Using Hashing to Solve the Dictionary Problem (In External Memory)

John Iacono
NYU Poly

Mihai Pătraşcu
AT&T Labs

February 9, 2011

Abstract

We consider the dictionary problem in external memory and improve the update time of the well-known *buffer tree* by roughly a logarithmic factor. For any $\lambda \geq \max\{\lg\lg n, \log_{M/B}(n/B)\}$, we can support updates in time $O(\frac{\lambda}{B})$ and queries in time $O(\log_\lambda n)$. We also present a lower bound in the cell-probe model showing that our data structure is optimal.

In the RAM, hash tables have been use to solve the dictionary problem faster than binary search for more than half a century. By contrast, our data structure is the first to beat the comparison barrier in external memory. Ours is also the first data structure to depart convincingly from the *indivisibility* paradigm.

1 Introduction

The case for buffer trees. The dictionary problem asks to maintain a set S of up to n keys from the universe U, under insertions, deletions, and membership queries. The keys may also have associated data (given at insert time), which the queries must retrieve. Many types of hash tables can solve the dictionary problem with constant time per operation, either in expectation or with high probability. These solutions assume a Random Access Machine (RAM) with words of $\Omega(\lg U)$ bits, which are sufficient to store keys and pointers.

In today's computation environment, the external memory model has become an important alternative to the RAM. In this model, we assume that memory is accessed in pages of B words. The processor is also equipped with a cache of M words (M/B pages), which is free to access. The model can be applied at various levels, depending on the size of the problem at hand. For instance, it can model the interface between disk and main memory, or between main memory and the CPU's cache.

Hash tables benefit only marginally from the external memory model: in simple hash tables like chaining, the expected time per operation can be decreased to $1 + 2^{-\Omega(B)}$ [Knu73]. Note that as long as M is smaller than n, the query time cannot go significantly below 1. However, the power of external memory lies in the paradigm of *buffering*, which permits significantly faster updates. In the most extreme case, if a data structure simply wants to record a fast stream of updates, it can do so with an amortized complexity of $O(1/B) \ll 1$ per insertion: accumulate B data items in cache, and write them out at once.

Buffer trees, introduced by Arge [Arg03], are one of the pillars of external memory data structures, along with B-trees. Buffer trees allow insertions at a rate close to the ideal $1/B$, while maintaining reasonably efficient (but superconstant) queries. For instance, they allow update time $t_u = O(\frac{\lg n}{B})$ and query time $t_q = O(\lg n)$. More generally, they allow the trade-offs:

$$t_u = O(\tfrac{\lambda}{B} \lg n) \qquad t_q = O(\log_\lambda n), \qquad \text{for } 2 \le \lambda \le B \tag{1}$$

$$t_u = O(\tfrac{1}{B} \log_\lambda n) \qquad t_q = O(\lambda \lg n), \qquad \text{for } 2 \le \lambda \le \tfrac{M}{B}. \tag{2}$$

In these bounds and the rest of the paper, we make the following reasonable and common assumptions about the parameters: $B \ge \lg n$; $M \ge B^{1+\varepsilon}$ (tall cache assumption); $n \ge M^{1+\varepsilon}$.

The motivation for fast (subconstant) updates in the external memory is quite strong. In applications where massive streams of data arrive at a fast rate, the algorithm may need to operate close to the disk transfer rate in order to keep up. On the other hand, we want reasonably efficient data structuring to later find the proverbial needle in the haystack.

A second motivation comes from the use of data structures in algorithms. Sorting in external memory takes $O(\frac{n}{B} \log_M n)$, which is significantly sublinear. Achieving a bound close to this becomes the holy grail for many algorithmic problems. Towards such an end, a data structure that spends constant time per operation is of little relevance.

Finally, fast updates can translate into fast *queries* in realistic database scenarios. Database records typically contain many fields, a subset of which are relevant in a typical queries. Thus, we would like to maintain various indexes to help with different query patterns. If updates are efficient, we can maintain more indexes, so it is more likely that we can find a selective index for a future query. (We note that this idea is precisely the premise of the start-up company Tokutek, founded by Michael Bender, Martín Farach-Colton and Bradley Kuszmaul. By using buffer-tree technology to support faster insertions in database indexes, the company is reporting[1] significant improvements in industrial database applications.)

[1] `http://tokutek.com/`.

The comparison/indivisibility barrier. In internal memory, hash tables can be traced back at least to 1953 [Knu73]. By contrast, in external memory the state-of-the-art data structures (buffer trees, B-trees, many others built upon them) are all comparison based!

The reason for using comparison data structures in external memory seems more profound that in internal memory. Consider the simple task of arranging n keys in a desired order (permuting data). The best known algorithm takes time $O(\min\{n, \frac{n}{B}\log_M n\})$: either implement the permutation ignoring the paging, or use external memory sorting. Furthermore it has been known since the seminal paper of Aggarwal and Vitter [AV88] that if the algorithm manipulates data items as indivisible atoms, this bound is tight. It is often conjectured this this lower bound holds for *any* algorithm, not just those in the indivisible model. This would imply that, whenever B is large enough for the internal memory solution to become irrelevant, a task as simple as permuting is naturally as hard as comparison-based sort.

While external-memory data structures do not need to be comparison based, they naturally manipulate keys as indivisible objects. This invariably leads to a comparison algorithm: the branching factors that the data structure can achieve are related to the number of items in a page, and such branching factors can be achieved even by simple comparison-based algorithms. For problems such as dictionary, predecessor search, or range reporting the best known bounds become (comparison-based) B-trees or buffer trees, when B is large enough (and the external memory solution overtakes the RAM-based solution). It is plausible to conjecture that this is an inherent limitation of external memory data structures (in fact, such a conjecture was put forth by [Yi10]).

Our work presents the first powerful use of hashing to solve the external memory dictionary problem, and the first data structure to depart significantly from the indivisibility paradigm. We obtain:

Theorem 1. *For any* $\max\{\lg\lg n, \log_M n\} \le \lambda \le B$, *we can solve the dictionary problem by a Las Vegas data structure with update time* $t_u = O(\frac{\lambda}{B})$ *and query time* $t_q = O(\log_\lambda n)$ *with high probability.*

At the high end of the trade-off, for $\lambda = B^\varepsilon$, we obtain update time $O(1/B^{1-\varepsilon})$ and query time $O(\log_B n)$. This is the same as standard buffer trees. Things are more interesting at the low end (fast updates), which is the *raison d'être* of buffer trees. Comparing to (1), our results are a logarithmic improvement over buffer trees, which could achieve $t_u = O(\frac{\lambda}{B}\lg n)$ and $t_q = O(\log_\lambda n)$.

Interestingly, the update time can be pushed very close to the ideal disk transfer rate of $1/B$: we can obtain $t_u^{\min} = O(\frac{1}{B}\cdot\max\{\log_M n, \lg\lg n\})$.

Note that it is quite natural to restrict the update time to $\Omega(\frac{1}{B}\log_M n)$. Unless one can break the permutation bound, this is an inherent limitation of any data structure that has some target order in which keys should settle after a long enough presence in the data structure (be it the sorted order, or a hash-based order). Since buffer trees work in the indivisible model, they share this limitation. However, the buffer tree pays a significant penalty in query time to achieve $t_u = O(\frac{1}{B}\log_M n)$: from (2), this requires $t_q \ge M^\varepsilon \lg n$, which is significantly more than polylogarithmic (for interesting ranges of M). By contrast, our bound on the query time is still (slightly) sublogarithmic.

If one assumes M is fairly large (such as n^ε), then $\lambda \ge \lg\lg n$ becomes the bottleneck. This is an inherent limitation of our new data structure. In this case, we can achieve $t_u = O(\frac{\lg\lg n}{B})$ and $t_q = O(\lg n/\lg\lg\lg n)$. By contrast, buffer trees naturally achieve $t_u = O(\frac{\lg n}{B})$ and $t_q = O(\lg n)$. With a comparable query time, our data structure gets exponentially closer to the disk transfer rate for updates. If we ask for $t_u = O(\frac{\lg\lg n}{B})$ in buffer trees, then, from (2), we must have a huge query time of $t_q = 2^{\Omega(\lg n/\lg\lg n)}$.

Our result suggests exciting possibilities in external memory data structures. It is conceivable that, by abandoning the comparison and indivisibility paradigms, long-standing running times of natural problems such as predecessor search or range reporting can also be improved.

Lower bounds. We complement our data structure with a lower bound that shows its optimality:

Theorem 2. *Let $\varepsilon > 0$ be an arbitrary constant. Consider a data structure for the membership problem with at most n keys from the universe $[2n]$, running in the cell-probe model with cells of $O(B \lg n)$ bits and a state (cache) of M bits. Assume $B \geq \lg n$ and $M \leq n^{1-\varepsilon}$. The data structure may be randomized. Let t_u be the expected amortized update time, and t_q be the query time. The query need only be correct with probability $1 - C_\varepsilon$, where C_ε is a constant depending on ε.*

If $t_u \leq 1 - \varepsilon$, then $t_q = \Omega(\lg n / \lg(B \cdot t_u))$.

Remember that for any desired $t_u \geq t_u^{\min}$, our new data structure obtained a query time of $t_q = O(\lg n / \lg(B \cdot t_u))$. In other words, we have shown an *optimal* trade-off for any $t_u \geq t_u^{\min}$. We conjecture that for $t_u = o(t_u^{\min})$, the query time cannot be polylogarithmic in n.

Our lower bound holds for any reasonable cache size, $M \leq n^{1-\varepsilon}$. One may wonder whether a better bound for smaller M is possible (e.g. proving that for $t_u = o(\frac{1}{B} \log_B n)$, the query time needs to be super-logarithmic). Unfortunately, proving this may be very difficult. If sorting n keys in external memory were to take time $O(n/B)$, then our data structure will work for any $t_u = \Omega(\lg \lg n / B)$, regardless of M. Thus, a better bound for small cache size would imply that sorting requires superlinear time (and, in particular, a superlinear circuit lower bound).

Remember that update time below $1/B$ is unattainable, regardless of the query time. Thus, the remaining gap in understanding membership is in the range $t_u \in \left[\frac{1}{B}, \frac{\lg \lg n}{B}\right]$.

At the high end of our trade-off, we see a sharp discontinuity between internal memory solutions (hash tables with $t_u = t_q \approx 1$) and buffer trees. For any $t_u < 1 - \varepsilon$, the query time blows up to $\Omega(\log_B n)$.

The lower bound works in the strongest possible conditions: it holds even for membership and allows Monte Carlo randomization. Note that the error probability can be made an arbitrarily small constant by $O(1)$ parallel constructions of the data structure. However, since we want a clean phase transition between $t_u = 1 - \varepsilon$ and $t_u = 1$, we mandate a fixed constant bound on the error.

The first cell-probe lower bounds for external-memory membership was by Yi and Zhang [YZ10] in SODA'10. Essentially, they show that if $t_u \leq 0.9$, then $t_q \geq 1.01$. This bound was significantly strengthened by Verbin and Zhang [VZ10] in STOC'10. They showed that for any $t_u \leq 1 - \varepsilon$, then $t_q = \Omega(\log_B n)$.

This bound is recovered as the most extreme point on our trade-off curve. However, our proof is significantly simpler than that of Verbin and Zhang. We also note that the technique of [VZ10] does not yield better query bounds for fast updates. A lower bound for small t_u is particularly interesting given our new data structure and the regime in which buffer trees are most appealing.

For update time $t_u \ll \frac{\lg n}{B}$, our lower bound even beats the best known comparison lower bound. This was shown by Brodal and Fagerberg [BF03] in SODA'03, and states that $t_q = \Omega(\lg n / \log(t_u B \lg n))$. On the other hand, in the comparison model, it was possible to show [BF03] that one cannot take $t_u \ll \frac{1}{B} \log_M n$ (below the permutation barrier) without a significant penalty in query time: $t_q \geq n^{\Omega(1)}$.

2 Upper Bound

As a warm-up, we show how we can assume that keys and associated data have $O(\lg n)$ bits. The data structure can simply log keys in an array, ordered by insertion time. In addition, it hashes keys to the universe $[n^2]$, and inserts each key into a buffer tree with a pointer (also $O(\lg n)$ bits) into the time-ordered array. A buffer-tree query may return several items with the same hash value. Using the associated pointers, we can inspect the true value of the key in the logging array. Each takes one memory access, but we know that there are only $O(1)$ false positives with high probability (w.h.p.), with a good enough hash function.

Deletions can be handled in a black-box fashion: add the key to the logging array with a special associated value that indicates a delete, and insert it into the buffer tree normally. Throughout the paper, we will consider buffer trees with an "overwrite" semantics: if the same key is inserted multiple times, with different associated values, only the last one is relevant. Thus, the query will return a pointer to the deletion entry in the log. After $O(n)$ deletions we perform global rebuilding of the data structure to keep the array bounded.

From now on, we assume the keys have $O(\lg n)$ bits. Let $b = \Theta(B \lg n)$ be the number of bits in a page.

2.1 Gadgets

For now, assume we are dealing with only $n \leq M$ keys. Our data structure hashes each key to the universe $[b] \times [n] \times [n]$, i.e. to an integer of $2 \lg n + \lg b$ bits. We call the hash code in $[b]$ the *page hash*. The other two hash codes are the *distribution hash* and the *shadow hash*.

At a high level, our data structure follows the van Emde Boas layout. The distribution hash is used to decide how the key should move between recursive components of the data structure. Inside such a component, some interval of the bits of the distribution hash are still active, and used to identify and further route the key. The same bit interval of the shadow hash is passed to this component. The page hash follows the key everywhere, unchanged.

Our data structure is easiest to define in terms of the following gadgets. A *t-gadget* stores a *multiset* $S \subset [b] \times [t] \times [t]$. Each key from S has as associated data a value from $[t^3]$.

To understand the functionality of the gadget, first note that $\frac{b}{\lg t}$ keys and their associated data can be represented in $O(1)$ pages. By adjusting constants, we will simply assume $\frac{b}{\lg t}$ keys fit in exactly one page. (For a pure I/O bound, the representation inside a page does not matter; it suffices that a succinct representation exists. However, very simple data structures will suffice for our purposes.)

Formally, the gadget implements two operations:

INSERT(T)**:** Insert a multiset $T \subseteq [b] \times [t]^2$ into the data structure ($S \leftarrow S \cup T$). The multiset and its data is presented in page-packed form, so we aim for time bounds proportional to $O(\lceil T / \frac{b}{\lg t} \rceil)$.

QUERY(x)**:** Given a key $x \in [b] \times [t]^2$, return the (possibly empty) list of associated data of x. We aim for time bounds close to the number of occurrences of x in the multiset.

It is an important invariant of our data structure that at most $|S| \leq \Theta(bt)$ keys are inserted into any t-gadget. Furthermore, $|S| \leq M$, so the entire gadget can be loaded into cache if desired. However, at the beginning of any operation, the gadget cannot assume that any of its pages are in cache, i.e. the running times are at least 1.

At any point in time, we ask the gadget to occupy $O(|S| / \frac{b}{\lg t})$ pages. Our gadget implementation can grow using standard memory management techniques (maintaining a global list of free pages, and allocating one when needed).

A recursive construction. The gadget will log its keys in an array of pages, with $\frac{b}{\lg t}$ keys per page. As long as the last page of the log is incomplete, we do nothing with it, since queries can afford to inspect the page with $O(1)$ I/Os. The support queries to keys older than the last page, we maintain $\sqrt{t} + 1$ recursive $\sqrt{t}$-gadgets. The first is called the *top gadget*, while the rest are *bottom gadgets*.

When a page of the log fills, we insert the keys (with one batch INSERT operation) into the top gadget. This gadget supports keys from universe $[b] \times [\sqrt{t}]^2$ with associated data in $[(\sqrt{t})^3] = [t^{1.5}]$. Every key x is mapped as follows:

- the page hash is preserved;
- the first $\frac{1}{2} \lg t$ bits of the distribution and shadow hash codes are passed to the recursive gadget;

- the associated data becomes the number of the page in our log where the key resides. Since $|S| \geq O(bt)$, there are at most $O(t \lg t)$ pages in the log, so the data is within $[t^{1.5}]$.

After we have inserted $b\sqrt{t}$ keys into the top gadget, we flush it into the bottom gadgets. The top gadget is deallocated (its pages are returned to the free list), and a new empty gadget is constructed. A key x gets distributed to one of the $\sqrt{t}$ bottom gadgets depending on the first $\frac{1}{2}\lg t$ bits of its distribution hash. The key is passed down as follows:

- the page hash is preserved;
- the last $\frac{1}{2}\lg t$ bits of the distribution and shadow hash codes are passed to the recursive gadget;
- as above, the associated data becomes the number of the page in our log where the key resides, which fits within $[t^{1.5}]$.

We show that flushing data into the bottom gadgets can be done with one INSERT per gadget, plus I/O complexity $O(\sqrt{t}\lg t)$. The goal is to flush that last $b\sqrt{t}$ keys of the log, i.e. the last $b\sqrt{t}/\frac{b}{\lg t} = \sqrt{t}\cdot\lg t$ pages. These pages can be read into the cache (remember that $|S| \leq M$). We can then sort the keys by the distribution code (which is free inside cache). Then pack the key going to each gadget into the succinct page format, and call INSERT for each bottom gadget.

The QUERY(x) operation proceeds as follows:

- Inspect the last incomplete page of the log, and retrieve any occurrence of x from there, in $O(1)$ I/Os.
- Recurse in the top gadget with the top half of the distribution and shadow hash codes. This may introduce false positives, if other keys with the same top hash code currently reside in the top gadget. For each answer returned by the top gadget, we go to the page in the log indicated by the associated data of the key. If the key truly resides in the page, we retrieve the associated data, and append it to the list of occurrences. Otherwise, we discard this occurrence as a false positive.
- Recurse in the bottom gadget indicated by the first $\frac{1}{2}\lg t$ bits of the distribution hash of x. The bottom half of the distribution and shadow hashes are passed to the recursive call. For each returned occurrence, we read the appropriate page of the log, as above.

Observe that recursing in the bottom gadget cannot introduce a false positive due to the distribution hash, since it is only keys with the same top bits as x that appear in the gadget. However, it is possible that trimming the shadow hash code in half can introduce false positives.

Base case. We switch to the base case of the recursion when $t \leq t^{\min}$, for a parameter $t^{\min}$ to be determined. This gadget maintains a single *buffer page* with the last $\leq \frac{b}{\lg t}$ inserted keys. The rest of the keys are simply stored in a hash table (e.g. chaining) addressed by the *page hash*. With a succinct representation, the table occupies $O(|S|/\frac{b}{\lg t}) = O(t \lg t)$ pages.

QUERY(x) inspects the buffer page and only one page of the hash table w.h.p., so it takes time $O(1)$. UPDATE simply appends keys to the buffer page. When the buffer page fills, all keys are inserted into the hash table. This operation may need to touch all $O(t \lg t)$ pages, since the new keys are likely to have hash codes that are spread out.

2.2 Putting Things Together

In this section, we assemble our data structure out of gadgets, and analyze its performance. However, all probabilistic analysis is delayed until the next section.

Update cost. To analyze cost of inserting a key into a gadget, we assume that each INSERT cost comes with one unit of potential (that the parent gadget must pay for). When inserting a set T, the first things we do is to copy it to the log. The real cost of this is $O(\lceil |T|/\frac{b}{\lg t}\rceil)$. However, using our unit of potential, we can pay for the ceiling and deal with the case of small sets. Then, the amortized cost is $O(|T|/\frac{b}{\lg t})$, or $O(\frac{\lg t}{b})$ per key.

The gadget must also pay for pushing each key into two recursive $\sqrt{t}$-gadgets, one unit of potential for each recursive INSERT, and touching keys during the flush of the top gadget. Since the flush only pays for reading the keys by a scan, its cost is $O(\frac{\lg t}{b})$ amortized per key. Insertions into the top gadget also cost $O(\frac{\lg t}{b})$ per key, since we always insert full pages. Insertion to the bottom gadget is even cheaper: we run $\sqrt{t}$ INSERT's for every $b\sqrt{t}$ keys.

In the base case, we pay $O(t \lg t)$ I/Os for each page of key, so $O(\frac{t \lg^2 t}{b})$ per key. We have the recursion:

$$U(t) = O(\tfrac{\lg t}{b}) + 2 \cdot U(\sqrt{t}); \qquad U(t^{\min}) = O(\tfrac{1}{b} t^{\min} \lg^2 t^{\min})$$

On each level of the recurrence, we have 2^i terms of $O(\frac{1}{b}\lg(t^{-2^i}))$, i.e. a constant cost of $O(\frac{\lg t}{b})$ per level. This is very intuitive: at each level, our key is broken into many components, but their total size stays $\lg t$ bits. Since the cost is proportional to the bit complexity of the key, the cost of each level is constant.

The recursion has $O(\lg\lg t)$ levels. In the base case, the recursion has $\lg t / \lg t^{\min}$ leaves, each of cost $O(\frac{1}{b} t^{\min} \lg^2 t^{\min})$. The total cost is $U(t) = \frac{\lg t}{b} \cdot O(\lg\lg t + t^{\min} \lg t^{\min})$.

Query cost. To understand the query time, we note that there is a cost of at least 1 for each gadget traversed. The number of gadgets traversed is:

$$Q(t) = 1 + 2 \cdot Q(\sqrt{t}); \qquad Q(t^{\min}) = 1$$

The number of gadgets grows exponentially with the level, and is dominated by the base case. The total cost is $Q(t) = O(\lg t / \lg t^{\min})$.

In addition, there is a total cost of $O(\lg \frac{\lg t}{\lg t^{\min}})$ for each (true or false) match. Indeed, each key is in at most $O(\lg \frac{\lg t}{\lg t^{\min}})$ levels at a time, and we need to do $O(1)$ work per level: we go to the appropriate page in the log to verify the identity of the key and retrieve its data. We will show later that the total cost of false positives is $O(Q(t))$ w.h.p.

Beyond the cache size. Our data structure is organized like the (original) buffer tree, except that each node is implemented by a gadget to support fast queries. This top-level data structure can be implemented by the standard comparison solution. Formally, we have a tree with branching factor M^{ε}. Each node can store up to M keys. These keys are stored in a plain array, but also in an $\frac{M}{b}$-gadget. The gadgets are independent random constructions (with different i.i.d. hash functions). When the capacity of the node fills, its keys are distributed to the children. Since $M \geq B^{1+\varepsilon}$ (the tall cache assumption), this distribution is I/O efficient, costing $O(\frac{1}{B})$ per key.

The total cost for inserting a key is:

$$O(\tfrac{1}{B} \log_M n) + U(\tfrac{M}{b}) \cdot \log_M n = O(\tfrac{\log_M n}{B}) + O(\log_M n) \cdot \tfrac{\lg M}{B \lg n} \cdot (\lg\lg M + t^{\min} \lg t^{\min})$$

Thus $t_u = O(\frac{1}{B}) \cdot \big(\log_M n + \lg\lg M + t^{\min} \lg t^{\min}\big)$. Note that $\log_M n + \lg\lg M = \Theta(\log_M n + \lg\lg n)$. For any $\lambda \geq \log_M n + \lg\lg n$, we can achieve $t_u = O(\lambda/B)$ by setting $t^{\min} \lg t^{\min} = \lambda$.

The cost of querying a key is $Q(\frac{M}{b}) \cdot O(\log_M n) = O(\frac{\lg M}{\lg t^{\min}} \cdot \log_M n) = O(\frac{\lg n}{\lg t^{\min}}) = O(\log_\lambda n)$.

The total space of our data structure is linear. Each keys is stored in $O(\lg\lg M)$ gadgets at the same time, but its bit complexity is decreasing geometrically, i.e. the key accounts for $O(\lg n)$ bits in total. Since each gadget takes linear space, the total space must be $O(n)$.

2.3 Probabilistic Analysis

We first prove that no t-gadget receives more than $O(bt)$ keys w.h.p. Note that shadow and page hashes are irrelevant to this question, and we only need to analyze distribution hashes.

To understand the keys that land in some target gadget, we reason recursively. Starting with the root gadget and the entire set of keys, we move one step down towards the target gadget, trimming the set. When we move from a t-gadget to its top $\sqrt{t}$-subgadget, the analysis preserves the last $b\sqrt{t}$ keys, in time order, from the t-gadget. When we move from a t-gadget to a bottom $\sqrt{t}$-subgadget, the analysis fixes the top $\frac{1}{2}\lg t$ bits of all hash codes, and preserves the set of keys with the appropriate top half of the hash. For simplicity, we think of fixing the top half of the distribution hash for the entire data set, even those keys that already out of consideration. Both cases are pessimistic upper bounds.

In other words, the analysis fixes an increasing prefix of the bits of all hash codes. When it our target gadget, it has fixed the first h bits. Assume the last time we recursed to a top subgadget was after fixing h' bits, into a T-gadget. The top recursion always puts a worst case bound on the set. Thus, we have selected exactly $T \cdot b$ keys, by some procedure that depends on how the first h' bits were fixed. The bits on positions $[h'+1, h]$ decide which keys land in our target t-gadget. More precisely, all keys with a designated value on the positions $[h'+1, h]$ land in the target gadget, since we only do bottom recursions in the mean time. This is a standard ball-in-bins analysis, with a fixed set of $T \cdot b$ keys choosing independently and uniformly among $2^{h-h'}$ bins. We expect $t \cdot b$ keys in the target gadget. Since $b \gg \lg n$, a Chernoff bound shows we have at most $O(tb)$ keys with high probability in n. This holds for all gadgets by union bound.

We now switch to analyzing the number of false positive encountered by QUERY(x). It suffices to consider $n \leq M$, i.e. analyze pure gadgets. Indeed, the gadgets that a query traverses in the buffer tree are independently random, so we only get more concentration. We count a false positive only once, in the first *top* recursion when it is introduced. As noted above, the cost of such a false positive is $O(\lg \frac{\lg t}{\lg t^{\min}})$ if it is introduced in a t-gadget.

We will show that for any t, the number of the false positives introduced in all t-gadgets that the query traverses is $O(\log_t n)$ w.h.p. The cost is $O(\frac{\lg n}{\lg t} \cdot \lg \frac{\lg t}{\lg t^{\min}})$ w.h.p. Since $\lg t = 2^i \lg t^{\min}$, the sum of such costs is $O(\frac{\lg n}{\lg t^{\min}}) \sum i 2^{-i} = O(\frac{\lg n}{\lg t^{\min}}) = O(Q(t))$, as promised.

There are $\log_t n$ t-gadgets on the path of the query, and each cares about a disjoint interval of $\lg t$ bits of the distribution and shadow hash codes. We will fix a growing prefix of the distribution and shadow hashes, in increments of $\lg t$ bits. At every step, the fixing so far decides which keys land in the next t-gadget. Among these, there is a deterministic set of $\sqrt{t} \cdot b$ that can be in the top gadget at query time. We fix page hash of these particular keys (but leave all other page hashes random); we also fix the next $\lg t$ bits of all distribution and shadow hash codes. The number of collisions in the top recursion is a function of the outcome of these random choices. The number of collisions is a geometric random variable with rate $\sqrt{t}b/((\sqrt{t})^2 b) = 1/\sqrt{t}$. By construction of our process, the $\log_t n$ random variables describing all t-gadgets traversed are independent. Thus, the Chernoff bound shows that the cost exceeds $C \cdot \log_t n$ with probability $(1/\sqrt{t})^{\Omega(C \cdot \log_t n)} = n^{-\Omega(C)}$. Thus, any polynomially low probability can be achieved.

Our analysis holds even if the hash functions are $\Theta(\lg n)$-independent, by Chernoff bounds for limited independence [SSS95]. We remark that a k-independent distribution remains k-independent if we fix a prefix of the bits of the outcome. The necessary hash functions can be represented in $O(\lg n)$ words, and easily fit in cache.

3 Lower Bounds

Formally, our model of computation is defined as follows. The update and query algorithms execute on a processor with M bits of state, which is preserved from one operation to the next. The memory is an array of cells of $O(B \lg n)$ bits each, and allows random access in constant time. The processor is nonuniform: at each step, it decides on a memory cell as an arbitrary function of its state. It can either write the cell (with a value chosen as an arbitrary function of the state), or read the cell and update the state as an arbitrary function of the cell contents.

We first define our hard distribution. Choose S to be a random set of n keys from the universe $[2n]$, and insert the keys of S in random order. The sequence contains a single query, which appears at a random position in the second half of the stream of operations. In other words, we pick $t \in [\frac{n}{2}, n]$ and place the query after the t-th update (say, at time $t + \frac{1}{2}$).

Looking back in time from the query, we group the updates into epochs of λ^i updates, where λ is a parameter to be determined. Let $S_i \subset S$ be the keys inserted in epoch i; $|S_i| = \lambda^i$. Let $i_{\max}$ be the largest value such that $\sum_{i=1}^{i_{\max}} \lambda^i \le \frac{n}{2}$ (remember that there are at least $n/2$ updates before the query). We always construct $i_{\max}$ epochs. Let $i_{\min}$ be the smallest value such that $\lambda^{i_{\min}-1} \ge M$; remember that $M < n^{1-\varepsilon}$ was the cache size in bits. Our lower bound will generally ignore epochs below $i_{\min}$, since most of those elements may be present in the cache. Note $i_{\max} - i_{\min} \ge \varepsilon \log_\lambda n - O(1)$.

We have not yet specified the distribution of the query. Let $\mathcal{D}_{\text{NO}}$ be the distribution in which the query is chosen uniformly at random from $[2n] \setminus S$. Let $\mathcal{D}_i$ be the distribution in which the query is uniformly chosen among S_i. Then, $\mathcal{D}_{\text{YES}} = \frac{1}{i_{\max} - i_{\min} + 1}(\mathcal{D}_{i_{\min}} + \cdots + \mathcal{D}_{i_{\max}})$. Observe that elements in smaller epochs have a much higher probability of being the query. We will prove a lower bound on the mixture $\frac{1}{2}(\mathcal{D}_{\text{YES}} + \mathcal{D}_{\text{NO}})$. Since we are working under a distribution, we may assume the data structure is deterministic by fixing its random coins (nonuniformly).

Observation 3. *Assume the data structure has error C_ε on $\frac{1}{2}(\mathcal{D}_{\text{YES}} + \mathcal{D}_{\text{NO}})$. Then the error on $\mathcal{D}_{\text{NO}}$ is at most $2C_\varepsilon$. At least half the choices of $i \in \{i_{\min}, \ldots, i_{\max}\}$ are* good *in the sense that the error on $\mathcal{D}_i$ is at most $4C_\varepsilon$.*

We will now present the high-level structure of our proof. We focus on some epoch i and try to prove that a random query in $\mathcal{D}_{\text{NO}}$ reads $\Omega(1)$ cells that are somehow "related" to epoch i. To achieve this, we consider the following encoding problem: for $k = \lambda^i$, the problem is to send a random bit vector $A[1 \,..\, 2k]$ with exactly k ones. The entropy of A is $\log_2 \binom{2k}{k} = 2k - O(\lg k)$ bits.

The encoder constructs a membership instance based on its array A and public coins (which the decoder can also see). It then runs the data structure on this instance, and sends a message whose size depends on the efficiency of the data structure. Comparing the message size to the entropy lower bound, we get an average case cell-probe lower bound.

The encoder constructs a data structure instance as follows:

- Pick the position of the query by public coins. Also pick $S^\star = S \setminus S_i$ by public coins.
- Pick $Q = \{q_1, \ldots, q_{2k}\}$ a random set of k keys from $[2n] \setminus S^\star$, also by public coins.
- Let $S_i = \{q_j \mid A[j] = 1\}$, i.e. the message is encoded in the choice of S_i out of Q.
- Run every query in Q on the memory snapshot just before the position chosen in step 1.

The idea is to send a message, depending on the actions of the data structure, that will also allow the decoder to simulate the queries. Note that, up to the small probability of a query error, this allows the decoder to recover A: $A[j] = 1$ iff $q_j \in S$.

It is crucial to note that the above process generates an S that is uniform inside $[2n]$. Furthermore, each q_j with $A[j] = 0$ is uniformly distributed over $[2n] \setminus S$. That means that for each negative query, the data structure is being simulated on $\mathcal{D}_{\text{NO}}$. For each positive query, the data structure is being simulated on $\mathcal{D}_i$.

Let R_i be the cells read during epoch i, and W_i be the cells written during epoch i (these are random variables). We will use the convenient notation $W_{<i} = \bigcup_{j=0}^{i-1} W_j$.

The encoding algorithm will require different ideas for $t_u = 1 - \varepsilon$ and $t_u = o(1)$.

3.1 Constant Update Time

We now give a much simpler proof for the lower bound of [VZ10]: for any $t_u \le 1 - \varepsilon$, the query time is $t_q = \Omega(\log_B n)$.

We first note that the decoder can compute the snapshot of the memory before the beginning of epoch i, by simply simulating the data structure ($S_{>i}$ was chosen by public coins). The message of the encoder will contain the cache right before the query, and the address and contents of $W_{<i}$. To bound $|W_{<i}|$ we use:

Observation 4. *For any epoch i, $\mathbf{E}[|W_i|] \le \lambda^i t_u$.*

Proof. We have a sequence of n updates, whose average expected cost (by amortization) is t_u cell probes. Epochs i is an interval of λ^i updates. Since the query is placed randomly, this interval is shifted randomly, so its expected cost is $\lambda^i t_u$. □

We start by defining a crucial notion: the footprint $\mathrm{Foot}_i(q)$ of query q, relative to epoch i. Informally, these are the cells that query q would read if the decoder simulated it "to the best of its knowledge." Formally, we simulate q, but whenever it reads a cell from $W_i \setminus W_{<i}$, we feed into the simulation the value of the cell from before epoch i. Of course, the simulation may be incorrect if at least one cell from $W_i \setminus W_{<i}$ is read. However, we insist on a worst-case time of t_q (remember that we allow Monte Carlo randomization, so we can put a worst-case bound on the query time). Let $\mathrm{Foot}_i(q)$ be the set of cells that the query algorithm reads in this, possibly bogus, simulation.

Note that the decoder can compute $\mathrm{Foot}_i(q)$ if it knows the cache contents at query time and $W_{<i}$: it need only simulate q and assume optimistically that no cell from $W_i \setminus W_{<i}$ is read.

For some set X, let $\mathrm{Foot}_i(X) = \bigcup_{q \in X} \mathrm{Foot}_i(q)$. Now define the footprint of epoch i: $F_i = \big(\mathrm{Foot}_i(S_i) \cup W_i\big) \setminus F_{<i}$. In other words, the footprint of an epoch contains the cells written in the epoch and the cells that the positive queries to that epoch would read in the decoder's optimistic simulation, but excludes the footprints of smaller epochs. Note that $F_{<i} = W_{<i} \cup \big(\bigcup_{j=0}^{i-1} \mathrm{Foot}_j(S_j)\big)$.

We are now ready for the main part of our proof. Let $H_b(\cdot)$ be the binary entropy.

Lemma 5. *Let $i \in \{i_{\min}, \dots, i_{\max}\}$ be good and $k = \lambda^i$. Let p be the probability over $\mathcal{D}_{\mathrm{NO}}$ that the query reads a cell from F_i. We can encode a vector of $2k$ bits with k ones by a message of expected size: $O(\lambda^{i-1} t_q \cdot B \lg n) + 2k \cdot \big[H_b(\frac{1-\varepsilon}{2}) + H_b(\frac{p}{2}) + H_b(3C_\varepsilon)\big]$.*

Before we prove the lemma, we show it implies the desired lower bound. Choose $\lambda = B \lg^2 n$, which implies $\lambda^{i-1} t_q \cdot B \lg n = o(\lambda^i) = o(k)$. Note that $H_b(\frac{1-\varepsilon}{2})$ is a constant bounded below 1 (depending on ε). On the other hand $H_b(3C_\varepsilon) = O(C_\varepsilon \lg \frac{1}{C_\varepsilon})$. Thus, there exist a small enough C_ε depending on ε such that $H_b(1-\varepsilon) + H_b(3C_\varepsilon) < 1$. Since the entropy of the message is $2k - O(\lg k)$, we must have $H_b(1-\varepsilon) + H_b(3C_\varepsilon) + H_b(p) \ge 1 - o(1)$. Thus $H_b(\frac{p}{2}) = \Omega(1)$, so $p = \Omega(1)$.

We have shown that a query over $\mathcal{D}_{\mathrm{NO}}$ reads a cell from F_i with constant probability, for any good i. Since half the i's are good and the F_i's are disjoint by construction, the expected query time must be $t_q = \Omega(\log_\lambda n) = \Omega(\log_B n)$.

Proof of Lemma 5. The encoding will consist of:

1. The cache contents right before the query.
2. The address and contents of cells in $F_{<i}$. In particular this includes $W_{<i}$, so the decoder can compute $\mathrm{Foot}_i(q)$ for any query.
3. Let $X \subseteq Q \setminus S_i$ be the set of negative queries that read at least one cell from F_i. We encode X as a subset of Q, taking $\log_2 \binom{2k}{|X|} + O(\lg k)$ bits.
4. For each cell $c \in W_i \setminus F_{<i}$, mark some *positive* query $q \in S_i$ such that $c \in \mathrm{Foot}_i(q)$. Some cells may not mark any query (if they are not in the footprint of any positive query), and multiple cells may mark the same query. Let M be the set of marked queries. We encode M as a subset of Q, taking $\log_2 \binom{2k}{|M|} \leq \log_2 \binom{2k}{|W_i|}$ bits.
5. The set of queries of Q that return incorrect results (Monte Carlo error).

The decoder immediately knows that queries from X are negative, and queries from M are positive. Now consider what happens if the decoder simulates some query $q \in Q \setminus (X \cup M)$. If q reads some cell from $\mathrm{Foot}_i(M) \setminus F_{<i}$, we claim it must be a positive query. Indeed, $M \subset S_i$, so $\mathrm{Foot}_i(M) \setminus F_{<i} \subseteq F_i$. But any negative query that reads from F_i was identified in X.

Claim 6. *If a query $q \in Q \setminus (X \cup M)$ does not read any cell from $\mathrm{Foot}_i(M) \setminus F_{<i}$, the decoder can simulate it correctly.*

Proof. We will prove that such a query does not read any cell from $W_i \setminus F_{<i}$, which means that the decoder knows all necessary cells. First consider positive q. If q reads a cell from $W_i \setminus F_{<i}$ and q is not marked, it means this cell marked some other $q' \in M$. But that means the cell is in $\mathrm{Foot}_i(q') \subseteq \mathrm{Foot}_i(M)$, contradiction.

Now consider negative q. Note that $W_i \setminus F_{<i} \subseteq F_i$, so if q had read a cell from this set, it would have been placed in the set X. □

Of course, some queries may give wrong answers when simulated correctly (Monte Carlo error). The decoder can fix these using component 5 of the encoding.

It remains to analyze the expected size of the encoding. To bound the binomial coefficients, we use the inequality: $\log_2 \binom{n}{m} \leq n \cdot H_b(\frac{m}{n})$. We will also use convexity of $H_b(\cdot)$ repeatedly.

1. The cache size is $M \leq \lambda^{i-1}$ bits, since $i \geq i_{\min}$.
2. We have $\mathbf{E}[|F_{<i}|] \leq \mathbf{E}[|W_{<i}|] + |S_{<i}| \cdot t_q \leq O(\lambda^{i-1} t_q)$. So this component takes $O(\lambda^{i-1} t_q \cdot B \lg n)$ bits on average.
3. Since $\mathbf{E}[|X|] = pk$, this takes $2k \cdot H_b(\frac{p}{2})$ bits on average.
4. Since $\mathbf{E}[|W_i|] = t_u k \leq (1-\varepsilon)k$, this takes $2k \cdot H_b(\frac{1-\varepsilon}{2})$ bits on average.
5. The probability of an error is at most $2C_\varepsilon$ on $\mathcal{D}_{\mathrm{NO}}$ and $4C_\varepsilon$ on $\mathcal{D}_i$. So we expect at most $6C_\varepsilon \cdot k$ wrong answers. This component takes $2k \cdot H_b(3C_\varepsilon)$ bits on average.

This completes the proof of Lemma 5, and our analysis of update time $t_u = 1 - \varepsilon$.

3.2 Subconstant Update Time

The main challenge in the constant regime was that we couldn't identify which cells were the ones in W_i; rather we could only identify the queries that read them. Since $t_u \ll 1$ here, we will be able to identify those cells among the cells read by queries Q, so the footprints are no longer a bottleneck.

The main bottleneck becomes writing $W_{<i}$ to the encoding. Following the trick of [PT07], our message will instead contain $W_i \cap R_{<i}$ and the cache contents at the end of epoch i. We note that this allows the decoder to recover $W_{<i}$. Indeed, the keys $S_{<i}$ are chosen by public coins, so the decoder can simulate the

data structure after the end of epoch i. Whenever the update algorithm wants to read a cell, the decoder checks the message to see if the cell was written in epoch i (whether it is in $W_i \cap R_{<i}$), and retrieves the contents if so.

We bound $W_i \cap R_{<i}$ by the following, which can be seen as a strengthening of Observation 4:

Lemma 7. *At least a quarter of the choices of $i \in \{i_{\min}, \dots, i_{\max}\}$ are good and satisfy $\mathbf{E}[|W_i \cap R_{<i}|] = O(\lambda^{i-1} t_u / \log_\lambda n)$.*

Proof. We will now choose i randomly, and calculate $\mathbf{E}[|W_i \cap R_{<i}|/\lambda^i]$, where the expectation is over the random distribution and random i. We will show $\mathbf{E}[|W_i \cap R_{<i}|/\lambda^i] = O(t_u/(\lambda \log_\lambda n))$, from which the lemma follows by a Markov bound and union bound (ruling out the i's that are not good).

A cell is included in $W_i \cap R_{<i}$ if it is read at some time r that falls in epochs $\{0, \dots, i-1\}$, and the last time it was written is some w that falls in epoch i. For fixed $w < r$, let us analyze the probability that this happens, over the random choice of the query's position. There are two necessary and sufficient conditions for the event to happen:

- the boundary between epochs i and $i+1$ must occur before w, so the query must appear before $w + \sum_{j \le i} \lambda^j < w + 2\lambda^i$. Since the query must also appear after r, the event is impossible unless $r - w < 2\lambda^i$.
- the boundary between epochs i and $i-1$ must occur in (w, r), so there are at most $r - w$ favorable choices for the query. Note also that the query must occur before $r + \sum_{j<i} \lambda^j$, so there are at most $2\lambda^{i-1}$ favorable choices.

Let $i^\star$ be the smallest value such that $r - w < 2\lambda^{i^\star}$. Let us analyze the contribution of this operation to $\mathbf{E}[|W_i \cap R_{<i}|/\lambda^i]$ for various i. If $i < i^\star$, the contribution is zero. For $i = i^\star$, we use the bound $2\lambda^{i-1}$ for the number of favorable choices. Thus, the contribution is $O(\frac{\lambda^{i-1}}{n} \cdot \lambda^{-i}) = O(\frac{1}{n\lambda})$. For $i > i^\star$, we use the bound $r - w$ for the number of favorable choices. Thus, the contribution is $O(\frac{r-w}{n} \cdot \lambda^{-i}) = O(\frac{1}{n} \lambda^{i^\star - i})$.

We conclude that the contribution is a geometric sum dominated by $O(\frac{1}{n\lambda})$. Overall, there are at most $n \cdot t_u$ memory reads in expectation, so averaging over the choice of i, we obtain $\frac{1}{\log_\lambda n} \cdot O(\frac{1}{n\lambda}) \cdot n t_u = O(\frac{t_u}{\lambda \log_\lambda n})$. ☐

Our main claim is:

Lemma 8. *Let $i \in \{i_{\min}, \dots, i_{\max}\}$ be chosen as in Lemma 7 and $k = \lambda^i$. Let p be the probability over $\mathcal{D}_{\mathrm{NO}}$ that the query reads a cell from $W_i \setminus W_{<i}$. We can encode a vector of $2k$ bits with k ones by a message of expected size: $O(\lambda^{i-1} t_u \cdot B \lg \lambda) + O(k t_u \lg \frac{t_q}{t_u}) + 2k \cdot \left[H_b(\frac{p}{2}) + H_b(3C_\varepsilon)\right]$.*

We first show how the lemma implies the desired lower bound. Let λ be such that the first term in the message size is $\le \lambda^i = k$. This requires setting $\lambda = O(t_u B \lg(t_u B))$.

Note that the lower bound is the same for $t_u = 1 - \varepsilon$ and, say, $t_u = 1/\sqrt{B}$. Thus, we may assume that $t_u \le 1/\sqrt{B} \le 1/\sqrt{\lg n}$. Therefore $k t_u \lg \frac{t_q}{t_u} = O(k \cdot \frac{\lg\lg n}{\sqrt{\lg n}}) = o(k)$.

Since the entropy is $2k - o(\lg n)$, we must have $\frac{1}{2} + H_b(\frac{p}{2}) + H_b(3C_\varepsilon) \ge 1 - o(1)$. For a small enough C_ε, we obtain $H_b(\frac{p}{2}) = \Omega(1)$, so $p = \Omega(1)$.

We have shown that a query over $\mathcal{D}_{\mathrm{NO}}$ reads a cell from $W_i \setminus W_{<i}$ with constant probability, for at least a quarter of the choices of i. By linearity of expectation $t_q = \Omega(\log_\lambda n) = \Omega(\lg n / \lg(B t_u))$.

Proof of Lemma 8. The encoding will contain:

1. The cache contents at the end of epoch i, and right before the query.
2. The address and contents of cells in $W_i \cap R_{<i}$. The decoder can recover $W_{<i}$ by simulating the updates after epoch i.

3. Let $X \subseteq Q \setminus S_i$ be the set of negative queries that read at least one cell from $W_i \setminus W_{<i}$. We encode X as a subset of Q.
4. For each cell $c \in W_i \setminus W_{<i}$, mark some positive query $q \in S_i$ such that c is the first cell from $W_i \setminus W_{<i}$ that q reads. Note that distinct cells can only mark distinct queries, but some cells may not mark any query if no positive query reads them first among the set. Let M be the set of marked queries, and encode M as a subset of Q.
5. For each $q \in M$, encode the number of cell probes before the first probed cell from $W_i \setminus W_{<i}$, using $O(\lg t_q)$ bits.
6. The subset of queries from Q that return a wrong answer (Monte Carlo error).

The decoder starts by simulating the queries M, and stops at the first cell from $W_i \setminus W_{<i}$ that they read (this is identified in part 5 of the encoding). The simulation cannot continue because the decoder doesn't know these cells. Let $W^\star \subseteq W_i \setminus W_{<i}$ be the cells where this simulation stops.

The decoder knows that queries from M are positive and queries from X are negative. It will simulate the other queries from Q. If a query tries to read a cell from $W^\star$, the simulation is stopped and the query is declared to be positive. Otherwise, the simulation is run to completion.

We claim this simulation is correct. If a query is negative and it is not in X, it cannot read anything from $W_i \setminus W_{<i}$, so the simulation only uses known cells. If a query is positive but it is not in M, it means either: (1) it doesn't read any cell from $W_i \setminus W_{<i}$ (so the simulation is correct); or (2) the first cell from $W_i \setminus W_{<i}$ that it reads is in $W^\star$, because it marked some other query in M. By looking at component 6, the decoder can correct wrong answers from simulated queries.

It remain to analyze the size of the encoding:

1. The cache size is $M = O(\lambda^{i-1})$ bits.
2. By Lemma 7, $\mathbf{E}[|W_i \cap R_{<i}|] = O(\lambda^{i-1} \frac{t_u}{\log_\lambda n})$, so this component takes $O(\lambda^{i-1} t_u \log \lambda \cdot B)$ bits.
3. Since $\mathbf{E}[|X|] \leq p \cdot k$, this component takes $\mathbf{E}[\log_2 \binom{2k}{|X|}] \leq 2k \cdot H_b(\frac{p}{2})$ bits.
4. Since $\mathbf{E}[|W_i|] \leq \lambda^i t_u = k t_u$, this takes $\mathbf{E}[\log_2 \binom{2k}{|W_i|}] \leq k \cdot O(t_u \lg \frac{1}{t_u})$ bits.
5. This takes $\mathbf{E}[|W_i|] \cdot O(\lg t_q) = O(k t_u \lg t_q)$ bits.
6. We expect $2C_\varepsilon \cdot k$ wrong negative queries and $4C_\varepsilon \cdot k$ wrong positive queries, so this component takes $2k \cdot H_b(3C_\varepsilon)$ bits on average.

This completes the proof of Lemma 8, and our lower bound.

References

[Arg03] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. See also WADS'95.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. See also ICALP'87.

[BF03] Gerth S. Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proc. 14th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, 2003.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[PT07] Mihai Pătraşcu and Corina Tarniţă. On dynamic bit-probe complexity. *Theoretical Computer Science*, 380:127–142, 2007. See also ICALP'05.

[SSS95] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995. See also SODA'93.

[VZ10] Elad Verbin and Qin Zhang. The limits of buffering: a tight lower bound for dynamic membership in the external memory model. In *Proc. 41st ACM Symposium on Theory of Computing (STOC)*, pages 447–456, 2010.

[Yi10] Ke Yi. External memory data structures with $o(1)$-I/O updates. Dagstuhl Proceedings for "Data Structures" Meeting, 2010.

[YZ10] Ke Yi and Qin Zhang. On the cell probe complexity of dynamic membership. In *Proc. 21st ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 123–133, 2010.