from the old subbox's output buffer to the new subbox's output buffer (involving two scans), and calling SAMPLE-UP on both subboxes, and updating the counter recording the number of elements in each subbox to reflect the number of real elements in each.

Observe that after all the recursions occur, the number of real elements in D's input buffer is at most $(1/2)\sqrt{x} \cdot (1/4)\sqrt{x} = (1/8)x$, as otherwise more elements would have moved down to a subbox.

After moving elements from the input buffer to the upper-level subboxes, resample from the upper-level subboxes' input buffers into D's input buffer. Note that the number of lookahead pointers introduced into the input buffer is at most $\frac{1}{16}\sqrt{x} \cdot \frac{1}{4}\sqrt{x}$. When combining the number of lookahead pointers with the number of real elements, we see that D's input buffer is far less than half full, and hence it can accommodate the next insertion.

When a split causes the last available subbox to be allocated, we abort any further recursive inserts and instead merge all of D's input-buffer and upper-level subbox elements into D's middle buffer. This merge entails first FLUSHing all the upper-level subboxes, and then merging into the middle buffer (similar to the process for the full FLUSH). We then perform an analogous movement from the middle buffer to the lower-level subboxes, matching the movement from the upper-level subboxes to the middle buffer. (If the last lower-level subbox is allocated, we move all elements from D's middle buffer and lower-level subboxes to D's output buffer and then call SAMPLE-UP on D.) When insertions into the lower-level subboxes complete, we allocate new upper-level subboxes (as in SAMPLE-UP) and sample from D's middle buffer into the upper-level suboxes' output buffers, call SAMPLE-UP recursively on these subboxes, and finally sample from the subboxes' input buffers into D's input buffer.

Observe that the upper-level subboxes' output buffers collectively contain at most $\frac{1}{16}x^{1+\alpha/2}$ lookahead pointers. Moreover, after elements are moved into the middle buffer, these are the only elements in the upper-level subboxes, and they are spread across at most half $(x^{1/2}/8)$ the upper-level subboxes, as specified for SAMPLE-UP. Hence, there must be at least $x^{1/2}/8$ subbox splits between moves into the middle buffer. Because subboxes splits only occur when the two resulting subboxes contain at least $(1/4)x^{1+\alpha/2}$ real elements, it follows that there must be at least $(1/4)x^{1+\alpha/2} \cdot (1/8)x^{1+\alpha/2} = \Omega(x^{1+alpha/2})$ insertions into D between moves into the middle buffer. A similar argument shows that there must be at least $\Omega(x^{1+\alpha})$ insertions into D between insertions into the output buffer.

THEOREM 4.1. *A* BATCH-INSERT *into an x-box, with $x > B$, costs an amortized $O((1+\alpha)\log_B(x)/B^{1/(1+\alpha)})$ memory transfers per element.*

Proof. An insert has several costs per element. First, there is the cost of merging into the input array, which is simply $O(1/B)$ per element. Next, each element is inserted recursively into a top-level subbox. These recursive insertions entail random accesses to load the first block of each of the subboxes. Then these subboxes must be sampled, which is dominated by the cost of the aforementioned random accesses and scans. An element may also contribute to a split of a subbox, but each split may be amortized against $\Omega(x^{1/2+\alpha/2})$ insertions. Then we must also consider the cost of moving elements from the upper-level subboxes to the middle buffer, but this movement may be amortized against the $\Omega(x^{1+\alpha/2})$ elements being moved. Finally, there are similar costs among the lower-level subboxes.

Let us consider the cost of the random accesses more closely. If all of the upper-level subboxes fit into memory, then the cost of random accesses is actually the minimum of performing the random accesses or loading the entire upper-level into memory. For the upper-level, we denote this value by $\mathit{UpperRA}(x)$. We thus have $\mathit{UpperRA}(x) = O(\frac{1}{4}x^{1/2})$ if $x^{1+\alpha/2} = \Omega(M)$, and $\mathit{UpperRA}(x) = O(\min\{\frac{1}{4}x^{1/2}, x^{1+\alpha/2}/B\})$ if $x^{1+\alpha/2} = O(M)$. In fact, we are really concerned with $\mathit{UpperRA}(x)/x$, as the random accesses can be amortized against the x elements inserted. We analyze the two cases separately, and we assume the tall-cache assumption that $M > B^2$.

1. Suppose that $x^{1+\alpha/2} = \Omega(M)$. Then we have $x^{1+\alpha/2} = \Omega(B^2)$ by the tall-cache assumption, and hence $x^{1/2} = \Omega(B^{2/(2+\alpha)})$. It follows that $\mathit{UpperRA}(x)/x = O(1/\sqrt{x}) = O(1/B^{2/(2+\alpha)})$.

2. Suppose that $x^{1+\alpha/2} = O(M)$. We have two subcases here. If $x > B^{2/(1+\alpha)}$, then we have a cost of at most $\mathit{UpperRA}(x)/x = O(x^{1/2}/x) = O(1/B^{1/(1+\alpha)})$. If, on the other hand, $x < B^{2/(1+\alpha)}$, then we have $\mathit{UpperRA}(x)/x = O(x^{1+\alpha/2}/Bx) = O(x^{\alpha/2}/B) = O(B^{\alpha/(1+\alpha)}/B) = O(1/B^{1/(1+\alpha)})$.

Because $1/B^{2/(2+\alpha)} < 1/B^{1/(1+\alpha)}$, we conclude that $\mathit{UpperRA}(x)/x = O(1/B^{1/(1+\alpha)})$.

We must also consider the cost of random accesses into the lower-level subboxes, which can be amortized against the $x^{1+\alpha/2}$ elements moved. A similar case analysis shows that $\mathit{LowerRA}(x)/x^{1+\alpha/2} = O(1/B^{1/(1+\alpha)})$.

We thus have a total insertion cost of

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

As we charge loading the first block of a subbox to an insert into the parent, we have a base case of $I(O(B^{1/(1+\alpha)})) = 0$, i.e., when the x-box fits into a single block. Solving the recurrence, we get a per element cost of $O(1/B^{1/(1+\alpha)})$ at $\lg\lg x - \lg\lg O(B^{1/(1+\alpha)})$ levels of recursion, and hence a total cost of $O(2^{\lg\lg x - \lg\lg O(B^{1/(1+\alpha)})}/B^{1/(1+\alpha)}) = O\left(\frac{(1+\alpha)\lg x}{B^{1/(1+\alpha)}\lg O(B)}\right) = O((1+\alpha)\log_B(x)/B^{1/(1+\alpha)})$. □

5 Building a dictionary out of x-boxes

The xDict data structure consists of $\log_{1+\alpha}\log_2 N + 1$ x-boxes of doubly increasing size, where α is the same parameter for the underlying x-boxes. Specifically, for $0 \le i \le \log_{1+\alpha}\log_2 N$, the ith box has $x = 2^{(1+\alpha)^i}$. The x-boxes are linked together by incorporating into the ith box's output buffer the lookahead pointers corresponding to a sample from the $(i+1)$st box's input buffer.

We can define the operations on an xDict in terms of x-box operations. To insert an element into an xDict, we simply insert it into the smallest x-box ($i = 0$), which has $x = \Theta(1)$ so supports individual element insertions. When the ith box reaches capacity (containing $2^{(1+\alpha)^{i+1}}/2$ elements), we FLUSH it, insert all of its elements (contained in its output buffer) into the $(i+1)$st box, and empty the ith box's output buffer. This process terminates after performing a batch insert into the jth box if the jth box is the first box that can accommodate the elements (having not yet reached capacity). At this point, all boxes preceding the jth box are entirely empty. We next rebuild the lookahead starting from the $(j-1)$st box down to the 0th box by sampling from the $(i+1)$st box's input buffer into the ith box's output buffer and then calling SAMPLE-UP on the ith box. As elements are first inserted into the 0th box and eventually move through all boxes, our insertion analysis accounts for an insertion into each box for each element inserted. The cost of the FLUSH and SAMPLE-UP incurred by each element as it moves through each of the boxes is dominated by the cost of the insertion.

To search in the xDict, we simply search in each of the x-boxes, and return the closest match. Specifically, we search the boxes in order from smallest to largest. If the element is not found in the ith box, we have a pointer into the ith box's output buffer. We use this pointer to find an appropriate lookahead pointer into the $(i+1)$st box's input buffer, and begin the search from that point.

The performance of the xDict is essentially a geometric series:

THEOREM 5.1. *The xDict supports searches in $O(\frac{1}{\alpha}\log_B \frac{N}{M})$ memory transfers and (single-element) inserts in $O(\frac{1}{\alpha}(\log_B \frac{N}{M})/B^{1/(1+\alpha)})$ amortized memory transfers, for $0 < \alpha \le 1$.*

Proof. A simple upper bound on the search cost is

$$\begin{aligned}
& \sum_{i=0}^{\log_{1+\alpha}\log_2 N} O((1+\alpha)\log_B(2^{(1+\alpha)^i}) \\
&= O\left(\frac{1+\alpha}{\lg B}\sum_{i=0}^{\log_{1+\alpha}\log_2 N}(1+\alpha)^i\right) \\
&= O\left(\frac{(1+\alpha)\lg N}{\lg B}\sum_{i=0}^{\infty}\frac{1}{(1+\alpha)^i}\right) \\
&= O\left(\frac{(1+\alpha)^2}{\alpha}\log_B N\right) \\
&= O\left(\frac{1}{\alpha}\log_B N\right).
\end{aligned}$$

The last step of the derivation follows from the assumption that $\alpha \le 1$ and hence that $(1+\alpha)^2 = O(1)$.

The above analysis, however, exploits only a constant number of cache blocks. If we assume that the memory already holds all x-boxes smaller than $O(M^{1/(1+\alpha)})$,[4] the first $O(\frac{1}{\alpha}\log_B M^{1/(1+\alpha)}) = O(\frac{1}{\alpha}\log_B M)$ transfers are free, resulting in a search cost of $O(\frac{1}{\alpha}\log_B \frac{N}{M})$. The cache-oblivious model assumes an optimal paging strategy, and using half the memory to store the smallest x-boxes is no better than optimal.

The analysis for insertions is identical except that all costs are multiplied by $O(1/B^{1/(1+\alpha)})$. □

COROLLARY 5.1. *For any ε with $0 < \varepsilon < 1$, there exists a setting of α such that the xDict supports searches in $O(\frac{1}{\varepsilon}\log_B \frac{N}{M})$ memory transfers and supports insertions in $O(\frac{1}{\varepsilon}\log_B(\frac{N}{M})/B^{1-\varepsilon})$ amortized memory transfers.*

Proof. First off, we consider only $\varepsilon < \frac{1}{2}$ (rolling up a particular constant into the big-O notation), as larger ε only hurt the performance of inserts.

Choose $\alpha = \varepsilon/(1-\varepsilon)$, which gives $B^{1/(1+\alpha)} = B^{1-\varepsilon}$. Because $\varepsilon < \frac{1}{2}$, we have $\alpha < 1$, and we can apply Theorem 5.1. The $1/\alpha$ term solves to $1/\alpha = (1-\varepsilon)/\varepsilon = O(1/\varepsilon)$ to complete the proof. □

[4] All x-boxes with size at most $O(M^{1/(1+\alpha)})$ fit in $O(M)$ memory as the x-box sizes increase supergeometrically.

6 Final notes

We did not address deletion in detail, but claim that it can be handed using standard techniques. For example, to delete an element we can insert an anti-element with the same key value. In the course of an operation, should a key value and its antivalue be discovered, they annihilate each other while releasing potential which is used to remove them from the buffers they are in. Rebuilding the whole structure when the number of deletions since the last rebuild is half of the structure ensures that the total size does not get out of sync with the number of not-deleted items currently stored.

Another detail is that, to hold n items, the xDict may create an n-box, which occupies up to $\Theta(n^{1+\alpha})$ of address space. However, only $\Theta(n)$ space of the xDict will ever be occupied, and the layout ensures that the unused space is at the end. Therefore the xDict data structure can use optimal $\Theta(n)$ space.

Acknowledgments

We would like to thank the organizers of the MADALGO Summer School on Cache-Oblivious Algorithms, held in Aarhus, Denmark, on August 18–21, 2008, for providing the opportunity for the authors to come together and create and analyze the structure presented in this paper.

References

[1] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

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

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

[4] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 53(2):115–136, 2004.

[5] 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 19th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 81–92, San Diego, CA, June 2007.

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

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

[8] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

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

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