Lower Bounds for External Memory Dictionaries

Gerth Stølting Brodal[*,†] Rolf Fagerberg[*]

Abstract

We study trade-offs between the update time and the query time for comparison based external memory dictionaries. The main contributions of this paper are two lower bound trade-offs between the I/O complexity of member queries and insertions: If $N > M$ insertions perform at most $\delta \cdot N/B$ I/Os, then (1) there exists a query requiring $N/(M \cdot (\frac{M}{B})^{\mathcal{O}(\delta)})$ I/Os, and (2) there exists a query requiring $\Omega(\log_{\delta \log^2 N} \frac{N}{M})$ I/Os when δ is $\mathcal{O}(B/\log^3 N)$ and N is at least M^2. For both lower bounds we describe data structures which give matching upper bounds for a wide range of parameters, thereby showing the lower bounds to be tight within these ranges.

1 Introduction

In this paper, we consider the complexity of maintaining dictionaries in external memory. The computational model used is the I/O model of Aggarwal and Vitter [1], which assumes a memory hierarchy containing two levels: an internal memory of size M, and an external memory of unbounded size. The transfer between the two levels takes place in blocks of B elements, where $M \geq 2B$. The size of the problem is denoted N, and the measure of cost is the number of blocks transferred. This model is adequate when the memory transfer between two levels of the memory hierarchy dominates the running time, which is often the case when the size of the data significantly exceeds the size of main memory, due to the very large access time for disks compared to the remaining levels of the memory hierarchy. During the last decade, a large number of results for the I/O model has been developed—see e.g. the surveys by Arge [3] and Vitter [19].

As in the sorting lower bound of Aggarwal and Vitter [1], we consider a comparison based version of the I/O model, where the operations allowed on elements are moving, copying, deleting, and comparing, and where two elements only can be compared if they are both in internal memory. An I/O touches B contiguous memory addresses in external memory.

The B-tree, introduced in 1972 by Bayer and McCreight, is a comparison based external memory dictionary which works well in the I/O model, supporting member, predecessor, successor, and range queries as well as insertions and deletions. They are widely used in practice, in particular in databases, which typically store large indexes as B-trees (or variants hereof, such as B$^+$-trees and B*-trees [14, 15]). B-trees support updates as well as member, predecessor, and successor queries in $\mathcal{O}(1 + \log_B \frac{N}{M})$ I/Os. It is well-known that in a comparison based model, this number of I/Os is best possible for member queries.

*BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,rolf}@brics.dk. Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

†Supported by the Carlsberg Foundation (contract number ANS-0257/20).

The question we raise in this paper is, whether the I/O cost for updates can be lowered. More generally, we ask what trade-offs can be achieved between update costs and query costs in comparison based external memory dictionaries. For simplicity, we restrict our attention to insertions and member queries. The lower bounds hold for the other types of queries as well, as member queries are a special case of these.

In internal memory, the corresponding question is well solved. First of all, a simple adversary argument shows that a query can be forced to use $\log_2 N$ comparisons, no matter what the insertion cost is. More generally, lower bounds have been proven [10, 11], stating that if insertions perform at most $\mathcal{O}(k)$ comparisons, then queries can be forced to use $\max\{\log_2 N\,,\, N/2^{\Theta(k)}\}$ comparisons. These bounds are asymptotically tight, as can be seen by considering balanced binary search trees where the subtrees rooted at depth $\Theta(k)$ have been substituted by unordered lists of size $N/2^{\Theta(k)}$.

In external memory, it is well-known that an adversary argument (see Lemma 1) analogous to the internal case gives a lower bound for queries of $\Omega(\log_B \frac{N}{M})$ I/Os, no matter what the insertion cost is. However, unlike the internal case, not much is known about the possible combinations of query and insertion cost when the insertion cost is below that of external memory search trees. This is the subject of the present paper. We summarize the situation in Figure 1.

	Internal	*External*
Query	$\Omega(\log_2 N)$ always	$\Omega(\log_B \frac{N}{M})$ always
Search Trees	Insert, Query $\Theta(\log_2 N)$	Insert, Query $\Theta(\log_B \frac{N}{M})$
Trade-Off	Insert $\Theta(k) \Rightarrow$ Query $N/2^{\Theta(k)}$	*This paper*

Figure 1: Insertions and queries in comparison based dictionaries

By the optimality of the query bound for B-trees, there is no need to consider insertion costs larger than for B-trees. On the other hand, if N insertions perform less than $(N-M)/B$ writes to external memory, then some elements have been lost by the dictionary, and member queries clearly cannot be answered correctly. In other words, we are interested in which query costs are possible when the cost of N insertions is $\delta \cdot N/B$ I/Os for $1 \leq \delta \leq B \log_B N$.

We note that the situation in external memory differs from that in internal memory in a fundamental way: in the comparison based I/O model, the cost of sorting N elements is $\text{Sort}(N) = \mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{M})$ I/Os [1], which is smaller than the cost $\mathcal{O}(N \log_B \frac{N}{M})$ of performing N operations on a B-tree. This is in contrast to the standard (internal) comparison model, where the cost per element is the same for sorting and for searching, namely $\Theta(\log N)$. Intuitively, this means that while information theoretical arguments in the internal case are adequate for all insertion costs below the searching lower bound, this is in the external case only true for small costs, namely costs with $1 \leq \delta \leq \log_{M/B} \frac{N}{M}$. For the remaining values of δ (the major part of its range for most choices of parameters N, B, and M), the fact that the dictionary algorithm does not know *when* the query will be asked must be used when proving lower bounds—if the algorithm knew the query time in advance, it could simply sort the elements and build a B-tree immediately before the query without violating the insertion bound.

Of previous work, only little is directly related to our question. Aggarwal and Vitter [1] proved comparison based lower bounds for external sorting. Arge [2] introduced the buffer tree as an external memory search tree to handle sequences of batched updates and queries. A sequence of N updates and queries is handled in $\mathcal{O}(\mathrm{Sort}(N))$ I/Os, assuming that the answer to queries are not required to be reported before the complete sequence has been processed, i.e. queries are not online. If used as an online data structure, buffer-trees support a sequence of N updates using $\mathcal{O}(\mathrm{Sort}(N))$ I/Os and answers an online query with $\mathcal{O}(\frac{M}{B}\log_{M/B}(N/M))$ I/Os. In [4], Arge et al. introduce an I/O version of comparison trees, which gives a general way to transfer standard (internal) comparison lower bounds for offline problems, such as sorting and element distinctness, into I/O bounds for the same problems.

Of more remotely related work, Samoladas et al. have considered multi-dimensional range query trade-offs between storage redundancy and access overhead in [7, 16, 17]. A systematic approach for deriving lower bounds on the trade-off between access overhead and redundancy can be found in [16]. These trade-offs are proved for static problems only. Trade-offs between time and space for sorting and element distinctness are given by Arge and Pagter in [6]. Arge and Miltersen [5] discusses other models than the comparison based I/O-model, and prove lower bounds for various problems, none of which involves online queries.

The main contributions of this paper are (2) and (3) in the following theorem, which summarizes Section 2.

Theorem 1 *If N insertions perform at most $\delta \cdot N/B$ I/Os, then*

(1) There exists a query requiring at least $\log_{B+1}\frac{N}{M} - \mathcal{O}(1)$ I/Os.

(2) There exists a query requiring $N/(M \cdot (\frac{M}{B})^{\mathcal{O}(\delta)})$ I/Os for $N > M$.

(3) There exists a query requiring $\Omega(\log_{\delta\log^2 N}\frac{N}{M})$ I/Os, provided $\delta \leq B/\log^3 N$ and $N \geq M^2$.

The first lower bound is the folklore statement that B-trees support queries in the optimal number of I/Os. The second is proved essentially by adapting the information theoretical methods used in [10, 11] to the comparison based external memory. The third is proved by constructing an adversary which chooses the ordering of newly inserted elements adaptively to the actions of the algorithm, and is able to build a set of elements, for which the algorithm does not know the order of any pair, and for which the elements all reside in different blocks in external memory. A query among these gives the lower bound.

In Section 3, we show that the lower bounds (2) and (3) are optimal for a wide range of parameters by describing data structures with upper bounds matching the lower bounds within the ranges.

Graphically, our contributions can be depicted as in Figure 2. In the figure, the x-axis depicts the number of I/Os per insertion, and the y-axis depicts the worst-case number of I/Os for a query. The solid lines denote matching upper and lower bounds. The three upper bounds are from left to right achieved by respectively truncated buffer trees, B-trees with buffers, and B-trees, as described in Section 3. We note that the lower bound $\Omega(\log_{\delta\log^2 N}\frac{N}{M})$ is equal to $\Omega(\log_{\delta}\frac{N}{M})$ for the values of δ stated for the middle curve. We also note that if $B \geq \log^{3+\epsilon} N$ for some constant $\epsilon > 0$, B-trees with buffers have asymptotically optimal queries of $\mathcal{O}(\log_{B+1}\frac{N}{M})$ when the cost of inserts is in the range $1/\log^3 N$ to $\log_{B+1}\frac{N}{M}$.

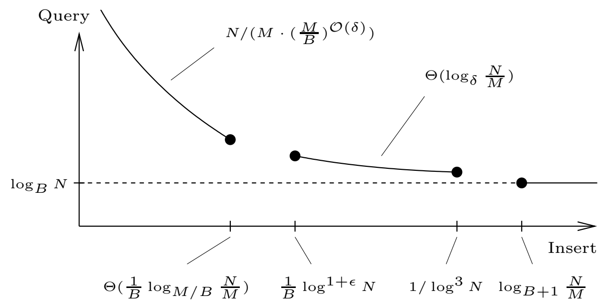

Figure 2: Trade-off between insertions and queries.

2 Lower bounds

In this section, we prove lower bound trade-offs between the insertion cost and the query cost for comparison based external memory dictionaries. We first recall the folklore lower bound for a single query, and then give our two main lower bounds.

2.1 Any number of I/Os per update

We give a proof for a lower bound on the number of I/Os for a single query. The lower bound even holds for the static case with no assumption on the redundancy in the stored dictionary, i.e. the data structure is allowed to have arbitrary size.

Lemma 1 *For any dictionary storing N elements, there exists a query requiring at least $\log_{B+1} \frac{N}{M} - \mathcal{O}(1)$ I/Os.*

Proof. The proof is by an adversary argument. The elements that still can be equal to the query element are denoted *candidate elements*. These always form a consecutive subsequence in the ordering of the N elements. Initially, at most M elements can reside in internal memory. The adversary can now select the answer to all comparisons between elements in internal memory and the query element such that there are at least $\frac{N-M}{M+1} > \frac{N}{M+1} - 1$ candidate elements left matching the query element. Each succeeding I/O reads B elements. If there are k candidate elements remaining before the I/O, then the adversary can choose the answers to the comparisons such that there are at least $\frac{k-B}{B+1} > \frac{k}{B+1} - 1$ candidate elements left, since the B elements read partition the at least $k - B$ remaining candidate elements into $B + 1$ consecutive subsequences, of which the largest has size at least $\frac{k-B}{B+1}$. By induction it follows that after i I/Os there are at least $\frac{N}{(M+1)(B+1)^i} - 2$ candidate elements. Since the adversary can consistently answer both member and not-member while there is at least one candidate element left, the lemma follows. □

2.2 Few I/Os per B updates

In this section we give a lower bound for the case when the number of I/Os done by a sequence of insertions is not sufficient to sort the set of inserted elements. The proof of the lemma below uses an adversary argument inspired by the adversary construction used in [11] for proving a comparison based lower bound trade-off between updates and queries. The interesting values of δ in the following lemma are $\delta < \log_{M/B} \frac{N}{M}$.

Lemma 2 *If N insertions into an initial empty dictionary perform at most $\delta \cdot N/B$ I/Os, then there exists a query requiring $N/(M \cdot (\frac{M}{B})^{\mathcal{O}(\delta)})$ I/Os for $N > M$.*

Proof. We will describe an adversary which during the inserting of N elements can provide consistent answers such that there exists a query with the claimed complexity.

The adversary strategy is the following. The N input elements $e_1, \ldots, e_N$ are initially placed at the root of an infinite binary tree. The placement of the elements on the infinite tree is used to capture the partial ordering of the elements revealed by the adversary so far. We let $v(e_i)$ denote the node where e_i is stored, and $d(v)$ the depth of a node, with the root having depth zero. We say that two elements e_i and e_j are *ordered* if e_i and e_j are stored at distinct nodes of the tree where no node is the ancestor of the other node. To answer comparisons consistently, the adversary will move the elements down in the tree such that when the adversary for two elements e_i and e_j answers $e_i < e_j$, then e_i and e_j are ordered and e_i is stored to the left of e_j in the tree. The important property is that moving an element from a node to a descendant cannot violate any of the order answers given so far.

The adversary maintains the following invariants with respect to the placement of the elements on the infinite binary tree.

1. Elements in internal memory are pairwise ordered.
2. Inserted elements are stored at nodes with depth at least $\lceil \log M \rceil$.
3. The elements stored in a block in external memory are pairwise ordered.

To maintain these invariants we consider the following cases. The first case is when an element e_i is inserted. Then e_i is moved from the root to a node with depth $\lceil \log M \rceil$ such that e_i is ordered with the at most $M-1$ elements already in internal memory. This is possible since there are at least M nodes with depth $\lceil \log M \rceil$, and by invariant 2 the at most $M-1$ elements already in internal memory after the insertion are stored at nodes with depth at least $\lceil \log M \rceil$.

Writing a block of B elements from internal memory to external memory does not require any action by the adversary to satisfy invariant 3, since the B elements written are already pairwise ordered by invariant 1.

Finally, we consider the case when a block of B elements is read from external memory into internal memory. By invariant 1 the $M-B$ elements already in internal memory are pairwise ordered, and by invariant 3 the B elements read are pairwise ordered. If e_i is an element read, we have that $v(e_i)$ is an ancestor of zero or more nodes storing elements in internal memory. Let n_i be the number of elements from internal memory stored at $v(e_i)$ or at a descendant of $v(e_i)$. Then there exists a descendant w of $v(e_i)$ such that $d(w) = d(v(e_i)) + \lceil \log(n_i + 1) \rceil$ and neither w or any descendant of w stores elements in internal memory. The adversary moves e_i to w. If an ancestor u of w stores an element e_j from internal memory (by invariant 1 there is at most one such node and element), then e_j is moved to the child of u not having w as a descendant. This ensures that e_i becomes pairwise ordered to all elements in internal memory. We repeat this for each of the B elements read. This concludes the description of the adversary strategy.

We now consider the sum of the depths of the elements, i.e. $\sum_{i=1}^{N} d(v(e_i))$. The initial sum equals zero. An insertion increases the sum by $\lceil \log M \rceil$. A block write does not change the sum. For a block read, note that $\sum_{i \in \mathcal{I}} n_i \leq M - B$, where $i \in \mathcal{I}$ if and only if e_i is among the B elements read. The sum increases by at most $B + \sum_{i \in \mathcal{I}} \lceil \log(n_i + 1) \rceil < 2B + \sum_{i \in \mathcal{I}} \log(n_i + 1) \leq 2B + B \log \frac{M}{B}$, where the last inequality follows from the convexity of the logarithm and the fact that $\sum_{i \in \mathcal{I}}(n_i + 1) \leq M$. It follows that N insertions performing a total of x block reads implies

$$\sum_{i=1}^{N} d(v(e_i)) \leq N \lceil \log M \rceil + x \cdot B \left(2 + \log \frac{M}{B} \right) .$$

For trees storing at most q elements in each node, the minimal value of $\sum_{i=1}^{N} d(v(e_i))$ is attained by a binary tree of perfect balance, where all nodes except one contain q elements. From this follows that if $\sum_{i=1}^{N} d(v(e_i)) \leq \alpha N$, then there exists a node storing $\Omega(N/2^{\alpha})$ elements. Since these elements are pairwise unordered, we by invariant 1 have that at most one of the elements is stored in internal memory and by invariant 3 each block in external memory can at most contain one of the elements. The adversary can now force the query algorithm to compare the query key with all these elements, i.e. the query takes at least $\Omega(N/2^{\alpha}) - 1$ I/Os, since all elements must be read using one I/O per element except for the single element already in internal memory.

Setting $x = \delta \cdot N/B$ we get $\alpha = \lceil \log M \rceil + \delta(2 + \log \frac{M}{B}) \leq \log M + \mathcal{O}(\delta \log \frac{M}{B})$. The total number of I/Os for the query becomes $N/2^{\log M + \mathcal{O}(\delta \log \frac{M}{B})} - 1 = N/(M \cdot (\frac{M}{B})^{\mathcal{O}(\delta)}) - 1$. The lemma follows, since for $N > M$, the adversary can force the algorithm to perform at least one I/O. □

2.3 Many I/Os per B updates

In this section, we give a lower bound for the case where the number of I/Os performed during insertions is above the sorting bound.

Our proof is an adversary argument. The adversary makes up to N insertions into an initial empty structure, ending with one query operation. To provide consistent answers to comparisons, the adversary maintains a total order on the inserted elements. For each new element, the adversary chooses the position in the ordering among the already inserted elements. The adversary also chooses when to make the single query, and for which element to query.

The aim of our adversary is to end up in a situation where the conditions in the following lemma hold:

Lemma 3 *Let I be the set of elements inserted so far, with the ordering chosen by the adversary, and let $S \subseteq I$ be a set which forms a consecutive sequence in this ordering. Assume the following holds:*

1. *The internal memory contains no copies of elements in S.*
2. *For all $x, y \in S$ $(x \neq y)$:*
   (a) *No block in external memory contains a copy of both x and y.*
   (b) *No copies of x and y have been compared by the algorithm.*

Then the adversary can force a query to use at least $|S|$ I/Os.

Proof. In the partial order induced by the comparisons performed by the algorithm so far, the elements of S form an anti-chain. The adversary can therefore choose any reordering of the elements of S without being inconsistent. In particular, the adversary can force the algorithm to compare a search key s with (a copy of) all elements $x \in S$ by declaring $x < s$ for the first $|S| - 1$ such comparisons—it is still free to choose between $x = s$ and $x \neq s$ for the last. By the conditions in the lemma, one I/O is needed for each such comparison. $\square$

In the proof below, we use a probabilistic argument to show that the adversary is able to make choices fulfilling various conditions. We consider a random choice, and by Chernoff bounds prove a non-zero probability of the desired conditions being true. Hence, at least one of the possible choices fulfills these conditions.

Lemma 4 *Let $\delta > 0$ be arbitrary, and assume $B \geq \delta \log^3 N$ and $N \geq M^2$. If N insertions into an initial empty dictionary perform at most $\delta \cdot N/B$ I/Os, then there exists a query requiring $\Omega(\log_{\delta \log^2 N} \frac{N}{M})$ I/Os.*

Proof. We first define the following values:

$$p = (\delta e \log^2 N)^{-1}$$

$$w_i = N(\frac{p}{4})^i \text{ for } i \geq 0$$

$$s_i = w_i / \log N \text{ for } i \geq 0$$

Let $x_1, x_2, \ldots, x_N$ be the sequence of N elements to be inserted. We partition the sequence into segments by a inductive procedure. A step at level i in the induction consists of splitting a segment into a first part of length s_i, and then splitting the rest into smaller segments, each of length $\Theta(w_{i+1})$. We denote the first part the *base segment* and the segments in the second part the *sub segments* of the step. At level zero of the induction, the partitioning step is applied to the entire sequence, and in each higher level, the partitioning step is applied to all sub segments from the previous level. The first two levels of the partitioning is illustrated in Figure 3, where the horizontal line depicts the sequence of insertions. We stop the induction at some level $K \leq \log N - 1$, where K is an integer to be chosen later.

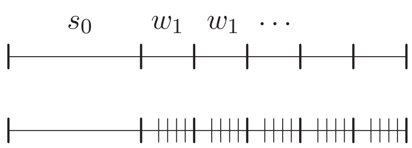

Figure 3: Two levels of the partitioning process

The partitioning of the sequence may be viewed as a tree, where nodes correspond to segments: the root is the entire sequence, and the children of a node v are the sub segments of the segment of v. Note that if we by a visit of a node mean the traversal of its base segment, then a traversal of the insertion sequence corresponds to a preorder traversal of the tree.

The strategy of the adversary is to use the partitioning above to find a point in the insertion sequence where few I/Os have been done which touches previously inserted elements.

By assumption, over the entire sequence the algorithm makes an average number of I/Os per insertion of at most δ/B I/Os. As the segments on level one (i.e. the sub segments of level zero) covers a fraction $(\log N - 1)/\log N$ of w_0, the average number of I/Os per insertion in the sub segments is at most $\delta/B \cdot \log N/(\log N - 1)$. Hence, no matter what actions the algorithm takes, there will exist at least one segment on level one inside which the average number of I/Os is at most this value. In this segment, at most $\mathcal{O}(w_1\delta/B \cdot \log N/(\log N-1))$ I/Os are made.

We denote the first of these the *sparse* segment σ_1 on level one, and extend the definition to higher levels by induction: the sparse segment σ_{i+1} on level $i+1$ is the first of the sub segments within σ_i for which the average number of I/Os per insertion is at most $\delta/B \cdot (\log N/(\log N-1))^i$. The sparse segments $\sigma_1 \supseteq \sigma_2 \supseteq \cdots \supseteq \sigma_K$ correspond to a root-to-leaf path in the tree view of the partition. At most $w_i\delta/B \cdot (\log N/(\log N-1))^i \leq ew_i\delta/B$ I/Os are made in σ_i, where the last inequality follows from $K \leq \log N - 1$ and $(1+1/x)^x < e$.

By counting I/Os, the adversary can online keep track of when a sub segment on level i is no longer a candidate for being the sparse segment on level i. By the existence of the sparse segments, a point in the insertion sequence is eventually reached which is at the end of a segment on level K, and for which the corresponding path in the tree consists of nodes representing segments which are still candidates for being sparse segments. The adversary will make a query at this point.

The point in time where the adversary makes the query corresponds to a root-to-leaf path $v_1, v_2, \ldots, v_K$ in the tree view of the partitioning. The nodes in the path represents the base parts of some segments. Except for the one in v_K, the segments may not have been passed completely, but we know that at most $ew_i\delta/B$ I/Os have been made up to the current point in the insertion sequence. As the adversary terminates the insertion process here, we will still call these segments the sparse segments $\sigma_1, \sigma_2, \ldots, \sigma_K$.

We will now describe how the adversary in an online fashion chooses the position of the inserted elements in the order. The elements $S_0 = \{x_1, x_2, \ldots, x_{s_0}\}$ inserted during the base segment at level zero are assigned the ordering $x_1 < x_2 < \ldots < x_{s_0}$. These will serve as fixed points in the order, next to which the rest of the inserted elements are placed. In base segments at level one, the adversary will make a sample of S_0 and in the ordering place the new elements next to the elements of the sample. In base segments at level two, the adversary will use a sample of this sample, and so forth for higher levels.

More precisely, let S_0 be the sample of the root of the tree representing the partitioning. When the preorder traversal of the tree reaches a child v of the root, the adversary makes a sample S'_v of S_0 by including each element in S_0 independently with probability p. The expected value of $|S'_v|$ is $p|S_0| = ps_0$, so by Chernoff bounds, $|S'_v| \geq ps_0/4$ with very high probability, i.e. the probability of failure is exponentially decreasing in the expected value. If failure occurs, the adversary gives up (remember, we only have to prove a non-zero probability of the adversary succeeding). Otherwise, the adversary sets S_v equal to the first $ps_0/4 = s_1$ elements of S'_v. The elements inserted during the base segment of v are associated arbitrarily in a one-to-one fashion to the elements of S_v, and if y is associated to x_i, then y is placed as the current predecessor of x_i in the order.

For a node v on level two, consider the point in time where the preorder traversal reaches v, i.e. the point in the insertion sequence at the start of the base segment in v. Let x_i be a member of the sample S_u at the parent u of v. We say that x_i is *alive* if no copy of x_i has been touched by an I/O in the time interval from the start of the segment at u to the start of the segment of v. Otherwise, we say it has been *killed*. Here, we for the moment assume that all elements are in external memory, and later take the effect of internal memory into account.

Consider a block in external memory, at the point in time where the insertion sequence reaches the start of the segment at u. It holds at most copies of B different elements from the sample at the parent of u (the root). Each of these were sampled with probability p to produce the sample at u. Hence, the expected number of elements killed by an I/O touching this block is at most Bp. By the assumption $B \geq \delta \log^3 N$, we have $Bp \geq (\log N)/e$. By Chernoff bounds, the probability that the number of elements killed is more than a constant factor from Bp is exponentially decreasing in Bp. Hence, this probability is less than $1/N^{\alpha}$, where the value of α may be varied by changing constants in our setup. The algorithm in total makes at most $\delta N/B$ I/Os, which by the assumption $B \geq \delta \log^3 N$ is less than N. Hence, the probability that *no* block can kill more than a constant factor times Bp is $1/N^{\alpha-1}$. If the segment at u is still a candidate for being σ_1, the number of I/Os made in it so far is bounded by $ew_1\delta/B$, as seen above. In that case, the number of elements in S_u which is not alive at the start of the segment at v is at most a constant times $ew_1(\delta/B)Bp = w_1/\log^2 N$. As $w_1/\log N = s_1$, the fraction of S_u killed is vanishing. Hence, the adversary samples all live elements of S_u with probability p, and as before is with high probability able to produce a reduced sample S_v of size s_2. The elements inserted during the base segment of v are associated in a one-to-one fashion to the elements of S_v, and if y is associated to x_i, then y is placed as the current predecessor of the element of the base segment of u which is associated to x_i. If the segment at u is no longer a candidate, the adversary has no use for the ordering of the elements inserted during the the rest of the preorder traversal of the subtree rooted at u, and simply sets their order to ∞, i.e. makes them all new largest elements during that part of the traversal (no sampling occurs).

The above behavior of the adversary happens inductively: When the preorder traversal reaches a node v, where all ancestors in the tree are still candidates for being the sparse segments, it samples each live element x_i of the sample S_u at its parent u with probability p. The term *alive* is extended to mean that for no ancestor z of u has any copy of the element in the base segment of z which is associated with x_i been touched by an I/O. Since x_i was alive when it was sampled at u, any touching has been done since the start of the segment at u. Hence, an argument analogous to the level two case above applies.

There are at most N nodes in the tree. We proved that at each node, the probability that the sample will make it possible for the algorithm to kill too many elements from the sample is at most $1/N^{\alpha-1}$. The algorithm can in the worst case force the adversary to move through all nodes of the tree by deciding which segments are the sparse segments. The probability that the adversary will fail at all is at most $1/N^{\alpha-2}$.

Setting up constants such that $\alpha > 2$ shows that with non-zero probability, the adversary will succeed with its strategy until the point it makes the query.

In the above, we have assumed that all elements were in external memory. The effect of internal memory is to kill M extra in each sampling step. This can only remove M from the sample for each level of the path, of which there are at most $\log N$. If we choose a K such that σ_K still contains at least $N^{3/4}$ when not considering the effect of internal memory, the assumption of $N \geq M^2$ means that there are still elements alive in the sample of σ_K.

It follows by induction on the levels that if the insertion point is in a base segment in a node for which all ancestors are still candidates for being the sparse sets, then if the inserted element y is associated to x_i, then the set of elements in the ancestors which also are associated to x_i is a set of consecutive elements in the current order.

In particular, for the at least one element y alive in σ_K when the adversary makes the query, the set S of all elements in ancestors which are associated to the same x_i as y is a set of consecutive elements in the order. By the definition of being alive, for no pair of elements in S does it hold that they have been compared or can reside in the same block (since they have not been touched by an I/O). Hence, S is a set of size K fulfilling the assumptions in Lemma 3.

We must choose K such that $s_K > N^{3/4}$. As $s_i = (p/4)^i N/\log N$, we can choose $K = \Theta(\log_{4/p} N) = \Theta(\log_{\delta \log^2 N} N)$. As $N \geq M^2$, we can substitute N/M for N. $\square$

3 Upper bounds

Here we discuss various upper bounds for external memory dictionaries. The main purpose of these constructions is to show that the lower bounds in Theorem 1 are asymptotically tight for a wide range of parameters.

3.1 B-trees

A common solution for maintaining a dictionary in external memory is to use a B-tree [8]. A B-tree is a multi-way tree where each leaf stores $\mathcal{O}(B)$ elements, internal nodes (except the root) have degree $\Theta(B)$, and all leaves have equal depth. Each internal node is stored in $\mathcal{O}(1)$ blocks and consists of pointers to the children and appropriate search keys to guide the queries. The degree bound of a B-tree implies that the height is $\mathcal{O}(\log_B N)$, and that queries and updates can be done with $\mathcal{O}(\log_B N)$ I/Os [8].

By maintaining the topmost $\mathcal{O}(M)$ elements of a B-tree in internal memory, updates and queries can be performed in $\mathcal{O}\left(\log_B \frac{N}{M}\right)$ I/Os: The topmost $\mathcal{O}(M)$ elements will be stored at levels $0, 1, \ldots, \ell$, where all of levels $0, 1, \ldots, \ell - 1$ can be placed in internal memory. Level $\ell + 1$ contains $\Omega(M)$ nodes, i.e. at least one subtree rooted at level $\ell + 1$ has size $\mathcal{O}(\frac{N}{M})$ and height $\mathcal{O}(\log_B \frac{N}{M})$. It follows that the B-tree has height $\ell + \mathcal{O}(\log_B \frac{N}{M})$ and queries require $\mathcal{O}(1 + \log_B \frac{N}{M})$ I/Os. By Lemma 1, this query time is optimal.

3.2 Buffer trees

The buffer trees of Arge [2] are a variant of B-trees where each internal node has degree $\Theta(M/B)$ and each leaf stores $\mathcal{O}(M)$ elements. Furthermore each internal node has a buffer of size $\mathcal{O}(M)$ containing delayed operations to be propagated down in the subtree rooted at the node. We refer the reader to [2] for a detailed treatment of buffer trees.

Buffer trees are designed to perform offline queries and updates efficiently. Used in an online setting, buffer trees support N updates with $\mathcal{O}(\mathrm{Sort}(N))$ I/Os, as argued by Arge. Online queries can be supported in $\mathcal{O}(\frac{M}{B} \log_{M/B} \frac{N}{M})$ I/Os, since all buffers along one root-to-leaf path must be examined when performing a query. While the I/O bound for updates is significantly better than for B-trees, namely a factor $B \cdot \log_B \frac{M}{B}$, online queries are a factor $\frac{M}{B} / \log_B \frac{M}{B}$ slower.

3.3 Truncated buffer trees

In the following, we outline a construction which achieves an upper bound corresponding to Lemma 2. The construction is basically a truncated version of buffer trees. Assume that N insertions perform $\mathcal{O}(\delta \cdot N/B)$ I/Os. Instead of maintaining all $\mathcal{O}(\log_{M/B} \frac{N}{M})$ levels of a buffer tree, we only maintain the topmost δ levels, i.e. all nodes have degree $\mathcal{O}(M/B)$ and a buffer of size $\mathcal{O}(M)$. Instead of maintaining levels $\delta + 1, \delta + 2, \ldots$ of a buffer tree, we associate a bucket to each node at level δ. Whenever there is a buffer overflow at a node at level δ, the content of the buffer is sorted and moved to the bucket. Buckets store $\mathcal{O}(N/(\frac{M}{B})^{\delta})$ elements. Whenever a bucket overflows, we split the bucket in a linear number of I/Os by applying an optimal external memory selection algorithm [9, 18], and insert the new bucket in the topmost levels of the tree similarly to a leaf split in a buffer tree. Performing N insertions then requires $\mathcal{O}(\delta \cdot N/B)$ I/Os, and queries require $\mathcal{O}(\delta \cdot M/B)$ I/Os plus the I/Os required to scan a bucket.

Since a bucket is formed by iteratively appending a sorted sequence of M elements (for each buffer overflow), we can adopt the ideas of fractional cascading [12, 13] to avoid scanning the complete bucket. Assume that a bucket consists of $S_1 \cup \cdots \cup S_k$, where each S_i is a sorted sequence resulting of a buffer overflow, i.e. $|S_i| = M$. Instead of only storing $S_1, \ldots, S_k$, we store slightly larger sets $\bar{S}_1, \ldots, \bar{S}_k$ where $\bar{S}_1 = S_1$ and $\bar{S}_i$ consists of S_i merged with every second element of $\bar{S}_{i-1}$, for $i \geq 2$. By induction $\bar{S}_i \leq 2M$. With every element in $\bar{S}_i$ we store a pointer to its position in $\bar{S}_{i-1}$ or its predecessor or successor in $\bar{S}_{i-1}$.

To search for an element in the bucket we first search for its predecessor or successor in $\bar{S}_k$ using $\mathcal{O}(M/B)$ I/Os. We can now use the stored pointers to find its position in $\bar{S}_{k-1}, \bar{S}_{k-2}, \ldots, \bar{S}_k$ only spending $\mathcal{O}(1)$ I/Os for each set $\bar{S}_i$. The total time for a query becomes $\mathcal{O}(\delta \cdot M/B + N/(M(M/B)^{\Omega(\delta)}))$, which for some c and $N \geq M(M/B)^{c\delta}$ is $\mathcal{O}(N/(M(M/B)^{\Omega(\delta)}))$.

Creating a new set $\bar{S}_{k+1}$ from S_{k+1} and $\bar{S}_k$ can be done by a linear scan using $\mathcal{O}(M/B)$ I/Os. Splitting a bucket requires the above described bucket structure to be recomputed. This can be performed in a linear number of I/Os by a scan over the two lists created by the selection, i.e. N insertions still perform $\mathcal{O}(\delta \cdot M/B)$ I/Os.

3.4 B-trees with buffers

By adding buffers to B-trees and varying the degree, the amortized I/O bound for updates can be improved significantly, without sacrificing the asymptotic query time of B-trees.

We consider the case where N updates are allowed to perform $\mathcal{O}(\delta \cdot \frac{N}{B})$ I/Os, for a parameter $\log N < \delta \leq B \log N$. The structure we use is a B-tree of degree $\Theta(\delta/\log N)$, i.e. at tree where leaves store $\Theta(B)$ elements and internal nodes (except the root) have degree $\Theta(\delta/\log N)$. The resulting tree has height $\mathcal{O}\left(\log_{\delta/\log N} \frac{N}{B}\right)$. Each node has a buffer that is stored in $\mathcal{O}(1)$ blocks containing $\mathcal{O}(B)$ delayed updates to the subtree below the node. Like in buffer trees, delayed updates in buffers are only propagated down whenever a buffer overflows. When the buffer of a node overflows, there exists a child such that $\Omega((B \log N)/\delta)$ elements can be moved from the buffer of the node to the buffer of the child. The remaining elements stay in the buffer. This can then imply that all buffers are overflowing along a single root-to-leaf path in the tree. When updates reach a leaf the tree is rebalanced like a buffer tree [2].

Since the topmost $\mathcal{O}(M)$ elements can be kept in internal memory, as described above for the case of B-trees, the result is a tree supporting queries with $\mathcal{O}\left(\log_{\delta/\log N} \frac{N}{M}\right)$ I/Os and N updates with $\mathcal{O}(\delta \cdot \frac{N}{B})$ I/Os. The update bound follows from that there are at most $\mathcal{O}(N/((B \log N)/\delta))$ buffer overflows at each of the at most $\mathcal{O}(\log N)$ levels of the tree, implying that there are at most $\mathcal{O}(\log N \cdot \delta N/(B \log N)) = \mathcal{O}(\delta \cdot \frac{N}{B})$ I/Os needed to handle buffer overflows (like for buffer trees, the I/Os required for rebalancing the tree will be dominated by the cost for handling buffer overflows). For $\delta \geq \log^{1+\varepsilon} N$ the query bound is $\mathcal{O}(\log_{\delta} \frac{N}{M})$, for any constant $\varepsilon > 0$.

For $\delta = B^\varepsilon \log N$, the resulting tree has degree $\mathcal{O}(B^\varepsilon)$ and height $\mathcal{O}(\frac{1}{\varepsilon} \log_B N)$. It then follows that queries require $\mathcal{O}(\frac{1}{\varepsilon} \log_B \frac{N}{M})$ I/Os and B updates require amortized $\mathcal{O}(\frac{B^\varepsilon}{\varepsilon} \log_B \frac{N}{M})$ I/Os, which matches the query time of standard B-trees within a constant factor, but with improved update bounds.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer Verlag, Berlin, 1995.

[3] L. Arge. External memory data structures. In *9th European Symposium on Algorithms (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 1–29. Springer Verlag, Berlin, 2001.

[4] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. 3rd Workshop on Algorithms and Data Structures (WADS)*, volume 709 of *Lecture Notes in Computer Science*, pages 83–94. Springer Verlag, Berlin, 1993.

[5] L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 139–160. American Mathematical Society Press, Providence, RI, 1999.

[6] L. Arge and J. Pagter. I/O-space trade-offs. In *Proc. 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 448–461. Springer Verlag, Berlin, 2000.

[7] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 346–357. ACM Press, 1999.

[8] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[9] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[10] A. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12:71–75, 1981.

[11] G. S. Brodal, S. Chaudhuri, and J. Radhakrishnan. The randomized complexity of maintaining the minimum. *Nordic Journal of Computing, Selected Papers of the 5th Scandinavian Workshop on Algorithm Theory (SWAT'96)*, 3(4):337–351, 1996.

[12] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[13] B. Chazelle and L. J. Guibas. Fractional cascading: II. applications. *Algorithmica*, 1:163–191, 1986.

[14] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[15] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison Wesley Longman, USA, 2 edition, 1998.

[16] V. Samoladas. *On Indexing Large Databases for Advanced Data Models*. PhD thesis, The University of Texas at Austin, 2001. Dept. of Computer Science.

[17] V. Samoladas and D. P. Miranker. A lower bound theorem for indexing schemes and its application to multidimensional range queries. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 44–51. ACM Press, 1998.

[18] J. F. Sibeyn. External selection. In *Proceedings of the 16th Symposium Theoretical Aspects of Computer Science (STACS)*, volume 1563 of *Lecture Notes in Computer Science*, pages 291–301. Springer-Verlag, 1999.

[19] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.