External-Memory Multimaps

Elaine Angelino[2,⋆], Michael T. Goodrich[1,⋆⋆], Michael Mitzenmacher[2,⋆⋆⋆], and Justin Thaler[2,†]

[1] Dept. of Computer Science, University of California, Irvine
goodrich@ics.uci.edu
[2] School of Engineering and Applied Sciences, Harvard University,
{michaelm,jthaler,elaine}@eecs.harvard.edu

Abstract. Many data structures support dictionaries, also known as maps or associative arrays, which store and manage a set of key-value pairs. A ***multimap*** is a generalization that allows multiple values to be associated with the same key. For example, the inverted file data structure commonly used in search engines is a type of multimap, with words as keys and document pointers as values. We study the multimap abstract data type and how it can be implemented efficiently online in external memory frameworks, with constant expected I/O performance. The key technique used to achieve our results is a combination of cuckoo hashing using buckets that hold multiple items with a multiqueue implementation to cope with varying numbers of values per key. Our results are provably optimal up to constant factors.

1 Introduction

A ***multimap*** is a simple abstract data type (ADT) that generalizes the map ADT to support key-value associations in a way that allows multiple values to be associated with the same key. Specifically, it is a dynamic container, C, of key-value pairs, which we call ***items***, supporting (at least) the following operations:

- insert(k, v): insert the key-value pair, (k, v). This operation allows for there to be existing key-value pairs having the same key k, but we assume w.l.o.g. that the particular key-value pair (k, v) is itself not already present in C.
- isMember(k, v): return true if the key-value pair (k, v) is present in C.
- remove(k, v): remove the key-value pair (k, v) from C. This operation returns an error condition if (k, v) is not currently in C.
- findAll(k): return the set of all key-value pairs in C having key equal to k.
- removeAll(k): remove from C all key-value pairs having key equal to k.
- count(k): Return the number of values associated with key k.

⋆ A preliminary version of this work appears as a Brief Announcement at SPAA 2011.
⋆⋆ Supported in part by the NSF under grants 0724806, 0713046, and 0847968, and by the ONR under MURI grant N00014-08-1-1015.
⋆⋆⋆ Supported in part by the NSF grants 0915922 and 0964473.
† Supported by a DoD NDSEG Fellowship, and partially by NSF grant CNS-0721491.

T. Asano et al. (Eds.): ISAAC 2011, LNCS 7074, pp. 384–394, 2011.

Surprisingly, we are not familiar with any previous discussion of this abstract data type in the theoretical algorithms and data structures literature. Nevertheless, abstract data types equivalent to the above ADT, as well as multimap implementations, are included in the C++ Standard Template Library (STL), Guava–the Google Java Collections Library, and the Apache Commons Collection 3.2.1 API. The existence of these implementations provides empirical evidence for the usefulness of this abstract data type. In this work we describe efficient external-memory implementations of the multimap ADT. Due to space constraints, this paper is abbreviated. Many more details and proofs are available in the full version [1].

Motivation: A primary motivation for studying multimaps is that associative data in the real world often exhibits extreme non-uniformities. For example, many real-world data sets follow a power law with respect to data frequencies indexed by rank. A standard example is the frequency of words in a corpus of documents. We could use a multimap data structure to allow us to retrieve all instances of a query word w in such a corpus, but would require one that could handle large skews in the number of values per key. In this case, the multimap could be viewed as providing dynamic functionality for a classic static data structure, known as an ***inverted file*** or ***inverted index*** (e.g., see Knuth [6]). Given a collection Γ of documents, an inverted file allows one to list, for any word w, all the places in Γ where w appears.

Another powerful motivation for studying multimaps is graphical data [2]. A multimap can represent a graph: keys correspond to nodes, values correspond to neighbors, findAll operations list all neighbors of a node, and removeAll operations delete a node from the graph. The degree distribution of many real-life graphs follow a power law, motivating efficient handling of non-uniformity.

Related Work: Inverted files have standard applications in text indexing (e.g., see Knuth [6]), and are important data structures for modern search engines. Several works expand inverted files to support incremental and batched insertions, based on hash tables or B-trees, but many do not support fast deletions. Büttcher and Clarke [3] consider the trade-offs for allowing for both insertions and deletions in an inverted file, and Guo *et al.* [5] describe a solution for performing such operations by using a type of B-tree. Blandford and Blelloch [2] consider dictionaries on variable-length strings, but not in an external memory model. Recent work by Pagh et al. [9] studies cache-oblivious hashing, and achieves extremely fast (expected average) query time under realistic assumptions on the cache, asymptotically matching known cache-aware solutions. Like all work on hashing that assumes each key has a single value, [9] does not support the fast findAll and removeAll operations as we do in this work.

Our work utilizes a variation on cuckoo hash tables. We assume the reader has some familiarity with such hash tables, as originally presented by Pagh and Rodler [8] (or as described on Wikipedia).

Finally, recent work by Verbin and Zhang [12] shows that in the external memory model, for any dynamic dictionary data structure with query cost $O(1)$,

the expected amortized cost of updates must be at least 1. As explained below, this implies our data structure is ***optimal*** up to constant factors.

Our Results: Our algorithms are for the standard two-level I/O model, which captures the memory hierarchy of modern computer architectures. In this model, there is a cache of size M connected to a disk of unbounded size, and the cache and disk are divided into blocks, where each block can store up to B items. Any algorithm can only operate on cached data, and algorithms must therefore make memory transfer operations, which read a block from disk into cache or vice versa. The cost of an algorithm is the number of I/Os required, with all other operations considered free. All of our time bounds hold even when $M = O(B)$, and we therefore omit reference to M throughout.

We provide an online implementation of the multimap abstract data type, where each operation must completely finish executing prior to our beginning execution of any subsequent operations. In describing our performance bounds, we use $\bar{O}(*)$ to denote an expected bound, B to denote the block size, N to denote the number of key-value pairs, and n_k to denote the number of key-value pairs with key equal to k. All bounds are ***unamortized***. Our bounds are $O(1)$ for isMember(k, v), remove(k, v), removeAll(k), and count(k); $\bar{O}(1)$ for insert(k, v); and $O(1+n_k/B)$ for findAll(k). Our constructions are based on the combination of two external-memory data structures—external-memory cuckoo hash tables and multiqueues—which may be of independent interest.

Since we achieve query cost $O(1)$, the lower bound of [12] implies our $O(1)$ update cost is optimal up to constant factors. That we in addition achieve efficient removeAll and findAll operations demonstrates the strength of our results.

Finally, we simulate our suggested implementation to test our performance guarantees. While our implementation is not especially space efficient, yielding a memory utilization of between .32 and .39, it uses very few I/O operations, and we believe the ideas we present will yield more effective future implementations.

2 External-Memory Cuckoo Hashing

In this section, we describe external-memory versions of cuckoo hash tables with multiple items per bucket, which can be used to implement the map ADT when all key-value pairs are distinct. Later we use this approach in concert with multiqueues to support multiple values for the same key for the multimap ADT.

Cuckoo hash tables that store multiple items per bucket have been studied previously, having been introduced in [4]. Generally the analysis has been limited to buckets of a constant size (number of items) d. For our external-memory cuckoo hash table, each bucket can store B items, where B is a parameter defining our block size and is not necessarily a constant.

Formally, let $\mathcal{T} = (T_0, T_1)$ be a cuckoo hash table such that each T_i consists of $\gamma n/2$ buckets, where each bucket stores a block of size B, with $n = N/B$. One setting of particular interest is when $\gamma = 1 + \epsilon$ for some (small) $\epsilon > 0$, so that space overhead of the hash table is only an ϵ factor over the minimum possible. The items in $\mathcal{T}$ are indexed by keys and stored in one of two locations, $T_0[h_0(k)]$

or $T_1[h_1(k)]$, where h_0 and h_1 are hash functions, and we assume for simplicity throughout the paper that all hash functions are completely random.

It should be clarified that, in some settings, the use of a cuckoo hash function may be unnecessary or even unwarranted. Indeed, if $B > c \log N$ for a suitable constant c and $\gamma = 1 + \epsilon$, we can insted use simple hash tables, with just one choice for each item; simple tail bounds suffice to show that with high probability all buckets will fit all the items that hash to it. Cuckoo hashing here allows us to avoid such "wide block assumptions," giving a more general approach.

The important feature of the cuckoo hashing implementation is the way it may reallocate items in $\mathcal{T}$ during an insertion. Standard cuckoo hashing, with one item per bucket, immediately evicts the previous (and only) item in a bucket when a new item is to be inserted in an occupied bucket. With multiple items per bucket, there is a choice available. We describe what is known in this setting, and how we modify it for our use here.

Let G be the ***cuckoo graph***, where each bucket in $\mathcal{T}$ is a vertex and, for each item x currently in C, we connect $T_0[h_0(x)]$ and $T_1[h_1(x)]$ as a directed edge, with the edge pointing toward the bucket it is not currently stored in. Suppose we wish to insert an item x into bucket X. If X contains fewer than B items, then we simply add x to X. Otherwise, we need to make room for the new item.

One approach for doing an insertion is to use a breadth first search on the cuckoo graph. The results of Dietzfelbinger and Weidling [4] show that for sufficiently large *constant* B, the expected insertion time is constant. Specifically, when $\gamma = 1 + \epsilon$ and $B \geq 16 \ln(1/\epsilon)$, the expected time to insert a new key is $(1/\epsilon)^{O(\log \log(1/\epsilon))}$, which is a constant. (This may require re-hashing all items in very rare cases when an item cannot be placed; the expected time remains constant.) Notice that if B grows in a fashion that is $\Omega(1)$, then breadth first search does not naturally take constant expected time, as even the time to look if items currently in the bucket can be moved will take $\Omega(B)$ time. (It might still take constant expected time, but this does not appear to follow from [4].)

For non-constant B, we can apply the following mechanism: we can use our buckets to mimic having B/c distinct subtables for some large constant c, where the ith subtable uses the ci/Bth fraction of the bucket space, and each item is hashed into a specific subtable. For $B = O(N^\delta)$ for $\delta < 1$, each subtable will contain close to its expected number of items with high probability. Further, by choosing c suitably large one can ensure that each subtable is within a $1+\epsilon$ factor of its total space while maintaining an expected $(1/\epsilon)^{O(\log \log(1/\epsilon))}$ insertion time. Specifically, we have the following theorem:

Theorem 1. *Suppose that for a cuckoo hash table $\mathcal{T}$ with at least $(1 + \epsilon)N/B$ blocks the block size satisfies $B = \Omega(1)$ and $B = O(N^\delta)$ for $\delta < 1$. Let $0 < \epsilon \leq 0.1$ be arbitrary and let C be a collection of N items. Suppose further we have B/c subtables, with $c = 16 \ln(1/\epsilon)$ with each item hashed to a subtable by a fully random hash function, and the hash functions for each subtable are fully random. Finally, suppose the items of C have been stored in $\mathcal{T}$ by an algorithm using the partitioning process described above and the cuckoo hashing process. Then the expected time for the insertion of a new item x using a BFS is $(1/\epsilon)^{O(\log \log(1/\epsilon))}$.*

As noted in [4], a more practical approach is to use *random walk cuckoo hashing* in place of breadth first search cuckoo hashing, and we use random walk cuckoo hashing in the simulations in Section 4. Finally, item lookups and removals use a worst-case constant number of I/Os.

3 External-Memory Multimaps

We now build on the result of Section 2 to describe how to maintain a multimap that allows fast dynamic access in external memory.

The Primary Structure: To implement the multimap ADT, we begin with a primary structure that is an external-memory cuckoo hash table storing just the set of keys. In particular, each record, $R(k)$, in $\mathcal{T}$, is associated with a specific key, k, and holds the following fields:

- the key, k, itself
- the number, n_k, of key-value pairs in C with key equal to k
- a pointer, p_k, to a block X in a secondary table, $\mathcal{S}$, that stores items in C with key equal to k. If $n_k < B$, then X stores all the items with key equal to k (plus possibly some items with keys not equal to k). Otherwise, if $n_k \geq B$, then p_k points to a ***first*** block of items with key equal to k, with the other blocks of such items being stored elsewhere in $\mathcal{S}$.

This secondary storage is an external-memory data structure we are calling a ***multiqueue***.

An External-Memory Location-Aware Multiqueue: The secondary storage that we need in our construction is a way to maintain a set $\mathcal{Q}$ of queues in external memory. We assume the ***header*** pointers for these queues are stored in an array, $\mathcal{T}$, which in our external-memory multimap construction is the external-memory cuckoo hash table described above.

For any queue, Q, we wish to support the following operations:

- enqueue(x, H): add the element x to Q, given a pointer to its header, H.
- remove(x): remove x from Q. We assume in this case that each x is unique.
- isMember(x): determine whether x is in some queue, Q.

In addition, we wish to maintain all these queues in a space-efficient manner, so that the total storage is proportional to their total size. To enable this, we store all the blocks used for queue elements in a secondary table, $\mathcal{S}$, of blocks of size B each. Thus, each header record, H in $\mathcal{T}$, points to a block in $\mathcal{S}$.

Our intent is to store each queue Q as a doubly-linked list of blocks from $\mathcal{S}$. Unfortunately, some queues in $\mathcal{Q}$ are too small to deserve an entire block in S dedicated to storing their elements. So small queues must share their first block of storage with other small queues until they are large enough to deserve an entire block dedicated to their elements. Initially, all queues are assumed to be empty; hence, we initially mark each queue as ***light***. In addition, the blocks in $\mathcal{S}$ are initially empty; hence, we link the blocks of $\mathcal{S}$ in a consecutive fashion as a doubly-linked list and identify this list as being the ***free list***, F, for $\mathcal{S}$.

We set a heavy-size threshold at $B/3$ elements. When a queue Q stored in a block X reaches this size, we allocate a block from $\mathcal{S}$ (taking a block off the free list F) exclusively to store elements of Q and we mark Q as ***heavy***. Likewise, to avoid wasting space as elements are removed from a queue, we require any heavy queue Q to have at least $B/4$ elements. If a heavy queue's size falls below this threshold, then we mark Q as being light again and we force Q to go back to sharing space with other small queues. This may involve returning a block to the free list F. In this way, each block X in $\mathcal{S}$ will either be empty or will have all its elements belonging to a single heavy queue or as many as $O(B)$ light queues. In addition, these rules imply that $O(B)$ element insertions are required to take a queue from the light state to the heavy state and $O(B)$ element removals are required to take a queue from the heavy state to the light state.

If a block X in $\mathcal{S}$ is being used for light queues, then we order the elements in X according to their respective queues. Each block for a heavy queue Q stores previous and next pointers to the neighboring blocks in the linked list of blocks for Q, with the first such block pointing back to the header record for Q. As we show, this organization allows us to maintain our size and label invariants during execution of enqueue and remove operations.

One additional challenge is that we want to support the remove(x) operation to have a constant I/O complexity. Thus, we cannot afford to search through a list of blocks of a queue looking for an element x we wish to remove. So, in addition to the table $\mathcal{S}$ and its free list, F, and the headers for each queue in $\mathcal{Q}$, we also maintain an external-memory cuckoo hash table, $\mathcal{D}$, to be a dictionary that maps each queue element x to the block in $\mathcal{S}$ that stores x. This allows our multiqueue to be ***location-aware***, that is, to support fast searches to locate the block in $\mathcal{S}$ that is holding any element x that belongs to some queue, Q.

We call any block in $\mathcal{S}$ containing fewer than $B/4$ items ***deficient***. To ensure that our multiqueue uses total storage proportional to its total size,we enforce the following two rules. Together, these rules guarantee that there are $O(N/B)$ deficient blocks in $\mathcal{S}$, and hence our multiqueue uses $O(N/B)$ blocks of memory.

1. Each block Y in $\mathcal{T}$ stores a pointer d, called the deficient pointer, to a block $d(Y)$; the identity of this block is allowed to vary over time. We ensure that at all times, $d(Y)$ is the only (possibly) deficient block associated with Y that stores light queues.
2. Each heavy queue Q also stores in its header block a deficient pointer d to a block $d(Q)$. At all times, $d(Q)$ is the only (possibly) deficient block devoted to storing values for Q.

For the remainder of this subsection, we describe how to implement all multiqueue operations to obtain constant ***amortized*** expected or worst-case runtime. We show how to deamortize these operations in the full version of the paper [1].

The Split Action. As we perform enqueue operations, a block X may overflow its size bound, B. In this case, we need to split X in two, which we do by allocating a new block X' from $\mathcal{S}$ (using its free list). We call X the ***source*** of the split, and X' the ***sink*** of the split. We then proceed depending on whether X contains elements from light queues or a single heavy queue.

1. X contains elements from light queues. We greedily copy elements from X into X' until X' has size has size at least $B/3$, keeping the elements from the same light queue together. Note that each light queue has less than $B/3$ elements, so this split will result in at least a 1/3–2/3 balance.
   To maintain our invariants, we must change the header records from X to X' for any queues that we just moved to X'. We achieve this by performing a look-up in $\mathcal{T}$ for each key corresponding to a queue that was moved from X to X', and modifying its header record, which requires $O(B)$ I/Os. Similarly, to support location awareness, we must also update the dictionary $\mathcal{D}$. So, for each element x that is moved to X', we look up x in $\mathcal{D}$ and update its pointer to now point to X'. In total this costs $O(B)$ I/Os.
2. X contains elements from a single heavy queue Q. In this case, we move ***no*** elements, and simply take a block X' from the free list and insert it as the head of the list of blocks devoted to Q, changing the header record H in $\mathcal{T}$ to point to X'. We also change the deficient pointer d for Q to point to X', and insert into X' the element that caused the split. This takes $O(1)$ I/O operations in total.

The Enqueue Operation. Given the above components, let us describe how we perform the enqueue and remove operations. We begin with the enqueue(x, H) operation. We consider how this operation acts, depending on a few cases.

1. The queue for H is empty (hence, H is a null pointer and its queue is light). We examine the block Y from $\mathcal{T}$ to which H belongs. If $d(Y)$ is null, we first take a block X of the free list and set $d(Y)$ to X before continuing. We follow the deficient pointer for Y, $d(Y)$, and add x to $d(Y)$. If this causes the size of $d(Y)$ to reach B, then we split $d(Y)$ as described above.
2. The queue Q for H is not empty. We proceed according to two cases.
   (a) If Q is a light queue, we follow H to its block X in $\mathcal{S}$ and add x to X. If this brings the size of Q above $B/3$, we perform a ***light-to-heavy transition***, taking a block X' off the free list, moving all elements in Q to X', and marking Q as heavy. If this brings the size of X below $B/4$, we process X as in the remove operation below.
   (b) If Q is heavy, we add x to $X = d(Q)$, the (possibly) deficient block for Q. If this brings the size of X to B, then we split X as described above.

Once the element x is added to a block X in $\mathcal{S}$, we then add x to the dictionary $\mathcal{D}$, and have its record point to X.

The Remove and isMember Operations. In both of these operations, we look up x in $\mathcal{D}$ to find the block X in $\mathcal{S}$ that contains x. In the isMember(x) case, we complete the operation by simply looking for x in X. In the remove(x) operation, we do this look up and then remove x from X if we find x. If this causes Q to become empty, then we update its header, H, to be null. In addition, if this operation causes the size of X to go below $B/4$, then we need to do some additional work, based on the following cases:

1. Q is a heavy queue.
   (a) If X is the only block for Q, then Q should now be considered a light queue; hence, we continue processing it according to the case listed below where X contains only light queues. We refer to the entirety of this action as a ***heavy-to-light*** queue transition.
   (b) Otherwise, if $X = d(Q)$, then we are done because $d(Q)$ is allowed to be deficient. If $X \neq d(Q)$, we proceed based on the following two cases:
      i. ***d-alteration action:*** If the size of $d(Q)$ is at least $2B/3$, we simply update Q's deficient pointer, d, to point to X instead of $d(Q)$.
      ii. ***Merge action***: If the size of $d(Q)$ is less than $2B/3$, then we move all of the elements of X into $d(Q)$ and we update the pointer in $\mathcal{D}$ for each moved element. X is returned to the free list. We call X the source of the merge, and $d(Q)$ the sink. (Note that in this case, the size of $d(Q)$ becomes at most $11B/12$.)
2. X contains light queues (hence, no heavy queue elements). In this case, we visit the header H for Q. Let Y denote the block containing H.
   (a) If $X = d(Y)$ we are done, since $d(Y)$ is allowed to be deficient.
   (b) If $X \neq d(Y)$, let Z be the size of $d(Y)$.
      i. ***d-alteration action:*** If $Z \geq 2B/3$ then we simply update d to point to X instead of $d(Y)$.
      ii. ***Merge action***: If $Z < 2B/3$, then we merge the elements in X into $d(Y)$, which now has size at most $11B/12$, and update pointers in $\mathcal{D}$ and $\mathcal{T}$ for the elements that are moved. We return X to the free list. We call X the source of the merge and $d(Y)$ the sink.

If a block X' is pointed to by any deficient pointer d, it is helpful to think of this as "protection" for X' from being the source of a merge. Once X' is afforded this protection, it will not lose it until its size is at least $2B/3$ (see the d-alteration action). At a high level, this will allow us to argue that if X and X' are respectively the source and sink of a merge action, neither X nor X' will be the source of a subsequent merge or split operation until it is the target of $\Omega(B)$ enqueue or remove operations, even though X' may have size very close to the deficiency threshold $B/4$. This will allow us to charge the $O(B)$ I/Os performed in any split or merge action to the $\Omega(B)$ operations that caused one of these blocks to shrink to size $B/4$ or grow to size B. We can deamortize these operations to obtain the following theorem.

Theorem 2. *We can implement a location-aware multiqueue so that the remove(x) and isMember(x) operations each use $O(1)$ I/Os, and the enqueue(x, H) operation uses $O(1 + t(N))$ expected I/Os, where $t(N)$ is the expected number of I/Os needed to perform an insertion in an external-memory cuckoo table of size N.*

It should be clear from our description that, except for trivial cases (such as having only a constant number of elements), the space usage of our multiqueue implementation is within a constant factor of the optimal. We have not attempted to optimize this factor, though there is some flexibility in the multiqueue operations (such as when to do a split) that allow some optimization.

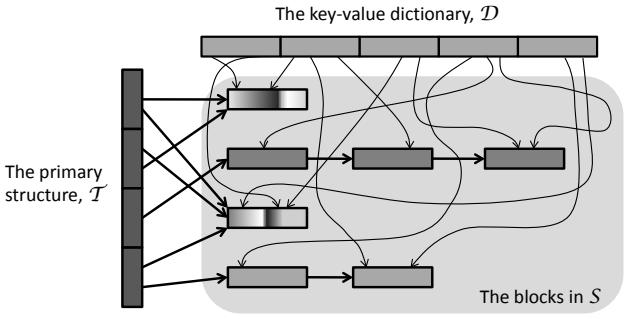

Fig. 1. The external-memory multimap

Combining Cuckoo Hashing and Location-Aware Multiqueues: We now describe how to construct an efficient external-memory multimap implementation by combining the data structures described above.

We store an external-memory cuckoo hash table, as described above, as our primary structure, $\mathcal{T}$, with each record pointing to a block in a multiqueue, $\mathcal{S}$, having an auxiliary dictionary, D, implemented as yet another external-memory cuckoo hash table. The multimap ADT then functions as follows.

- insert(k, v): To insert the key-value pair (k, v) we first perform a look up for k in $\mathcal{T}$. If there is already a record for k in $\mathcal{T}$, we increment its count. We then follow its pointer to the appropriate block X in $\mathcal{S}$, and add the pair (k, v) to $\mathcal{S}$, as in the enqueue multiqueue method. Otherwise we insert k into $\mathcal{T}$ with a null header record and count 1 and then add the pair (k, v) to $\mathcal{S}$ as in the enqueue multiqueue method.
- isMember(k, v): This is identical to the isMember(k, v) multiqueue operation.
- remove(k, v): To remove the key-value pair (k, v) from C, we perform a look up for (k, v) in $\mathcal{D}$. If there is no record for (k, v) in $\mathcal{D}$, we return an error condition. Otherwise, we follow this pointer to the appropriate block X of $\mathcal{S}$ holding the pair (k, v). We remove the pair (k, v) from $\mathcal{S}$ and $\mathcal{D}$ as in the remove multiqueue method, and decrement its count.
- findAll(k): To return all key-value pairs in C having key k, we perform a look up for k in $\mathcal{T}$, and follow its pointer to the appropriate block of $\mathcal{S}$. If this is a light queue, then we return the items with key k. Otherwise, we return the entire block and all the other blocks of this queue as well.
- removeAll(k): We give here a constant amortized time implementation, and provide a deamortization in [1]. To remove from C all key-value pairs having key k, we perform a look up for k in $\mathcal{T}$, and follow its pointer to the appropriate block X of $\mathcal{S}$. If this is a light queue, then we remove from X all items with key k and remove all affected pointers from $\mathcal{D}$; if this causes X to become deficient, we perform a merge action or d-alteration action as in the remove multiqueue method. If this is a heavy queue, we walk through all blocks of this queue and remove all items from these blocks and return each block to the free list. We also remove all affected pointers from $\mathcal{D}$. Finally,

we remove the header record for k from $\mathcal{T}$, which implicitly sets the count of k to zero as well. We charge, in an amortized sense, the work for all the I/Os to the insertions that added these key-value pairs to C originally.
- count(k): Return n_k, which we track explicitly for all keys k in $\mathcal{T}$.

Theorem 3. *One can implement the multimap ADT in external memory using $O(N/B)$ blocks of memory with the following: I/O performance: $O(1)$ for isMember(k, v), remove(k, v), removeAll(k), and count(k); $O(1)$ for insert(k, v); and $O(1 + n_k/B)$ for findAll(k).*

4 Experiments

We performed extensive simulations in order to explore how various settings of the design parameters affect I/O complexity and space usage, for both our basic algorithm and our deamortized algorithm. To summarize, in both versions our memory utilization was between .32 and .39 for all parameter settings tested, and the average I/O cost over all insert and remove operations is extremely low: never more than about 3.5 I/Os per operation. As expected, the cost distribution in the basic algorithm is bimodal – the vast majority (over 99.9%) of operations require about 4 I/Os, but a small fraction of operations require several hundred, with the maximum number of I/Os ranging between 400 and 650. In stark contrast, the deamortized implementation never requires more than a few dozen I/Os for any given operation; it also used slightly fewer I/Os on average. We have tested our performance against a B-tree variant, and preliminary tests show our performance is competitive.

References

1. Angelino, E., Goodrich, M.T., Mitzenmacher, M., Thaler, J.: External-Memory Multimaps. CoRR abs/1104.5533 (2011)
2. Blandford, D., Blelloch, G.: Compact dictionaries for variable-length keys and data with applications. ACM Trans. Alg. 4(2), 1–25 (2008)
3. Büttcher, S., Clarke, C.L.A.: Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In: Proc. of 14th ACM Conf. on Information and Knowledge Management (CIKM), pp. 317–318 (2005)
4. Dietzfelbinger, M., Weidling, C.: Balanced allocation and dictionaries with tightly packed constant size bins. Theoretical Computer Science 380, 47–68 (2007)
5. Guo, R., Cheng, X., Xu, H., Wang, B.: Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In: Proc. of 16th ACM Conf. on Information and Knowledge Management (CIKM), pp. 751–760 (2007)
6. Knuth, D.E.: Sorting and Searching. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading (1973)
7. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)
8. Pagh, R., Rodler, F.: Cuckoo hashing. Journal of Algorithms 52, 122–144 (2004)

9. Pagh, R., Wei, Z., Yi, K., Zhang, Q.: Cache-Oblivious Hashing. In: Proc. of PODS, pp. 297–304 (2010)
10. Panigrahy, R.: Efficient hashing with lookups in two memory accesses. In: Proc. of the 16th Annual ACM-SIAM Symp. on Discrete Algorithms, pp. 830–839 (2005)
11. Panigrahy, R.: Hashing, Searching, Sketching. Ph.D. thesis, Dept. of Computer Science, Stanford University (2006)
12. Verbin, E., Zhang, Q.: The limits of buffering: a tight lower bound for dynamic membership in the external memory model. In: Proc. STOC, pp. 447–456 (2010)