Lazy B-Trees

Casper Moldrup Rysgaard
Aarhus University, Denmark

Sebastian Wild
Philipps-University Marburg, Germany
University of Liverpool, UK

Abstract

Lazy search trees (Sandlund & Wild FOCS 2020, Sandlund & Zhang SODA 2022) are sorted dictionaries whose update and query performance smoothly interpolates between that of efficient priority queues and binary search trees – automatically, depending on actual use; no adjustments are necessary to the data structure to realize the cost savings. In this paper, we design *lazy B-trees*, a variant of lazy search trees suitable for external memory that generalizes the speedup of B-trees over binary search trees wrt. input/output operations to the same smooth interpolation regime.

A key technical difficulty to overcome is the lack of a (fully satisfactory) external variant of *biased* search trees, on which lazy search trees crucially rely. We give a construction for a subset of performance guarantees sufficient to realize external-memory lazy search trees, which we deem of independent interest.

As one special case, lazy B-trees can be used as an external-memory priority queue, in which case they are competitive with some tailor-made heaps; indeed, they offer faster decrease-key and insert operations than known data structures.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases B-tree, lazy search trees, lazy updates, external memory, deferred data structures, database cracking

Digital Object Identifier 10.4230/LIPIcs.MFCS.2025.87

Related Version *Full Version*: `https://arxiv.org/abs/2507.00277`

Funding *Casper Moldrup Rysgaard*: Independent Research Fund Denmark, grant 9131-00113B.
Sebastian Wild: EPSRC grant EP/X039447/1.

1 Introduction

We introduce the *lazy B-tree* data structure, which brings the adaptive performance guarantees of lazy search trees to external memory.

The binary search tree (BST) is a fundamental data structure, taught in every computer science degree and widespread in practical use. Wherever rank-based operations are needed, e.g., finding a kth smallest element in a dynamic set or determining the rank of an element in the set, i.e., its position in the sorted order of the current elements, augmented BSTs are the folklore solution: binary search trees using one of the known schemes to "balance" them, i.e., guarantee $\mathcal{O}(\log N)$ height for a set of size N, where we additionally store the subtree size in each node. On, say, AVL-trees, all operations of a sorted dictionary, i.e., rank, select, membership, predecessor, successor, minimum, and maximum, as well as insert, delete, change-key, split, and merge can all be supported in $\mathcal{O}(\log N)$ worst case time, where N is the current size of the set.

From the theory of efficient priority queues, it is well known that much more efficient implementations are possible if not all of the above operations have to be supported: When only minimum, insert, delete, decrease-key, and meld are allowed, all can be supported to run in constant time, except for $\mathcal{O}(\log N)$ delete. Lazy search trees [44, 46] (LSTs) show that

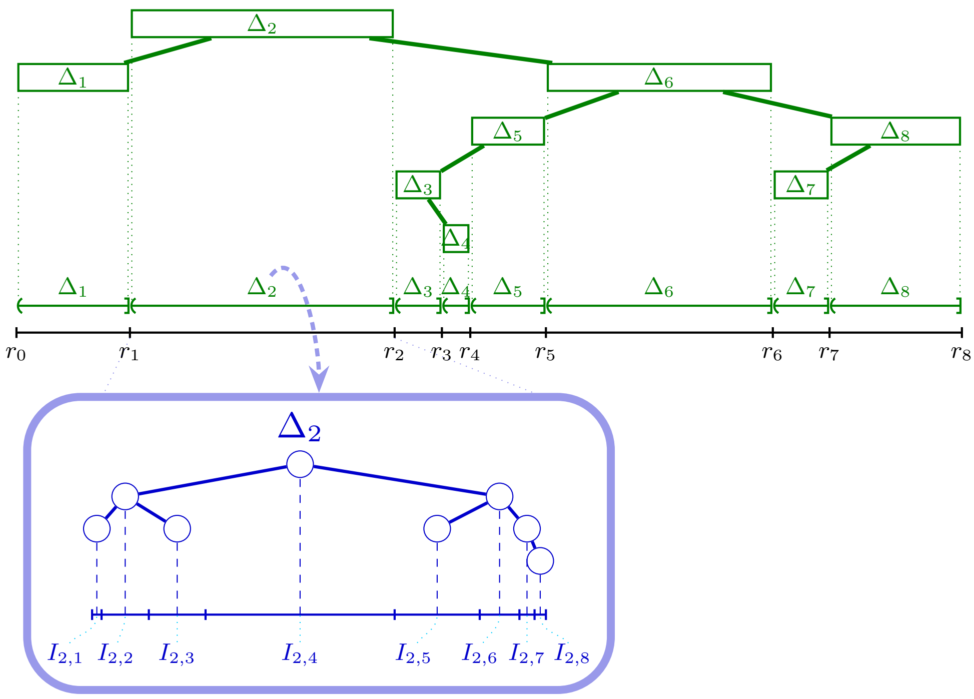

Figure 1 Schematic view of the original lazy search tree data structure [44].

we can get the same result with full support for all sorted-dictionary operations, so long as they are not used. When they do get used, lazy search trees give the optimal guarantees of any comparison-based data structure for the given query ranks, gracefully degrading from priority-queue performance to BST performance, as more and more queries are asked. They therefore serve as a versatile drop-in replacement for both priority queues and BSTs.[1]

The arguably most impactful application of sorted dictionaries across time is the database index. Here, economizing on accesses to slow secondary storage can outweigh any other costs, which gave rise to external-memory data structures; in particular B-trees and their numerous variants. They achieve a $\log_2(B)$-factor speedup over BSTs in terms of input/output operations (I/Os), for B the block size of transfers to the external memory, for scanning-dominated operations such as a range query, even a factor B speedup can be achieved.

Much ingenuity has been devoted to (a) write-optimized variants of B-trees [3, 15], often with batched queries, and (b), adaptive indices [31], or more recently learned indices [36] that try to improve query performance via adapting to data at hand. *Deferred data structuring* – known as *database cracking* in the systems community – is a special case of (b) that, like lazy search trees, refines a sorted order (of an index) successively upon queries. Yet in the past, these two approaches often remained incompatible, if not having outright conflicting goals, making either insertions or queries fast. Lazy search trees are a principled solution to get both (as detailed in Section 1.2); unfortunately, standard lazy search trees are not competitive with B-trees in terms of their I/O efficiency, thus losing their adaptive advantages when data resides on secondary storage.

[1] We point out that a related line of work on "deferred data structures" also adapts to the queried ranks in the way LSTs do, but all data structures prior to LSTs had $\Omega(\log N)$ insertion time (with the assumption that any insertion is preceded by a query for said element). These data structures are therefore inherently unable to function as an efficient priority queue. Further related work is discussed in Section 1.3.

In this paper, we present *lazy B-trees*, an I/O-efficient variant of the original lazy search trees [44].[2] Lazy search trees consist of three layers (see Figure 1). The topmost layer consists of a biased search tree of *gaps* (defined in Section 1.2), weighted by their size. The second layer consists, for each gap Δ_i, of a balanced binary search tree of $\mathcal{O}(\log|\Delta_i|)$ *intervals* $I_{i,j}$. Intervals are separated by splitter elements (pivots as in Quicksort), but are unsorted within. The third layer represents intervals as a simple unordered list.

A key obstacle to external-memory variants of lazy search trees – both for the original version [44] and for optimal lazy search trees [46] – is the topmost layer. As discussed in Section 1.3, none of the existing data structures fully solves the problem of how to design a general-purpose *external-memory biased search tree* – and we likewise leave this as an open problem for future work. However, we show that for a slightly restricted set of performance guarantees sufficient for lazy search trees, a (to our knowledge) novel construction based on partitioning elements by *weight* with doubly-exponentially increasing bucket sizes provides an I/O-efficient solution. This forms the basis of our lazy B-trees, which finally bring the adaptive versatility of lazy search trees to external memory.

1.1 The External-Memory Model

In this paper we study the sorted dictionary problem in a hierarchical-memory model, where we have an unbounded external memory and an internal memory of capacity M elements, and where data is transferred between the internal and external memory in blocks of B consecutive elements. A block transfer is called an *I/O* (input/output operation). The I/O cost of an algorithm is the number of I/Os it performs. Aggarwal and Vitter [2] introduced this as the *external-memory model* (and proved, e.g., lower bounds for sorting in the model).

1.2 Lazy Search Trees

In this section we briefly describe lazy search trees [44]. Lazy search trees support all operations of a dynamic sorted dictionary (details in [44]). We consider the following operations (sufficient to base a full implementation on):

Construct(S): Constructs the data structure from the elements of set S.[3]
Insert(e): Adds element e to the set.
Delete(ptr): Deletes the element at pointer ptr from the set.
ChangeKey(ptr, e'): Changes the element e at pointer ptr to e'.
QueryElement(e): Locates the predecessor e' to element e in the set, and returns rank r of e', and a pointer to e'.
QueryRank(r): Locates the element e with rank r in the set, and returns a pointer to e.

Lazy search trees maintain the elements of the set partitioned into G *"gaps"*, $\Delta_1, \ldots, \Delta_G$. All elements in Δ_i are weakly smaller than all elements in Δ_{i+1}, but the gaps are otherwise treated as unsorted bags. Boundaries between gaps (and new gaps) are introduced only via a query; initially we have a single gap Δ_1. Every query is associated with its *query rank* r [45, §4], i.e., the rank of its result, and splits an existing gap into two new gaps,

[2] The follow-up work [46] replaces the second and third layer using *selectable priority queues*; we discuss why these are challenging to make external in Section 1.4.

[3] Construct and iterative insertion have the same complexity in internal memory, but in external memory, the bulk operation can be supported more efficiently.

Table 1 Overview of results on Search Trees and Priority Queues for both the internal- and external-memory model. For the internal model the displayed time is the number of RAM operations performed, while for the external model the displayed time is the number of I/Os performed. Amortized times are marked by "am." and results doing batched updates and queries are marked by "batched". For priority queues, query is only for the minimum. All results use linear space.

	Insert	**Query**
Internal-Memory		
Balanced BST [1, 26]	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Priority Queue [14, 19]	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$
Lazy Search Tree [44]	$\mathcal{O}\big(\log \frac{N}{\|\Delta_i\|} + \log\log N\big)$	$\mathcal{O}(\log N + x \log c)$ am.
Optimal LST [46]	$\mathcal{O}\big(\log \frac{N}{\|\Delta_i\|}\big)$	$\mathcal{O}(\log N + x \log c)$ am.
External-Memory		
B-tree [8]	$\mathcal{O}(\log_B N)$	$\mathcal{O}(\log_B N)$
B$^\varepsilon$-tree [15]	$\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\big)$ am.	$\mathcal{O}\big(\frac{1}{\varepsilon} \log_B N\big)$ am.
Buffer Tree [3, 4]	$\mathcal{O}\big(\frac{1}{B} \log_{M/B} \frac{N}{B}\big)$ batched	$\mathcal{O}\big(\frac{1}{B} \log_{M/B} \frac{N}{B}\big)$ batched
I/O-eff. heap [37]	$\mathcal{O}\big(\frac{1}{B} \log_2 \frac{N}{B}\big)$ am.	$\mathcal{O}\big(\frac{1}{B} \log_2 \frac{N}{B}\big)$ am.
x-treap heap [28]	$\mathcal{O}\big(\frac{1}{B} \log_{M/B} \frac{N}{B}\big)$ am.	$\mathcal{O}\big(\frac{M^\varepsilon}{B} \log^2_{M/B} \frac{N}{B}\big)$ am.
This paper, LST (Theorem 1)	$\mathcal{O}\big(\log_B \frac{N}{\|\Delta_i\|} + \log_B \log_B \|\Delta_i\|\big)$	$\mathcal{O}\big(\log_B \min\{N, q\} + \frac{1}{B} \log_2 \|\Delta_i\| + \log_B \log_B \|\Delta_i\| + \frac{1}{B} x \log_2 c\big)$ am.
This paper, PQ (Corollary 2)	$\mathcal{O}(\log_B \log_B N)$	$\mathcal{O}\big(\frac{1}{B} \log_2 N + \log_B \log_B N\big)$ am.

such that its query rank is the largest rank in the left one of these gaps. After queries with ranks $r_1 < r_2 < \cdots < r_q$, we thus obtain the gaps $\Delta_1, \ldots, \Delta_{q+1}$ where $|\Delta_i| = r_i - r_{i-1}$ upon setting $r_0 = 0$ and $r_{q+1} = n$.

Lazy search trees support insertions landing in gap Δ_i in $\mathcal{O}(\log(N/|\Delta_i|) + \log\log|\Delta_i|)$ (worst-case) time and queries in $\mathcal{O}(x \log c + \log n)$ amortized time, where the query splits a gap into new gaps of sizes x and cx for some $c \geq 1$. Deletions are supported in $\mathcal{O}(\log n)$ (worst-case) time. To achieve these times, gaps are further partitioned into intervals, with an amortized splitting and merging scheme (see Section 3 for details).

1.3 Related Work

We survey key prior work in the following; for a quick overview with selected results and comparison with our new results, see also Table 1.

(External) Search trees. Balanced binary search trees (BSTs) exist in many varieties, with AVL trees [1] and red-black trees [26] the most widely known. When augmented with subtree sizes, they support all sorted dictionary operations in $\mathcal{O}(\log N)$ worst-case time. Simpler variants can achieve the same via randomization [47, 39] or amortization [48]. In external memory, B-trees [8], often in the leaf-oriented flavor as B$^+$-trees and augmented with subtree sizes, are the benchmark. They support all sorted-dictionary operations in $\mathcal{O}(\log_B N)$ I/Os. By batching queries, buffer trees [3, 4] substantially reduce the cost to amortized $\mathcal{O}\big(\frac{1}{B} \log_{M/B} \frac{N}{B}\big)$ I/Os, but are mostly useful in an offline setting due to the long delay from query to answer. B$^\varepsilon$-trees [15] avoid this with a smaller buffer of operations per node to achieve amortized $\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\big)$ I/Os for updates and $\mathcal{O}\big(\frac{1}{\varepsilon} \log_B N\big)$ I/Os for queries (with immediate answers), where $\varepsilon \in (0, 1]$ is a parameter.

Dynamic Optimality. The dynamic-optimality conjecture for Splay trees [48] resp. the GreedyBST [38, 41, 21] algorithm posits that these methods provide an instance-optimal binary-search-tree algorithm for any (long) sequence of searches over a static set of keys in a binary search tree. While still open for Splay trees and GreedyBST, the dynamic optimality of data structures has been settled in some other models: it holds for a variant of skip lists [13], multi-way branching search trees, and B-trees [13]; it has been refuted for tournament heaps [42]. As in lazy search trees, queries clustered in time or space allow a sequence to be served faster. Unlike in lazy search trees, insertions can remain expensive even when no queries ever happen close to them. A more detailed technical discussion of similarities and vital differences compared to lazy search trees appears in [44, 45].

(External) Priority Queues. When only minimum-queries are allowed, a sorted dictionary becomes a priority queue (PQ) (a.k.a. "heap"). In internal memory, all operations except delete-min can then be supported in $\mathcal{O}(1)$ amortized [25] or even worst-case time [14, 19]. In external memory, developing efficient PQs has a long history. For batch use, buffer-tree PQs [4] and the I/O-efficient heap of [37] support k operations of insert and delete-min in $\mathcal{O}\big(\frac{k}{B}\log_{M/B}\frac{N}{M}\big)$ I/Os, with N denoting the maximum number of stored keys over the k operations. The same cost per operation can be achieved in a worst-case sense [16] (i.e., B subsequent insert/delete-min operations cost $\mathcal{O}\big(\log_{M/B}\frac{N}{M}\big)$ I/Os).

None of these external-memory PQs supports decrease-key. The I/O-efficient tournament trees of [37] support a sequence of insert, delete-min, and decrease-key with $\mathcal{O}\big(\frac{1}{B}\log_2\frac{N}{B}\big)$ I/Os per operation. A further $\log\log N$ factor can be shaved off [32] (using randomization), but that, surprisingly, is optimal [23]. A different trade-off is possible: insert and decrease-key are possible in $\mathcal{O}\big(\frac{1}{B}\log_{M/B}\frac{N}{B}\big)$ amortized I/Os at the expense of driving up the cost for delete/delete-min to $\mathcal{O}\big(\frac{M^{\varepsilon}}{B}\log^2_{M/B}\frac{N}{B}\big)$ [28].

(External) Biased Search Trees. Biased search trees [9] maintain a sorted set, where each element e has a (dynamic) weight $w(e)$, such that operations in the tree spend $\mathcal{O}\big(\log\frac{W}{w(e)}\big)$ time, for W the total weight, $W = \sum_e w(e)$. Biased search trees can be built from height-balanced trees ($(2,b)$-globally-biased trees [9]) or weight-balanced trees (by representing high-weight elements several times in the tree [40]), and Splay trees automatically achieve the desired time in an amortized sense [48, 40].

None of the original designs are well suited for external memory. Feigenbaum and Tarjan [24] extended biased search trees to (a,b)-trees for that purpose. However, during the maintenance of (a,b)-trees, some internal nodes may have a degree much smaller than a (at least 2), which means that instead of requiring $\mathcal{O}(N/B)$ blocks to store N weighted elements, they require $\mathcal{O}(N)$ blocks in the worst case.[4] The authors in [24] indeed leave it as an open problem to find a space-efficient version of biased (a,b)-trees. Another attempt at an external biased search tree data structure is based on deterministic skip lists [5]. Yet again, the space usage seems to be $\Omega(N)$ blocks of memory.[5]

[4] In particular, in [24], they distinguish internal nodes between minor and major, minor being the nodes that have degree $< a$ or have a small rank. All external nodes are major.

[5] Unfortunately, it remains rather unclear from the description in the article exactly which parts of the skiplist "towers" of pointers of an element are materialized. Unlike in the unweighted case, an input could have large and small weight alternating, with half the elements of height $\approx \log_b W$. Fully materializing the towers would incur $\Omega(n\log_b W)$ space; otherwise this seems to require a sophisticated scheme to materialize towers on demand, e.g., upon insertions, and we are not aware of a solution.

To our knowledge, no data structure is known that achieves $\mathcal{O}\big(\log_B \frac{W}{w(e)}\big)$ I/Os for a dynamic weighted set in external memory while using $\mathcal{O}(N/B)$ blocks of memory.

Deferred Data Structures & Database Cracking. Deferred data structuring refers to the idea of successively establishing a query-driven structure on a dataset. This is what lazy search trees do upon queries. While the term is used more generally, it was initially proposed in [35] on the example of a (sorted) dictionary. For an offline sequence of q queries on a given (static) set of N elements, their data structure uses $\mathcal{O}(N \log q + q \log N)$ time. In [20], this is extended to allow update operations in the same time (where q now counts all operations). The refined complexity taking query *ranks* into account was originally considered for the (offline) multiple selection problem: when q ranks $r_1 < \cdots < r_q$ are sought, leaving gaps $\Delta_1, \ldots, \Delta_{q+1}$, $\Theta\big(\sum_{i=1}^{q+1} |\Delta_i| \log(N/\Delta_i)\big)$ comparisons are necessary and sufficient [22, 33]. For multiple selection in external memory, $\Theta\big(\sum_{i=1}^{q+1} \frac{|\Delta_i|}{B} \log_{M/B} \frac{N}{|\Delta_i|}\big)$ I/Os are necessary and sufficient [7, 17], even cache-obliviously [17, 18].

Closest to lazy search trees is the work on *online dynamic multiple selection* by Barbay et al. [6, 7], where online refers to the query ranks arriving one by one. As pointed out in [44], the crucial difference between all these works and lazy search trees is that the analysis of dynamic multiple selection assumes that every insertion is preceded by a query for the element, which implies that insertions must take $\Omega(\log N)$ time. (They assume a nonempty set of elements to initialize the data structure with, for which no pre-insertion queries are performed.) Barbay et al. also consider online dynamic multiple selection in external memory. By maintaining a B-tree of the pivots for partitioning, they can support updates – again, implicitly preceded by a query – at a cost of $\Theta(\log_B N)$ I/Os each.

In the context of adaptive indexing of databases, deferred data structuring is known under the name of *database cracking* [30, 29, 27]. While the focus of research is on systems engineering, e.g., on the partitioning method [43], some theoretical analyses of devised algorithms have also appeared [50, 49]. These consider the worst case for q queries on N elements similar to the original works on deferred data structures.

1.4 Contribution

Our main contribution, the lazy B-tree data structure, is summarized in Theorem 1 below.

▶ **Theorem 1** (Lazy B-Trees). *There exists a data structure over an ordered set, that supports*

- *Construct(S) in worst-case $\mathcal{O}(|S|/B)$ I/Os,*
- *Insert in worst-case $\mathcal{O}\big(\log_B \frac{N}{|\Delta_i|} + \log_B \log_B |\Delta_i|\big)$ I/Os,*
- *Delete in amortized $\mathcal{O}\big(\log_B \frac{N}{|\Delta_i|} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|\big)$ I/Os,*
- *ChangeKey in worst-case $\mathcal{O}\big(\log_B \log_B |\Delta_i|\big)$ I/Os if the element is moved towards the nearest queried element but not past it, and amortized $\mathcal{O}\big(\frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i|\big)$ I/Os otherwise, and*
- *QueryElement and QueryRank in amortized $\mathcal{O}\big(\log_B \min\{N, q\} + \frac{1}{B} \log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B} x \log_2 c\big)$ I/Os.*

Here N denotes the size of the current set, $|\Delta_i|$ denotes the size of the manipulated gap, q denotes the number of performed queries, and x and cx for $c \geq 1$ are the resulting sizes of the two gaps produced by a query. The space usage is $\mathcal{O}(N/B)$ blocks.

From the above theorem, the following corollary can be derived, which states the performance of lazy B-trees when used as a priority queue.

▶ **Corollary 2** (Lazy B-Trees as external PQ). *A lazy B-tree may be used as a priority queue, to support, within $\mathcal{O}(N/B)$ blocks of space, the operations*

- *Construct(S) in worst-case $\mathcal{O}(|S|/B)$ I/Os,*
- *Insert in worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os,*
- *Delete in amortized $\mathcal{O}\big(\frac{1}{B}\log_2 N + \log_B \log_B N\big)$ I/Os,*
- *DecreaseKey in worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os, and*
- *Minimum in amortized $\mathcal{O}\big(\frac{1}{B}\log_2 N + \log_B \log_B N\big)$ I/Os.*

The running times of lazy B-trees when used as a priority queue are not competitive with heaps targeting sorting complexity (such as buffer trees [3, 4]); however these data structures do not (and cannot [23]) support decrease-key efficiently. By contrast, for very large N, lazy B-trees offer exponentially faster decrease-key and insert than previously known external priority queues, while only moderately slowing down delete-min queries.

Our key technical contribution is a novel technique for partially supporting external biased search tree performance, formally stated in Theorem 3 below. In the language of a biased search tree, it supports searches (by value or rank) as well as incrementing or decrementing[6] a weight $w(e)$ by 1 in $\mathcal{O}(\log_B(W/w(e)))$ I/Os for an element e or weight $w(e)$; inserting or deleting an element, however, takes $\mathcal{O}(\log_B N)$ I/Os irrespective of weight, where N is the number of elements currently stored. Unlike previous approaches, the space usage is the $\mathcal{O}(N/B)$ blocks throughout. A second technical contribution is the streamlined potential-function-based analysis of the interval data structure of lazy search trees.

We mention two, as yet insurmountable, shortcomings of lazy B-trees. The first one is the $\log\log N$ term we inherit from the original Lazy search trees [44]. This cost term is in addition to the multiple-selection lower bound and thus not necessary. Indeed, it was in internal memory subsequently removed [46], using an entirely different representation of gaps, which fundamentally relies on soft-heap-based selection on a priority queue [34]. The route to an external memory version of this construction is currently obstructed by two road blocks. First, we need an external-memory soft heap; the only known result in this direction [11] only gives performance guarantees when $N = \mathcal{O}\big(B(M/B)^{M/2(B+\sqrt{M/B})}\big)$ and hence seems not to represent a solution for the general problem. Second, the selection algorithm from [46] requires further properties of the priority queue implementation, in particular a bound on the fanout; it is not clear how to combine this with known external priority queues.

The second shortcoming is that – unlike for comparisons – we do not get close to the I/O-lower bound for multiple-selection with lazy B-trees. Doing so seems to require a way of buffering as in buffer trees, to replace our fanout of B by a fanout of M/B. This again seems hard to achieve since an *insertion* in the lazy search tree uses a *query* on the gap data structure, and a *query* on the lazy search tree entails an *insertion* into the gap data structure (plus a re-weighting operation).

Outline. The remainder of the paper is structured as follows. Section 2 describes the gap data structure (our partial external biased search tree) and key innovation. Section 3 sketches the changes needed to turn the interval data structure from [44] into an I/O-efficient data structure; the full details, including our streamlined potential function, appear in Appendix A of the arXiv version of this paper. In Section 4, we then show how to assemble the pieces into a proof of our main result, Theorem 1. We conclude in Section 5 with some open problems.

[6] General weight changes are possible in that time with $w(e)$ the minimum of the old and new weight.

2 The Gap Structure: A New Restricted External Biased Search Tree

In this section we present a structure on the gaps, which allows for the following operations. Let N denote the total number of elements over all gaps and G denote the number of gaps. Note that $G \leq N$, as no gaps are empty. We let Δ_i denote the ith gap when sorted by element, and $|\Delta_i|$ denote the number of elements contained in gap Δ_i.

GapByElement(e) Locates the gap Δ_i containing element e.
GapByRank(r) Locates the gap Δ_i containing the element of rank r.
GapIncrement(Δ_i) / **GapDecrement**(Δ_i) Changes the weight of gap Δ_i by 1.
GapSplit$(\Delta_i, \Delta'_i, \Delta'_{i+1})$ Splits gap Δ_i into the two non-overlapping gaps Δ'_i and Δ'_{i+1}, s.t. the elements of Δ_i are equal to the elements of Δ'_i and Δ'_{i+1}.

For the operations, we obtain I/O costs, as described in the theorem below.

▶ **Theorem 3** (Gap Data Structure). *There exists a data structure on a weighted ordered set, that supports* GapByElement*,* GapByRank*,* GapIncrement *and* GapDecrement *in $\mathcal{O}\left(\log_B \frac{N}{|\Delta_i|}\right)$ I/Os, and* GapSplit *in $\mathcal{O}(\log_B G)$ I/Os. Here N denotes the total size of all gaps, $|\Delta_i|$ denotes the size of the touched gap and G denotes the total number of gaps. The space usage is $\mathcal{O}(G/B)$ blocks.*

▶ Remark 4 (Comparison with biased search trees). A possible solution would be an external-memory version of biased search trees, but as discussed in the introduction, no fully satisfactory such data structure is known. Instead of supporting all operations of biased trees in full generality, we here opt for a solution solving (just) the case at hand. The solution falls short of a general biased search trees, as the insertion or deletion costs are not a function of the *weight* of the affected gap, but the *total number G* of gaps in the structure. Moreover, we only describe how to change the weight of gaps by 1; however, general weight changes could be supported, with the size of the change entering the I/O cost, matching what we expect from a general biased search tree.

Note that the gaps in the structure are non-overlapping, and that their union covers the whole element range. The query for an element contained in some gap is therefore a predecessor query on the left side of the gaps, however, as their union covers the entire element range, the queries on the gap structure behave like *exact queries*: we can detect whether we have found the correct gap and can then terminate the search. (By definition, there are no unsuccessful searches in our scenario, either.)

For consistency with the notation of biased search trees, we write in the following $w_i = |\Delta_i|$ and $W = \sum_i w_i$. Note that $W = N$. Consider a conceptual list, where the gaps are sorted decreasingly by *weight*, and let gap Δ_i be located at some index ℓ in this list. This conceptual list is illustrated in Figure 2, and Figure 3 gives a two-dimensional view of the list, which considers both the gap weight as well as the gaps ordered by element. As the total weight before index ℓ in the conceptual list is at most W and the weight of each gap before index ℓ is at least w_i, then it must hold that $\ell \leq \frac{W}{w_i}$. If we were to search for a gap of known *weight*, it can therefore be found with an exponential search [10] in time $\mathcal{O}(\log \ell) = \mathcal{O}\left(\log \frac{W}{w_i}\right)$. However, searches are based on *element values* (and, e.g., for insertions, without knowing the target gap's weight), so this alone would not work.

2.1 Buckets by Weight and Searching by Element

Instead, the gaps are split into buckets b_j, s.t. the weight of all gaps in bucket b_j is greater than or equal to the weight of all gaps in bucket b_{j+1}. We further impose the invariant that the size of (number of gaps in) bucket b_j is *exactly* B^{2^j} for all but the last bucket,

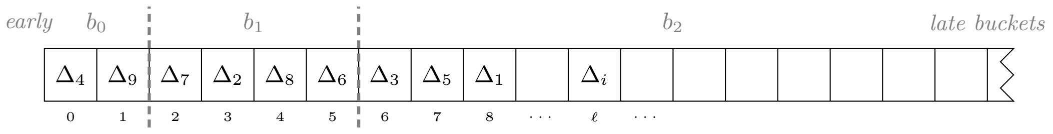
early
b_0
b_1
b_2
late buckets
Δ_4 Δ_9 Δ_7 Δ_2 Δ_8 Δ_6 Δ_3 Δ_5 Δ_1 Δ_i
0 1 2 3 4 5 6 7 8 ⋯ ℓ ⋯

Figure 2 The conceptual list of the gaps. The gaps are sorted by decreasing weight, with the heaviest gap (largest weight), Δ_4 at index 0. Gap Δ_i with weight w_i is stored at index $\ell \leq W/w_i$ in the list. The gaps are split into buckets $b_0, b_1, b_2, \ldots$ of doubly exponential size.

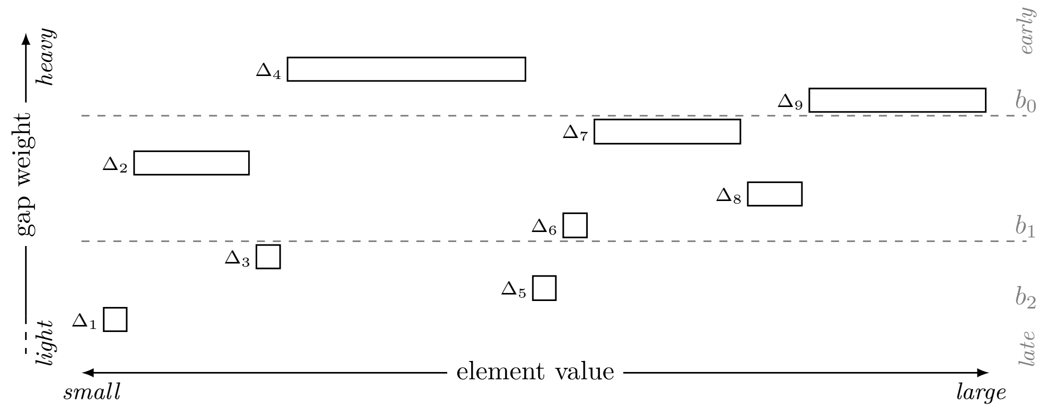

Figure 3 Two dimensional view of the first gaps of Figure 2 represented by rectangles, with the width of the rectangle denoting the weight (size) of the gap. The horizontal axis sorts gaps by element value; the vertical axis by weight. Dashed lines show bucket boundaries.

which may be of any (smaller) size. Each bucket is a B-tree containing the gaps in the bucket, sorted by element value. The time to search for a given element in bucket b_j is therefore $\mathcal{O}(\log_B(|b_j|)) = \mathcal{O}(2^j)$ I/Os. For consistent wording, we will in the following always use *smaller/larger* to refer to the order by element values, *lighter/heavier* for gaps of smaller/larger weight/size, and *earlier/later* for buckets of smaller/larger index. Note that earlier buckets contain fewer, but heavier gaps.

GapByRank. A search for a gap proceeds by searching in buckets $b_0, b_1, b_2, \ldots$ until the desired gap is found in some bucket b_k. To search in all buckets up until b_k requires $\sum_{j=0}^{k} \mathcal{O}(2^j) = \mathcal{O}(2^k)$ I/Os, which is therefore up to constant factors the same cost as searching in only bucket b_k. Consider some gap Δ_i, which has the ℓth heaviest weight, i.e., it is located at index ℓ, when sorting all gaps decreasingly by weight. By the invariant, the buckets are sorted decreasingly by the weight of their internal elements. Let the bucket containing Δ_i be b_k. It must then hold that the sizes of the earlier buckets does not allow for Δ_i to be included, but that bucket b_k does. Therefore,

$$\sum_{j=0}^{k-1} |b_j| \;<\; \ell \;\leq\; \sum_{j=0}^{k} |b_j| \;.$$

As $|b_j| = B^{2^j}$, the sums are asymptotically equal to the last term of the sum up to constant factors (arXiv version, Lemma 10). It then holds that $\ell = \mathcal{O}\big(B^{2^k}\big)$ and $\ell = \Omega\big(B^{2^{k-1}}\big) = \Omega\big((B^{2^k})^{1/2}\big)$. Thus $\log_B(\ell) = \Theta(\log_B |b_k|)$, and conversely $2^k = \Theta(\log_B \ell)$. The gap Δ_i can thus be found using $\mathcal{O}\big(\log_B \frac{W}{w_i}\big)$ I/Os, concluding the GapByElement operation.

2.2 Updating the Weights

Both explicit weight changes in GapIncrement/GapDecrement as well as the GapSplit operation require changes to the weights of gaps. Here, we have to maintain the sorting of buckets by weight.

GapIncrement/GapDecrement. When performing GapIncrement on gap Δ_i, the weight is increased by 1. In the conceptual list, this may result in the gap moving to an earlier index, with the order of the other gaps the same. This move is made in the conceptual list (Figure 2) by swapping Δ_i with its neighbor. As the buckets are ordered by element value, swapping with a neighbor (by weight) in the same bucket does not change the bucket structure. When swapping with a neighbor in an earlier bucket, however, the bucket structure must change to reflect this. Following the conceptual list (Figure 2), this gap is a lightest (minimum-weight) gap in bucket b_{k-1}. This may then result in Δ_i moving out of its current bucket b_k, into some earlier bucket. As the gap moves to an earlier bucket, some other gap must take its place in bucket b_k. Following the conceptual list, it holds that this gap is a lightest (minimum-weight) gap in bucket b_{k-1}, and so on for the remaining bucket moves.

When performing GapDecrement, the gap may move to a later bucket, where the heaviest gap in bucket b_{k+1} is moved to bucket b_k.

In both cases, a lightest gap in one bucket is swapped with a heaviest gap in the neighboring (later) bucket. To find lightest or heaviest gaps in a bucket, we augment the nodes of the B-tree of each bucket with the values of the lightest and heaviest weights in the subtree for each of its children. This allows for finding a lightest or heaviest weight gap in bucket b_j efficiently, using $\mathcal{O}(\log_B |b_j|)$ I/Os. Further, this augmentation can be maintained under insertions and deletions.

Upon a GapIncrement or GapDecrement operation, the desired gap Δ_i can be located in the structure using $\mathcal{O}\big(\log_B \frac{W}{w_i}\big)$ I/Os, and the weight is adjusted. We must then perform swaps between the buckets, until the invariant that buckets are sorted decreasingly by weight holds. This swapping may be performed on all buckets up to the last touched bucket. For GapIncrement the swapping only applies to earlier buckets as the weight of Δ_i increases, and the height of the tree in the latest bucket is $\mathcal{O}\big(\log_B \frac{W}{w_i}\big)$. For GapDecrement the swapping is performed into later buckets, but only up to the final landing place for weight $w_i - 1$. If $w_i \geq 2$, the height of the tree in the last touched bucket is $\mathcal{O}\big(\log_B \frac{W}{w_i-1}\big) = \mathcal{O}\big(1 + \log_B \frac{W}{w_i}\big)$. If $w_i = 1$, gap Δ_i is to be deleted. The gap is swapped until it is located in the last bucket, from where it is removed. We have $\mathcal{O}\big(\log_B \frac{W}{w_i}\big) = \mathcal{O}\big(\log_B W\big)$, as $w_i = 1$, whereas the height of the last bucket is $\mathcal{O}(\log_B G) = \mathcal{O}(\log_B W)$, as $G \leq W$. Therefore both GapIncrement and GapDecrement can be performed using $\mathcal{O}\big(\log_B \frac{W}{w_i}\big)$ I/Os.

GapSplit. When GapSplit is performed on gap Δ_i, the gap is removed and replaced by two new gaps Δ'_i and Δ'_{i+1}, s.t. $|\Delta_i| = |\Delta'_i| + |\Delta'_{i+1}|$. In the conceptual list, the new gaps must reside at later indexes than Δ_i. As the sizes of the buckets are fixed, and the number of gaps increases, some gap must move to the last bucket. Similarly to GapIncrement and GapDecrement, the order can be maintained by performing swaps of gaps between consecutive buckets: First, we replace Δ_i (in its spot) by Δ'_i; if this violates the ordering of weights between buckets, we swap Δ'_i with the later neighboring bucket, until the correct bucket is reached. Then we insert Δ'_{i+1} into the last bucket and swap it with its earlier neighboring bucket until it, too, has reached its bucket. Both processes touch at most all $\log_B \log_2 G$ buckets and spend a total of $\mathcal{O}(\log_B G)$ I/Os.

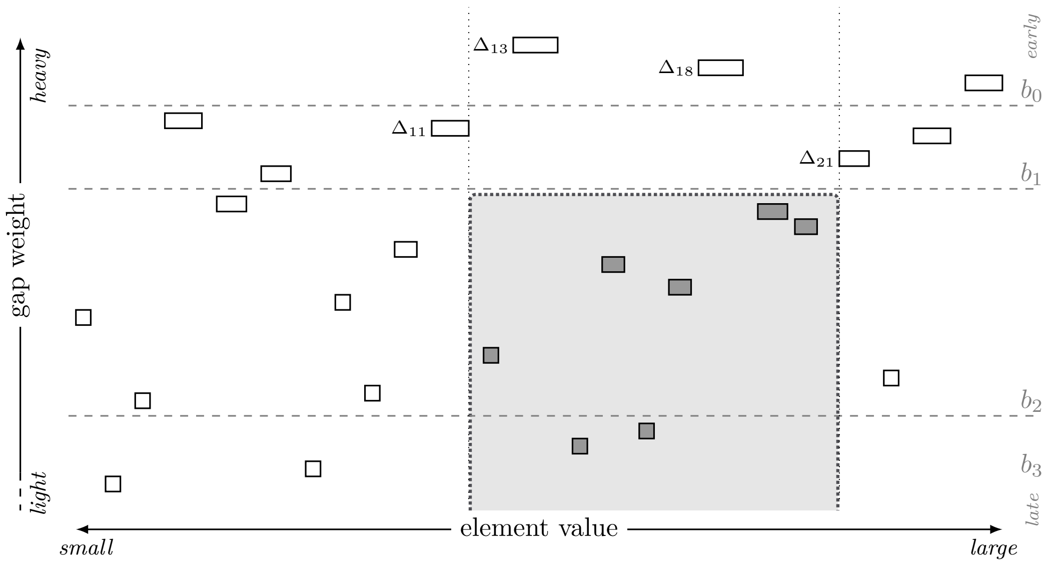

Figure 4 Illustration of "holes". The figure shows the two-dimensional view of gaps described in Figure 3 (gap widths not to scale). The *hole* between Δ_{11} and Δ_{21} in bucket b_1, marked by the dotted line, contains the total weight of all gaps marked in gray. These are precisely the gaps that fall both (a) between Δ_{11} and Δ_{21} by element value and (b) in later buckets by (lighter) weight. Note specifically that Δ_{13} and Δ_{18} are *not* counted in the hole, as they reside in an earlier bucket.

2.3 Supporting Rank Queries

The final operation to support is GapByRank. Let the *global rank* of a gap Δ_i, $r(\Delta_i) = |\Delta_1| + \cdots + |\Delta_{i-1}|$, denote the rank of the smallest element located in the gap, i.e., the number of elements residing in gaps of smaller elements. The *local rank* of a gap Δ_i in bucket b_j, $r_j(\Delta_i)$, denotes the rank of the gap among elements located in bucket b_j only.

Computing the rank of a gap. To determine the global rank of a gap Δ_i, the total weight of all smaller gaps must be computed. This is equivalent to the sum of the local ranks of Δ_i over all buckets: $r(\Delta_i) = \sum_j r_j(\Delta_i)$. First we augment the B-trees of the buckets, such that each node contains the total weight of all gaps in the subtree. This allows us to compute the local rank $r_j(\Delta_i)$ of a gap Δ_i inside a bucket b_j using $\mathcal{O}(\log_B |b_j|)$ I/Os. The augmented values can be maintained under insertions and deletions in the tree within the stated I/O cost. When searching for the gap Δ_i, the local rank $r_j(\Delta_i)$ of all earlier buckets b_j up to the bucket b_k, which contains Δ_i, can then be computed using $\mathcal{O}\big(\log_B \frac{W}{w_i}\big)$ I/Os in total.

It then remains to compute the total size of gaps smaller than Δ_i in all buckets after b_k, i.e., the sum of the local ranks $r_\ell(\Delta_i)$ for all later buckets b_ℓ, $\ell > k$, to compute in total the global rank $r(\Delta_i)$. As these buckets are far bigger, we cannot afford to query them. Note that any gaps in these later buckets must fall between two *consecutive* gaps in b_k, as the gaps are non overlapping in element-value space. Denote the space between two gaps in a bucket as a *hole*. We then further augment the B-tree to contain in each hole of bucket b_j the total size of gaps of *later* buckets contained in that hole, and let each node contain the total size of all holes in the subtree. See Figure 4 for an illustration of which gaps contributes to a hole.

This then allows computing the global rank $r(\Delta_i)$ of gap Δ_i, by first computing the local rank $r_j(\Delta_i)$ in all earlier buckets b_j, i.e., for all $j \le k$, and then adding the total size of all smaller holes in b_k. The smaller holes in b_k exactly sum to the local ranks $r_\ell(\Delta_i)$ for all later buckets b_ℓ for $\ell > k$. As this in total computes $\sum_j r_j(\Delta_i)$, then by definition, we obtain the global rank $r(\Delta_i)$ of the gap Δ_i.

GapByRank. For a GapByRank operation, some rank r is given, and the gap Δ_i is to be found, s.t. Δ_i contains the element of rank r, i.e., the global rank $r(\Delta_i) \leq r < r(\Delta_{i+1})$. The procedure is as follows. First, the location of the queried rank r is computed in the first bucket. If this location is contained in a gap Δ_i of the bucket, then Δ_i must be the correct gap, which is then returned. Otherwise, the location of r must be in a hole. As the sizes of the gaps contained in the first bucket is not reflected in the augmentation of later buckets, the value of r is updated to reflect the missing gaps of the first bucket: we subtract from t the local rank $r_0(\Delta_i)$ of the gap Δ_i immediately after the hole containing the queried rank r. This adjusts the queried rank r to be relative to elements in gaps of later buckets. Put differently, the initial queried rank r is the queried global rank, which is the sum local ranks; we now remove the first local rank again. This step is recursively applied to later buckets, until the correct gap Δ_i is found in some bucket b_k, which contains the element of the initially queried global rank r. As the correct gap Δ_i must be found in the bucket b_k containing it, the GapByRank operation uses $\mathcal{O}\big(\log_B \frac{W}{w_i}\big)$ I/Os to locate it.

Maintaining hole sizes. The augmentation storing the sizes of all holes in a subtree can be updated efficiently upon insertions or deletions of gaps in the tree, or upon updating the size of a hole. However, computing the sizes of the holes upon updates in the tree is not trivial. If a gap is removed from a bucket, the holes to the left resp. right of that gap are merged into a single hole, with a size equal to the sum of the previous holes. If the removed gap is moved to a later bucket, the hole must now also include the size of the removed gap. A newly inserted gap must, by the non-overlapping nature of gaps, land within a hole of the bucket, which is now split in two. If the global rank of the inserted gap is known, then the sizes of the resulting two holes can be computed, s.t. the global rank is preserved.

In the operations on the gap structure, GapByElement or GapByRank do not change the sizes of gaps, so the augmentation does not change. Upon a GapIncrement or GapDecrement operation, the size of some gap Δ_i changes. This change in size can be applied to all holes containing Δ_i in earlier buckets using $\mathcal{O}\big(\log_B \frac{W}{w_i}\big)$ I/Os in total. Then swaps are performed between neighboring buckets until the invariant that buckets are sorted by weight, holds again. During these swaps, the sizes of gaps do not change, which allows for computing the global rank of the gaps being moved, and update the augmentation without any overhead in the asymptotic number of I/Os performed. The total number of I/Os to perform a GapIncrement or GapDecrement operation does not increase asymptotically, and therefore remains $\mathcal{O}\big(\log_B \frac{W}{w_i}\big)$.

When a GapSplit is performed, a single gap Δ_i is split. As the elements in the two new gaps Δ'_i and Δ'_{i+1} remain the same as those of the old gap Δ_i, there cannot be another gap between them. The value in all smaller holes therefore remains correct. Moving Δ'_{i+1} to the last bucket then only needs updating the value of the holes of the intermediate buckets, which touches at most all $\log_B \log_2 G$ buckets spending $\mathcal{O}(\log_B G)$ I/Os in total. Swaps are then performed, where updating the augmentation does not change the asymptotic number of I/Os performed.

Space Usage. To bound the space usage, note that the augmentation of the B-trees at most increase the node size by a constant factor. Since the space usage of a B-tree is linear in the number of stored elements, and since each gap is contained in a single bucket, the space usage is $\mathcal{O}(G/B)$ blocks in total. This concludes the proof of Theorem 3.

3 The Interval Structure

A gap Δ_i contains all elements in the range the gap covers. Note that by Theorem 1, a query on the gap for an element must be *faster* for elements *closer* to the border of a gap; information-theoretically speaking, such queries reveal less information about the data. It is therefore not sufficient to store the elements in the gap in a single list.

External memory interval data structure. We follow the construction of Sandlund and Wild [44], in their design of the interval structure. In this section we present the construction, and argue on the number of I/Os this construction uses. The main difference from the original construction is in the speed-up provided by the external-memory model; namely that scanning a list is faster by a factor B, and that B-trees allows for more efficient searching. These differences are also what allows for slightly improving the overall I/O cost of the original construction, when moving it to the external-model. Due to space constraints, the full analysis can be found in Appendix A of the arXiv version of this paper.

We allow for the following operations:

IntervalsInsert(e): Inserts element e into the structure.
IntervalsDelete(ptr): Deletes the element e at pointer ptr from the structure.
IntervalsChange(ptr, e'): Changes the element e at pointer ptr to the element e'.
IntervalSplit(e) or IntervalSplit(r): Splits the set of intervals into two sets of intervals at element e or rank r.

A gap has a *"sidedness"*, which denotes the number of sides the gap has had a query. Denote a side as a *queried side*, if that rank has been queried before (cf. [44]). If there have been no queries yet, the (single) gap is a 0-sided gap. When a query in a 0-sided gap occurs, two new gaps are created which are both 1-sided. Note that "1-sided" does not specify which side was queried – left or right. When queries have touched both sides, the gap is 2-sided.

We obtain the following I/O costs for the above operations.

▶ **Theorem 5** (Interval Data Structure). *There exists a data structure on an ordered set, maintaining a set of intervals, supporting*

- *IntervalsInsert in worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os,*
- *IntervalsDelete in amortized $\mathcal{O}\big(\frac{1}{B}\log_2 |\Delta_i| + \log_B \log_B |\Delta_i|\big)$ I/Os,*
- *IntervalsChange in worst-case $\mathcal{O}(\log_B \log_B |\Delta_i|)$ I/Os, if the element is moved towards the nearest queried side or amortized $\mathcal{O}\big(\frac{1}{B}\log_2 |\Delta_i| + \log_B \log_B |\Delta_i|\big)$ I/Os otherwise, and*
- *IntervalSplit in amortized $\mathcal{O}\big(\frac{1}{B}\log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B}x\log_2 c\big)$ I/Os.*

Here $|\Delta_i|$ denotes the number of elements contained in all intervals, and x and cx for $c \geq 1$ are the resulting sizes of the two sets created by a split. The space usage is $\mathcal{O}(|\Delta_i|/B)$ blocks.

Intervals in external memory. Let the gap Δ_i contain multiple non-overlapping intervals $I_{i,j}$, which contain the elements located in the gap. The elements of the intervals are sorted between intervals, but not within an interval. Intervals therefore span a range of elements with known endpoints. Each such interval is a blocked-linked-list containing the elements of the interval. Additionally, we store an augmented B-tree over the intervals, allowing for efficiently locating the interval containing a given element (using $\mathcal{O}(\log_B(\#\text{intervals}))$ I/Os). The B-tree is augmented to hold in each node the total sizes of all intervals in the subtrees of the node, which allows for efficient rank queries.

By packing all intervals into a single blocked-linked-list, and noting that there can be no more intervals than there are elements, the space usage of the intervals and the augmented B-tree over the intervals is $\mathcal{O}(|\Delta_i|/B)$ blocks.

Intuition of interval maintenance of [44]. Intuitively, there is a trade-off in maintaining intervals: having many small intervals reduces future query costs since these are typically dominated by the linear cost of splitting one interval; but it increases the time to search for the correct interval upon insertions (and other operations). Lazy search trees handle this trade-off as follows. To bound the worst case insertion costs, we enforce a hard limit of $\mathcal{O}(\log_2|\Delta_i|)$ on the number of intervals in a gap Δ_i, implemented via a merging rule. To amortize occasional high costs for queries, we accrue potential for any intervals that have grown "too large" relative to their proximity to a gap boundary.

Given the logarithmic number of intervals, the best case for queries would be to have interval sizes grow exponentially towards the middle. This makes processing of intervals close to the boundary (i.e., close to previous queries) cheap; for intervals close to the middle, we can afford query costs linear in Δ_i. It can be shown that for exponentially growing intervals, the increase of the lower bound from any query allows us to precisely pay for the incurred splitting cost. However, the folklore rule of having each interval, say, 2–4 times bigger than the previous interval seems too rigid to maintain. Upon queries, it triggers too many changes.

Merging & Potential. We therefore allow intervals to grow bigger than they are supposed to be, but charge them for doing so in the potential. By enforcing the merging rule when the number of elements in the gap decreases, we maintain the invariant that the number of intervals is $\mathcal{O}(\log_2|\Delta_i|)$, which allows for locating an interval using $\mathcal{O}(\log_B\log_2|\Delta_i|) = \mathcal{O}(\log_B\log_B|\Delta_i|)$ I/Os. Enforcing the merging rule is achieved by scanning the intervals using the B-tree, and it can be shown that this operation uses amortized $\mathcal{O}\big(\frac{1}{B}\log_2|\Delta_i|\big)$ I/Os. Merging intervals takes $\mathcal{O}(1)$ I/O, but by adding extra potential to intervals, this step of the merge can be amortized away.

Upon this, we can show that IntervalsInsert uses $\mathcal{O}(\log_B\log_B|\Delta_i|)$ I/Os, and that IntervalsDelete uses amortized $\mathcal{O}\big(\frac{1}{B}\log_2|\Delta_i| + \log_B\log_B|\Delta_i|\big)$ I/Os. When performing IntervalsChange, moving the element from one interval to another uses $\mathcal{O}(\log_B\log_B|\Delta_i|)$ I/Os. However, the potential function causes an increase of $\mathcal{O}\big(\frac{1}{B}\log_2|\Delta_i|\big)$ I/Os in the amortized cost when an element is moved away from the closest queried side of the gap.

When a query occurs, an interval $I_{i,j}$ is split around some element or rank. The interval can be located using the B-tree over intervals on either element or rank in $\mathcal{O}(\log_B\log_B|\Delta_i|)$ I/Os. For splitting around an element, the interval is scanned and partitioned using $\mathcal{O}(|I_{i,j}|/B)$ I/Os. For splitting around a rank, deterministic selection [12] is applied, which uses $\mathcal{O}(|I_{i,j}|/B)$ I/Os (see, e.g., [17, 18]). In both cases the number of I/Os grows with the interval size. Analyzing the change in the potential function upon this split, we can show that IntervalSplit uses amortized $\mathcal{O}\big(\frac{1}{B}\log_2|\Delta_i| + \log_B\log_B|\Delta_i| + \frac{1}{B}x\log_2 c\big)$ I/Os.

This (with Appendix A of the arXiv version) concludes the proof of Theorem 5.

4 Lazy B-Trees

In this section, we combine the gap structure of Section 2 and the interval structure of Section 3, to achieve an external-memory lazy search-tree. Recall our goal, Theorem 1.

▶ **Theorem 1** (Lazy B-Trees). *There exists a data structure over an ordered set, that supports*

- *Construct(S) in worst-case $\mathcal{O}(|S|/B)$ I/Os,*
- *Insert in worst-case $\mathcal{O}\big(\log_B\frac{N}{|\Delta_i|} + \log_B\log_B|\Delta_i|\big)$ I/Os,*

- *Delete in amortized $\mathcal{O}\big(\log_B \frac{N}{|\Delta_i|} + \frac{1}{B}\log_2 |\Delta_i| + \log_B \log_B |\Delta_i|\big)$ I/Os,*
- *ChangeKey in worst-case $\mathcal{O}\big(\log_B \log_B |\Delta_i|\big)$ I/Os if the element is moved towards the nearest queried element but not past it, and amortized $\mathcal{O}\big(\frac{1}{B}\log_2 |\Delta_i| + \log_B \log_B |\Delta_i|\big)$ I/Os otherwise, and*
- *QueryElement and QueryRank in amortized $\mathcal{O}\big(\log_B \min\{N, q\} + \frac{1}{B}\log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B}x \log_2 c\big)$ I/Os.*

Here N denotes the size of the current set, $|\Delta_i|$ denotes the size of the manipulated gap, q denotes the number of performed queries, and x and cx for $c \geq 1$ are the resulting sizes of the two gaps produced by a query. The space usage is $\mathcal{O}(N/B)$ blocks.

We use the I/O bounds shown in Theorems 3 and 5 to bound the cost of the combined operations. These operations are performed as follows.

A Construct(S) operation is performed by creating a single gap over all elements in S, and in the gap create a single interval with all elements. This can be done by a single scan of S, and assuming that S is represented compactly/contiguously, this uses $\mathcal{O}(|S|/B)$ I/Os.

To perform an Insert(e) operation, GapByElement is performed to find the gap Δ_i containing element e. Next, IntervalsInsert is performed to insert e into the interval structure of gap Δ_i. Finally, GapIncrement is performed on Δ_i, as the size has increased. In total this uses worst-case $\mathcal{O}\big(\log_B \frac{N}{|\Delta_i|} + \log_B \log_B |\Delta_i|\big)$ I/Os.

Similarly, upon a Delete(ptr) operation, IntervalsDelete is performed using ptr, to remove the element e at the pointer location. Next GapDecrement is performed on the gap Δ_i. In total this uses amortized $\mathcal{O}\big(\log_B \frac{N}{|\Delta_i|} + \frac{1}{B}\log_2 |\Delta_i| + \log_B \log_B |\Delta_i|\big)$ I/Os.

A ChangeKey operation may only change an element, s.t. it remains within the same gap (otherwise we need to use Delete and Insert). This operation therefore does not need to perform operations on the gap structure, but only on the interval structure of the relevant gap. The operation is therefore performed directly using the IntervalsChange operation.

The QueryElement and QueryRank operations are performed in similar fashions. First, GapByElement or GapByRank is performed to find the relevant gap Δ_i containing the queried element. Next IntervalSplit is performed on the interval structure, which yields two new gaps Δ'_i and Δ'_{i+1}, which is updated into the gap structure using GapSplit. Note that the number of gaps is bounded by the number of elements N, but also by the number of queries q performed, as only queries introduce new gaps. In total, the QueryElement and QueryRank operations uses amortized $\mathcal{O}\big(\log_B \min\{N, q\} + \frac{1}{B}\log_2 |\Delta_i| + \log_B \log_B |\Delta_i| + \frac{1}{B}x \log_2 c\big)$ I/Os.

The space usage of the gap structure is $\mathcal{O}(G/B) = \mathcal{O}(N/B)$ blocks. For gap Δ_i, the space usage of the interval structure is $\mathcal{O}(|\Delta_i|/B)$ blocks. If $|\Delta_j| = \mathcal{O}(B)$, for some Δ_j, we cannot simply sum over all the substructures. By instead representing all such "lightweight" gaps in a single blocked-linked-list, we obtain that the space usage over the combined structure is $\mathcal{O}(N/B)$ blocks. The gaps are located in the combined list using the gap structure, and updates or queries to the elements of the gap may then be performed using $\mathcal{O}(1)$ I/Os, allowing the stated time bounds to hold. Similarly, an interval structure may be created for a gap, when it contains $\Omega(B)$ elements using $\mathcal{O}(1)$ I/Os. The stated time bounds therefore still applies to the lightweight gaps.

This concludes the proof of Theorem 1.

Priority Queue Case. When a lazy B-tree is used a priority queue, the queries are only performed on the element of smallest rank, and this element is deleted before a new query is performed. If this behavior is maintained, we first note that the structure only ever has a single 1-sided gap of size N, with the queried side to the left. This allows for Insert to be performed in worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os, as the search time in the gap structure reduces to $\mathcal{O}(1)$. Similarly, Delete is performed using amortized $\mathcal{O}\big(\frac{1}{B}\log_2 N + \log_B \log_B N\big)$ I/Os. To perform the DecreaseKey operation, a ChangeKey operation is performed. As the queried side is to the left, this must move the element towards the closest queried side, leading to a DecreaseKey operation being performed using worst-case $\mathcal{O}(\log_B \log_B N)$ I/Os. Finally, Minimum is performed as a QueryRank(r) with $r = 1$. As this must split the gap at $x = 1$ and $cx = N - 1$, the operation is performed using amortized $\mathcal{O}\big(\frac{1}{B}\log_2 N + \log_B \log_B N\big)$ I/Os. This concludes the proof of Corollary 2.

5 Open Problems

The internal lazy search trees [44] discuss further two operations on the structure; *Split* and *Merge*, which allows for splitting or merging disjoint lazy search tree structures around some query element or rank. These operations are supported as efficient as queries in the internal-model. In the external-memory case, the gap structure designed in this paper, however, does not easily allow for splitting or merging the set of gaps around an element or rank, as the gaps are stored by element in disjoint buckets, where all buckets must be updated upon executing these extra operations. By sorting and scanning the gaps, the operations may be supported, but not as efficiently as a simple query, but instead in sorting time relative to the number of gaps. It remains an open problem to design a gap structure, which allows for efficiently supporting Split and Merge.

In the external-memory model, bulk operations are generally faster, as scanning consecutive elements saves a factor B I/Os. One such operation is *RangeQuery*, where a range of elements may be queried and reported at once. In a B-tree, this operation is supported in $\mathcal{O}(\log_B N + k/N)$ I/Os, when k elements are reported. The lazy B-tree designed in this paper allows for efficiently reporting the elements of a range in unsorted order, by querying the endpoints of the range, and then reporting all elements of the gaps between the endpoints. Note that for the priority-queue case, this allows reporting the k smallest elements in $\mathcal{O}\big(\frac{1}{B}k\log_2 \frac{N}{k} + \frac{1}{B}\log_2 N + \log_B \log_B N\big)$ I/Os. If the elements must be reported in sorted order, sorting may be applied to the reported elements. However, this effectively queries all elements of the range, and should therefore be reflected into the lazy B-tree. As this introduces many gaps of small size, the I/O cost increases over simply sorting the elements of the range. It remains an open problem on how to efficiently support sorted RangeQuery operations, while maintaining the properties of lazy B-trees.

In the internal-memory model, an optimal version of lazy search trees was constructed [46], which gets rid of the added $\log\log N$ term on all operations. It remains an open problem to similarly get rid of the added $\log_B \log_B N$ term for the external-memory model.

Improvements on external-memory efficient search trees and priority queues use buffering of updates to move more data in a single I/O, thus improving the I/O cost of operations. The first hurdle to overcome in order to create buffered lazy B-trees is creating buffered biased trees used for the gap structure. By buffering updates to the gaps, it must then hold that the weights are not correctly updated, which imposes problems on searching; both by element and rank. It remains an open problem to overcome this first hurdle as a step towards buffering lazy B-trees.

References

1 Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962.

2 Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. doi:10.1145/48529.48535.

3 Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Workshop on Algorithms and Data Structures (WADS)*, pages 334–345. Springer, 1995. doi:10.1007/3-540-60220-8_74.

4 Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/S00453-003-1021-X.

5 Amitabha Bagchi, Adam L Buchsbaum, and Michael T Goodrich. Biased skip lists. *Algorithmica*, 42:31–48, 2005. doi:10.1007/S00453-004-1138-6.

6 Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jon Sorenson. Dynamic online multiselection in internal and external memory. In *WALCOM: Algorithms and Computation*, pages 199–209. Springer, 2015. doi:10.1007/978-3-319-15612-5_18.

7 Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jon Sorenson. Near-optimal online multiselection in internal and external memory. *Journal of Discrete Algorithms*, 36:3–17, 2016. WALCOM 2015. doi:10.1016/J.JDA.2015.11.001.

8 Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.

9 Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985. doi:10.1137/0214041.

10 Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976. doi:10.1016/0020-0190(76)90071-5.

11 Alka Bhushan and Sajith Gopalan. *External Memory Soft Heap, and Hard Heap, a Meldable Priority Queue*, pages 360–371. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-32241-9_31.

12 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

13 Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and B-trees. In *Symposium on Discrete Algorithms (SODA)*, pages 1106–1114. SIAM, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347203.

14 Gerth Stølting Brodal. Worst-case efficient priority queues. In *Symposium on Discrete Algorithms (SODA)*, 1996.

15 Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554. Society for Industrial and Applied Mathematics, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644201.

16 Gerth Stølting Brodal and Jyrki Katajainen. Worst-case efficient external-memory priority queues. In Stefan Arnborg and Lars Ivansson, editors, *Algorithm Theory — SWAT'98*, pages 107–118, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:10.1007/BFb0054359.

17 Gerth Stølting Brodal and Sebastian Wild. Funnelselect: Cache-oblivious multiple selection. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *European Symposium on Algorithms (ESA)*, pages 25:1–25:17, 2023. doi:10.4230/LIPIcs.ESA.2023.25.

18 Gerth Stølting Brodal and Sebastian Wild. Deterministic cache-oblivious funnelselect. In *Scandinavian Symposium on Algorithm Theory (SWAT)*, 2024. doi:10.4230/LIPIcs.SWAT.2024.17.

19 Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict fibonacci heaps. In *Symposium on Theory of Computing (STOC)*, STOC'12, pages 1177–1184. ACM, May 2012. doi:10.1145/2213977.2214082.

20 Yu-Tai Ching, Kurt Mehlhorn, and Michiel H.M. Smid. Dynamic deferred data structuring. *Information Processing Letters*, 35(1):37–40, 1990. doi:10.1016/0020-0190(90)90171-S.

21 Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Pătraşcu. The geometry of binary search trees. In *Symposium on Discrete Algorithms SODA 2009*, pages 496–505. SIAM, 2009. doi:10.1137/1.9781611973068.55.

22 David Dobkin and J. Ian Munro. Optimal time minimal space selection algorithms. *Journal of the Association for Computing Machinery*, 28(3):454–461, 1981. doi:10.1145/322261.322264.

23 Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. Decreasekeys are expensive for external memory priority queues. In *Symposium on Theory of Computing (STOC)*, pages 1081–1093. ACM, June 2017. doi:10.1145/3055399.3055437.

24 Joan Feigenbaum and Robert E Tarjan. Two new kinds of biased search trees. *Bell System Technical Journal*, 62(10):3139–3158, 1983. doi:10.1002/J.1538-7305.1983.TB03469.X.

25 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987. doi:10.1145/28869.28874.

26 Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.

27 Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: towards robust adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 5(6):502–513, February 2012. doi:10.14778/2168651.2168652.

28 John Iacono, Riko Jacob, and Konstantinos Tsakalidis. External memory priority queues with decrease-key and applications to graph algorithms. In *European Symposium on Algorithms (ESA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ESA.2019.60.

29 Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, CWI and University of Amsterdam, 2010.

30 Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Conference on Innovative Data Systems Research (CIDR)*, 2007.

31 Stratos Idreos, Stefan Manegold, and Goetz Graefe. Adaptive indexing in modern database kernels. In *International Conference on Extending Database Technology (EDBT)*, EDBT '12, pages 566–569. ACM, March 2012. doi:10.1145/2247596.2247667.

32 Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with decreasekeys. In *Symposium on Discrete Algorithms (SODA)*, pages 1331–1343. Society for Industrial and Applied Mathematics, January 2019. doi:10.1137/1.9781611975482.81.

33 Kanela Kaligosi, Kurt Mehlhorn, J. Ian Munro, and Peter Sanders. Towards optimal multiple selection. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 103–114. Springer, 2005. doi:10.1007/11523468_9.

34 Haim Kaplan, László Kozma, Or Zamir, and Uri Zwick. Selection from heaps, row-sorted matrices, and $x + y$ using soft heaps. In *Symposium on Simplicity in Algorithms (SOSA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/OASICS.SOSA.2019.5.

35 Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. Deferred data structuring. *SIAM Journal on Computing*, 17(5):883–902, 1988. doi:10.1137/0217055.

36 Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *International Conference on Management of Data*, SIGMOD/PODS '18, pages 489–504. ACM, May 2018. doi:10.1145/3183713.3196909.

37 Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996*, pages 169–176. IEEE Computer Society, 1996. doi:10.1109/SPDP.1996.570330.

38 Joan M. Lucas. Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Rutgers University, 1988.

39 Conrado Martínez and Salvador Roura. Randomized binary search trees. *J. ACM*, 45(2):288–323, 1998. doi:10.1145/274787.274812.

40 Kurt Mehlhorn. *Data Structures and Efficient Algorithms, Volume 1: Sorting and Searching.* Springer, 1984.

41 J. Ian Munro. On the competitiveness of linear search. In *ESA 2000*, pages 338–345. Springer, 2000. doi:10.1007/3-540-45253-2_31.

42 J. Ian Munro, Richard Peng, Sebastian Wild, and Lingyi Zhang. Dynamic optimality refuted – for tournament heaps. working paper, 2019. arXiv:1908.00563.

43 Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin Kersten. Database cracking: fancy scan, not poor man's sort! In *International Workshop on Data Management on New Hardware (DaMoN)*, SIGMOD/PODS'14, pages 1–8. ACM, June 2014. doi:10.1145/2619228.2619232.

44 Bryce Sandlund and Sebastian Wild. Lazy search trees. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 704–715. IEEE, 2020. doi:10.1109/FOCS46700.2020.00071.

45 Bryce Sandlund and Sebastian Wild. Lazy search trees, 2020. arXiv:2010.08840.

46 Bryce Sandlund and Lingyi Zhang. Selectable heaps and optimal lazy search trees. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1962–1975, 2022. doi:10.1137/1.9781611977073.78.

47 Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. doi:10.1007/BF01940876.

48 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.

49 Yufei Tao. Maximizing the optimality streak of deferred data structuring (a.k.a. database cracking). In *International Conference on Database Theory (ICDT)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ICDT.2025.10.

50 Fatemeh Zardbani, Peyman Afshani, and Panagiotis Karras. Revisiting the theory and practice of database cracking. In *International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2020. doi:10.5441/002/EDBT.2020.46.