External-Memory Priority Queues with Optimal Insertions

Gerth Stølting Brodal
Aarhus University, Denmark

Michael T. Goodrich
University of California, Irvine, USA

John Iacono
Université libre de Bruxelles, Belgium

Jared Lo
University of Hawai'i at Mānoa, USA

Ulrich Meyer
Goethe University Frankfurt am Main, Germany

Victor Pagan
University of Hawai'i at Mānoa, USA

Nodari Sitchinava
University of Hawai'i at Mānoa, USA

Rolf Svenning
Aarhus University, Denmark

Abstract

We present an external-memory priority queue structure supporting Insert and DeleteMin with amortized $\mathcal{O}(1)$ and $\mathcal{O}(\lg N)$ comparisons, respectively, and amortized $\mathcal{O}\left(\frac{1}{B}\right)$ and $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ I/Os, respectively. Here, M is the size of the internal memory, B is the block size of I/Os between internal and external memory, and N is the number of elements in the priority queue just before an operation is performed. Previous external-memory priority queues required amortized $\mathcal{O}(\lg N)$ comparisons and $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ I/Os for both Insert and DeleteMin. The construction requires the minimal assumption $M \geq 2B$.

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

Keywords and phrases priority queues, external memory, cache aware, amortized complexity

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.3

Funding *Gerth Stølting Brodal*: Supported by Independent Research Fund Denmark, grant 9131-00113B.
Michael T. Goodrich: Supported by NSF grant 2212129.
John Iacono: Research supported by the Fonds de la Recherche Scientifique — FNRS.
Jared Lo: Supported by NSF grant 2432018.
Ulrich Meyer: Supported by Deutsche Forschungsgemeinschaft (DFG) - ME 2088/5-2 (FOR 2975 - Algorithms, Dynamics, and Information Flow in Networks).
Victor Pagan: Supported by NSF grant 2432018.
Nodari Sitchinava: Supported by NSF grant 2432018.
Rolf Svenning: Supported by Independent Research Fund Denmark, grant 9131-00113B.

Acknowledgements Work initiated while attending the Third AlgoPARC Workshop on Parallel Algorithms and Data Structures at the University of Hawaii at Manoa, in part supported by the National Science Foundation under Grant No. 2452276.

1 Introduction

Priority queues are fundamental data structures with a host of applications. For example, algorithmic applications for priority queues range from classic results for sorting (such as heapsort) [13, 31, 32] and network optimization [19] to recent instance-optimal results for graph algorithms [22, 23]. In addition, priority queues have been developed for external-memory models [3, 6, 12, 18, 25, 26], ideal-cache models [4, 9], concurrent models [29], and RAM models [30]. Thus, it is desirable to design efficient priority queue data structures.

Throughout the paper, we assume a priority queue stores a multi-set of elements, where each element is a $\langle key, value\rangle$ pair, and the keys are from some totally ordered universe.

In the following, a comparison between two elements refers to the comparison of the two elements' keys.

Part of the reason priority queues are so broadly applicable is that they are defined in terms of two simple and useful operations:

- Insert(e) inserts an element e.
- DeleteMin() extracts and returns an element from the priority queue with minimum key. If several elements have equal keys, an arbitrary one of those is returned. This operation requires the priority queue to be non-empty before the operation.

Moreover, in internal memory, it is possible to design priority queues that hold N elements and achieve $\mathcal{O}(1)$ time for Insert and $\mathcal{O}(\lg N)$ time for DeleteMin, either in the worst case [14] or amortized [13, 31][1]. Unfortunately, prior to this work, we are not aware of any efficient data structures in external-memory models [1, 20, 21] that can match the performance of these classic internal-memory data structures.

Internal-memory priority queues

Williams introduced the binary heap in 1964 [32], which supports Insert and DeleteMin with $\mathcal{O}(\lg N)$ comparisons. Since then, many priority queues have been proposed; see, e.g., the survey by Brodal [7]. Vuillemin [31] introduced the binomial queue and demonstrated that it supports a sequence of N Insert and DeleteMin operations using a total of $\mathcal{O}(N \lg N)$ comparisons. Shortly after, Brown [13] gave a detailed analysis of binomial queues. Binomial queues support a sequence of I Insert and D DeleteMin operations in total $\mathcal{O}(I + D \lg I)$ comparisons. Carlson, Munro, and Poblete [14] showed how to achieve binomial queues supporting Insert with worst-case $\mathcal{O}(1)$ comparisons and DeleteMin with worst-case $\mathcal{O}(\lg N)$ comparisons. Fibonacci heaps, introduced by Fredman and Tarjan [19], achieve the same amortized performance as binomial queues for Insert and DeleteMin operations. Additionally, they support the DecreaseKey operation in amortized $\mathcal{O}(1)$ time and comparisons, allowing the key of an element to be replaced with a smaller key. Relaxed heaps, proposed by Driscoll, Gabow, Shrairman, and Tarjan [16], matched these bounds in the worst case.

External-memory model and priority queues

Aggarwal and Vitter [1] introduced the external-memory model as a model of computation focusing on the communication between two levels of memory: an *internal memory* of size M and an infinite *external memory*, where an *I/O* transfers a block of B consecutive memory cells between the two levels of memory. The *I/O cost* of an algorithm is the number of I/Os the algorithm performs. The minimal assumption is that $B \geq 1$ and $M \geq 2B$.

Aggarwal and Vitter proved that comparison-based sorting of N elements in the external-memory model can be done with $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os, and proved a matching lower bound for comparison-based sorting. Since sorting can be performed using a priority queue by performing N Insert operations followed by N DeleteMin operations, the amortized I/O cost of either Insert or DeleteMin must be $\Omega\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os. Arge [3], Kumar and Schwabe [27], and Fadel, Jakobsen, Katajainen, and Teuhola [18] presented external-memory priority queues supporting Insert and DeleteMin in amortized $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$

[1] $\lg x$ denotes the binary logarithm of x.

I/Os, assuming $M \geq 2B$. Brengel, Crauser, Ferragina and Meyer [6] presented a simple external-memory priority queue denoted an *array heap* with matching I/O bounds, assuming $B \log_{M/B} \frac{N}{B} \leq M$. Brodal and Katajainen [12] achieved a matching worst-case I/O cost. Since the number of DeleteMin operations is always no more than the number of Insert operation, we can charge the deletion cost to the insertions, i.e., we can restate the amortized costs to be $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os for Insert and $\mathcal{O}(1/B)$ I/Os for DeleteMin (or even zero). The question we address in this paper is whether we can swap the two I/O costs, i.e., achieve amortized $\mathcal{O}(1/B)$ I/Os for Insert and $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os for DeleteMin, which is relevant when the number of Insert operations is asymptotically larger than the number of DeleteMin operations, i.e., not all elements inserted into the priority queue are eventually also extracted. Table 1 summarizes previous results, where we let $\mathrm{Sort}(N) = \frac{N}{B} \log_{M/B} \frac{N}{B}$; hence, $\frac{1}{N}\mathrm{Sort}(N) = \frac{1}{B} \log_{M/B} \frac{N}{B}$.

Frigo, Leiserson, Prokop, and Ramachandran [20, 21] introduced the *ideal-cache* model by augmenting the external-memory model with an optimal offline paging mechanism. In this model, instead of having each algorithm explicitly transfer data between the two levels of memory, the data transfers are assumed to be performed by an optimal offline paging mechanism. While this might seem like a strong assumption, they demonstrated that under reasonable resource augmentation assumptions, a more realistic paging mechanism, e.g., the one implementing *least recently used (LRU)* replacement policy, is competitive with an optimal one. Consequently, ideal-cache model allows the design of so-called *cache-oblivious* algorithms – algorithms that are oblivious to the parameters M and B. Frigo et al. [20, 21] presented cache-oblivious sorting algorithms achieving optimal I/O cost, provided a *tall-cache assumption* of $M \geq B^2$. Brodal and Fagerberg [8] proved that the same bounds can be achieved under the weaker tall cache assumption $M \geq B^{1+\varepsilon}$ for any constant $\varepsilon > 0$, with a constant overhead of $\frac{1}{\varepsilon}$ in the I/O bounds. Brodal and Fagerberg [10] proved that this trade-off between the tall-cache assumption and the I/O cost is inherent to cache-oblivious algorithms. Cache-oblivious priority queues were presented by Arge, Bender, Demaine, Holland-Minkley, and Munro [4] and Brodal and Fagerberg [9] achieving $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os for Insert and DeleteMin, under the tall cache assumptions $M \geq B^2$ and $M \geq B^{1+\varepsilon}$, respectively.

In internal memory, several priority queues support Insert and DeleteMin in amortized $\mathcal{O}(\lg N)$ time, and additionally a DecreaseKey operation in amortized $\mathcal{O}(1)$ time (some also in the worst-case and Insert in $\mathcal{O}(1)$ time). A natural question is if it is possible to achieve similar results in external memory, i.e., N Insert and DeleteMin operations with $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os and DecreaseKey with $\mathcal{O}\left(\frac{1}{B}\right)$ I/Os. Eenberg, Larsen, and Yu [17] proved a lower bound that this is not possible. An external-memory priority queue structure by Kumar and Schwabe [27] supports each of the three operations with amortized $\mathcal{O}\left(\frac{1}{B} \lg \frac{N}{B}\right)$ I/Os. Similar bounds were achieved cache-obliviously by Brodal, Fagerberg, Meyer, and Zeh [11] and Chowdhury and Ramachandran [15]. External-memory priority queues with improved DecreaseKey operations were presented by Iacono, Jacob, and Tsakalidis [25], who support Update (a combination of Insert and DecreaseKey) and DeleteMin in amortized $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ and $\mathcal{O}\left(\left\lceil \frac{M^\varepsilon}{B} \log_{M/B} \frac{N}{B} \right\rceil \log_{M/B} \frac{N}{B}\right)$ I/Os, respectively, and by Jiang and Larsen [26], who support Insert, DeleteMin and DecreaseKey in expected amortized time $\mathcal{O}\left(\frac{1}{B} \lg \frac{N}{B} / \lg \lg N\right)$ I/Os, assuming $M > B \lg^{0.01} N$.

Table 1 Selected previous and new comparison and I/O bounds for priority queues. $\mathcal{O}_A$ denotes amortized bounds, whereas $\mathcal{O}$ denotes worst-case bounds.

	Comparisons		I/Os	
	Insert	DeleteMin	Insert	DeleteMin
Internal memory				
Binary heap [32]	$\mathcal{O}(\lg N)$	$\mathcal{O}(\lg N)$		
Binomial queue [13, 31]	$\mathcal{O}_A(1)$	$\mathcal{O}_A(\lg N)$		
Binomial queue [14]	$\mathcal{O}(1)$	$\mathcal{O}(\lg N)$		
External memory				
Buffer tree [3]	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$
Buffered multiway heap [18, 27]				
Brodal-Katajainen [12]	$\mathcal{O}(\lg N)$	$\mathcal{O}(\lg N)$	$\mathcal{O}\left(\frac{1}{N}\text{Sort}(N)\right)$	$\mathcal{O}\left(\frac{1}{N}\text{Sort}(N)\right)$
New (Theorem 1)	$\mathcal{O}_A(1)$	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A\left(\frac{1}{B}\right)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$
Cache oblivious				
Arge et al. [4], Funnel-heap [9]	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$

Result

In this paper, we present an external-memory priority queue that is optimal both with respect to the amortized I/Os and comparisons performed, achieving the following result.

▶ **Theorem 1.** *There exists an external-memory priority queue supporting* Insert *with amortized* $\mathcal{O}(1)$ *comparisons and* $\mathcal{O}\left(\frac{1}{B}\right)$ *I/Os, and* DeleteMin *with amortized* $\mathcal{O}(\lg N)$ *comparisons and* $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ *I/Os, where* N *is the current number of elements in the priority queue. The space usage is* $\mathcal{O}\left(\frac{N}{B}\right)$ *blocks. The memory size only needs to satisfy the minimal requirement* $M \geq 2B$.

We achieve this result using what can intuitively be considered an element "juggling" scheme, where we maintain three types of data structures—two in internal memory and one in external memory, transferring elements between them as needed. In a nutshell, our result is obtained by maintaining an internal-memory priority queue with the $\mathcal{O}(M)$ smallest elements supporting Insert and DeleteMin with $\mathcal{O}(1)$ and $\mathcal{O}(\lg M)$ comparisons. We also maintain an insert-buffer in internal memory as a buffer between the internal-memory priority queue and an external-memory structure comprising a forest of multi-way heaps. Specifically, in external memory, we store multi-way heap structures with buffers at the nodes, somewhat similar to an approach by Fadel, Jakobsen, Katajainen, and Teuhola [18] but adapted to support the insertion of batches of $\mathcal{O}(M)$ elements using $\mathcal{O}\left(\frac{M}{B}\right)$ I/Os. Whereas nodes in this previous construction store sorted buffers, in our construction, we maintain the forest of heaps in external memory to use partially sorted buffers (*semi-sorted* at internal nodes and *lazy semi-sorted* at the leaves) to avoid insertions requiring amortized $\Omega(\lg M)$ comparisons.

2 External-memory priority queue

In Sections 3–7, we describe the details of an external-memory priority queue achieving Lemma 2 below, where we bound the total number of comparisons and I/Os performed by a sequence of priority queue operations in terms of the total numbers of Insert and DeleteMin operations performed, denoted I and D, respectively. In Section 8, we then apply global rebuilding to make the bounds depend on the current number of elements in the priority queue, achieving the amortized bounds in Theorem 1.

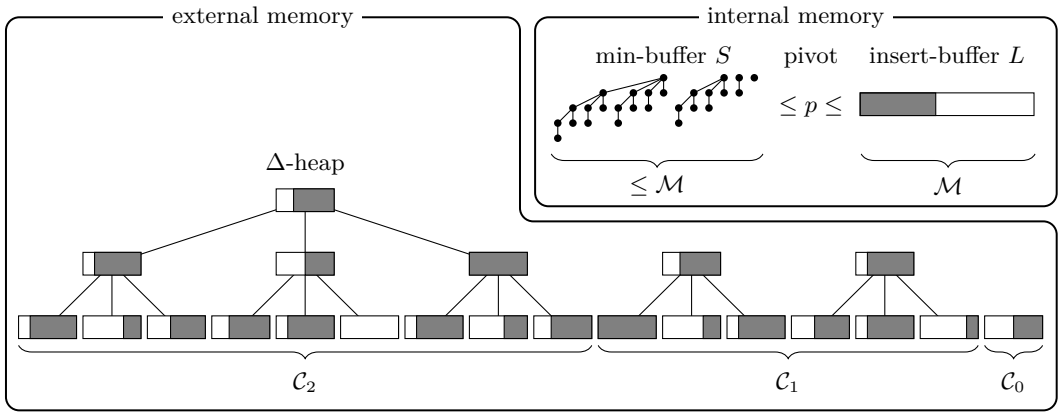

Figure 1 External-memory priority queue. All rectangles are buffers of capacity $\mathcal{M}$, where the gray area illustrates elements in buffers.

▶ **Lemma 2.** *There exists an external-memory priority queue supporting a sequence of I* Insert *and D* DeleteMin *operations, using a total of $\mathcal{O}(I + D \lg I)$ comparisons and $\mathcal{O}\left(\frac{1}{B}\left(I + D \log_{M/B} \frac{I}{B}\right)\right)$ I/Os, assuming $M \geq 2B$.*

Our priority queue consists of an internal-memory part and an external-memory part; see Figure 1 for an overview. Each element is stored exactly once in our data structure. In internal memory we store a *min-buffer*, a *pivot* element, and an *insert-buffer*, where the elements in the min-buffer are all smaller than or equal to the pivot, and the elements in the insert-buffer and external memory are all larger than or equal to the pivot element. By default, Insert and DeleteMin are performed in internal memory without performing any I/Os. If internal memory contains too many elements, $\Theta(M)$ elements are moved from the insert-buffer to the external part. To perform DeleteMin when the min-buffer is empty, $\Theta(M)$ smallest elements from external memory are moved to internal memory. We call moving $\Theta(M)$ elements between internal and external memory a *transfer*. In the subsequent sections, we describe the two parts in detail. We let $\mathcal{M}$ be a parameter for our construction, such that $\mathcal{M}$ is even, $\mathcal{M}$ is divisible by B, and $\mathcal{M} = \Theta(M)$. We choose $\mathcal{M}$ such that the $\mathcal{O}(\mathcal{M})$ space internal-memory data structure in Section 3 fits into internal memory. To be able to choose $\mathcal{M}$, M is required to be sufficiently large and the internal memory to be able to hold a sufficiently large number of blocks of size B. If $M = \mathcal{O}(1)$, then an internal-memory data structure like a binomial queue already achieves the result of Lemma 2, and if the number of blocks fitting in internal memory is too small we can simulate a smaller block size with a constant blow up in the I/O complexity. I.e., in the following we w.l.o.g. can assume that M and M/B are sufficiently large to be able to choose $\mathcal{M}$.

3 The internal-memory part

The internal-memory part consists of a *min-buffer* S, a *pivot* element p, and an *insert-buffer* L. The pivot is larger than or equal to any element in the min-buffer, and smaller than or equal to any element in the insert-buffer and external memory. The insert-buffer is an unordered array of at most $\mathcal{M}$ elements. The min-buffer stores the overall at most $\mathcal{M}$ smallest elements in the priority queue, and is implemented using an internal-memory linear-space priority queue supporting Insert in amortized $\mathcal{O}(1)$ time and DeleteMin in amortized $\mathcal{O}(\lg n)$ time, where n is the number of elements in the min-buffer. The min-buffer can, e.g., be

implemented using a binomial queue [31] or the implicit post-order heap by Harvey and Zatloukal [24]. The total space for the internal part is $\mathcal{O}(\mathcal{M})$.

Insert(e) first compares e to the pivot p. If $e \leq p$, e is inserted in the min-buffer using the min-buffer's Insert operation. Otherwise, e is appended to the insert-buffer. If the min-buffer overflows, i.e., gets size $\mathcal{M}+1$, we find a new pivot p' by applying the linear-time internal-memory median finding algorithm by Blum, Floyd, Pratt, Rivest, and Tarjan [5] to the elements in the min-buffer. The median, i.e., the $\left(\frac{\mathcal{M}}{2}+1\right)$th smallest element, becomes the new pivot p', the $\frac{\mathcal{M}}{2}$ larger elements and the old pivot p are appended to the insert-buffer, and the $\frac{\mathcal{M}}{2}$ smaller elements are inserted into a new empty min-buffer structure using its Insert operations. If the insert-buffer overflows, i.e., gets size $> \mathcal{M}$ (due to the insertion of a single element or the old pivot p and $\frac{\mathcal{M}}{2}$ elements being moved from the min-buffer to the insert-buffer), we transfer $\mathcal{M}$ elements from the insert-buffer to the external part (see Section 4).

For DeleteMin, if the min-buffer is non-empty, the smallest element is extracted from the min-buffer using the min-buffer's DeleteMin operation and returned. Otherwise, the min-buffer is empty and the pivot p is the smallest element to be returned. To establish a new pivot p' and min-buffer, the $\frac{\mathcal{M}}{2}$ smallest elements are extracted from external memory (see Section 4) and transferred to the insert-buffer (except when the external part is empty). This ensures that the $\frac{\mathcal{M}}{2}$ smallest elements in the insert-buffer are now smaller than or equal to all elements in the external part. The $\frac{\mathcal{M}}{2}$th smallest element in the insert-buffer is selected as the new pivot p' in linear time using the internal-memory selection algorithm [5]. If the insert-buffer contains $< \frac{\mathcal{M}}{2}$ elements, we set the pivot p to be $+\infty$. The at most $\frac{\mathcal{M}}{2}-1$ elements smaller than or equal to the new pivot p' in the insert-buffer are deleted from the insert-buffer and inserted into the min-buffer structure using its Insert operation. The old pivot p is returned as the extracted minimum.

Note that a transfer either moves $\mathcal{M}$ elements from the internal to the external part, or $\frac{\mathcal{M}}{2}$ elements from the external to the internal part, i.e., the external memory always contains a multiple of $\frac{\mathcal{M}}{2}$ elements. Note also that the size of the insert-buffer is unchanged during DeleteMin, provided that there are elements in external memory. Otherwise, the number of elements in the insert-buffer decreases by $\frac{\mathcal{M}}{2}$.

Initially, the pivot $p = +\infty$, such that all elements are inserted into the min-buffer until the priority queue contains $\mathcal{M}+1$ elements, where a real element is selected as the pivot and $\frac{\mathcal{M}}{2}$ elements are moved to the insertion-buffer and a new min-buffer is created for the $\frac{\mathcal{M}}{2}$ smallest elements using repeated insertions.

4 The external-memory part

The external-memory part supports the insertion of a batch of $\mathcal{M}$ elements and the extraction of the $\mathcal{M}/2$ smallest elements. It consists of a set of Δ-*heaps*, where $\Delta = \frac{\mathcal{M}}{B} \geq 2$. A Δ-heap with height h is a tree where all leaves have depth h (the root having depth zero), and all non-leaves have exactly Δ children. Each node contains a buffer storing up to $\mathcal{M}$ elements. The elements must satisfy *extended heap order*, i.e., the elements in the buffer of a node u are smaller than or equal to any element in the buffers of the children of u. A Δ-heap is a generalization of a binary heap [32] and is very similar to the external-memory priority queue of Fadel, Jakobsen, Katajainen and Teuhola [18], except that we do not require buffers to be sorted (this would not be possible when insertions only perform amortized $\mathcal{O}(1)$ comparisons.) The buffers in a Δ-heap store elements in various degrees of sortedness to be able to achieve the stated insertion cost: non-leaves are *semi-sorted*, whereas leaves are *lazy semi-sorted* as

discussed in Sections 5 and 6, respectively.

The Δ-heaps are maintained as a sequence of collections $\mathcal{C}_0, \ldots, \mathcal{C}_H$, where $\mathcal{C}_h$ contains all Δ-heaps with height h. We maintain the invariant (**I1**) that $|\mathcal{C}_h| < \Delta$ and that the total number of leaves in the Δ-heaps is $\leq \frac{I}{\mathcal{M}}$ (Lemma 7), where I is the total number of insertions. This invariant ensures that the maximum height H of a Δ-heap is $\leq \lfloor \log_\Delta \frac{I}{\mathcal{M}} \rfloor$. The buffers of all nodes must satisfy the following invariant (**I2**): The buffer of a node u stores at most $\mathcal{M}$ elements. If no descendant of u stores elements, there is no lower bound on the buffer size of u. Otherwise, the buffer of u contains at least $\frac{\mathcal{M}}{2}$ elements. Together with the extended heap order, this invariant implies that the $\frac{\mathcal{M}}{2}$ smallest elements in a Δ-heap are stored in the root buffer, except if the Δ-heap contains fewer than $\frac{\mathcal{M}}{2}$ elements, in which case all elements are stored in the root buffer.

When $\mathcal{M}$ elements are transferred from the internal to the external part, the $\mathcal{M}$ elements become a single node Δ-heap with height zero that is added to collection $\mathcal{C}_0$. If the collection is full, that is, $|\mathcal{C}_0| = \Delta$, then a new root r is created for a Δ-heap of height one with each leaf of $\mathcal{C}_0$ as its children. Invariant **I2** is established for r by recursively *pulling* the $\frac{\mathcal{M}}{2}$ smallest elements from its children and inserting them in the buffer (the pull operation is described below). This leaves $\mathcal{C}_0$ empty, and r is promoted to $\mathcal{C}_1$. As in a Δ-ary number system, this promotion may propagate up through the collections, repeatedly combining Δ Δ-heaps with height h to a Δ-heap with height $h+1$.

For a node with $< \frac{\mathcal{M}}{2}$ elements in its buffer, a *pull* operation moves $\frac{\mathcal{M}}{2}$ elements from its children to the buffer of the node while preserving extended heap order, using Lemma 5 in Section 5. Note that this will make the buffer contain at least $\frac{\mathcal{M}}{2}$ elements and less than $\mathcal{M}$. If a child buffer gets size $< \frac{\mathcal{M}}{2}$, we recursively apply the pull operation to the child, provided that not all subtrees below the child are exhausted.

To extract the $\frac{\mathcal{M}}{2}$ smallest elements from the external part, first, for each collection $\mathcal{C}_i$, the $\frac{\mathcal{M}}{2}$ smallest elements are found by considering the elements in the buffers of the roots by applying Lemma 5 to the roots of the collection. As the elements are just identified but not pulled from the roots, we call this a *virtual pull*. Second, the $\frac{\mathcal{M}}{2}$ smallest elements among the $(H+1)\frac{\mathcal{M}}{2}$ candidates from the virtual pulls are found by applying a linear-time selection algorithm, Corollary 3 below. These elements are removed from their respective roots and transferred to the internal part. For each root buffer now containing $< \frac{\mathcal{M}}{2}$ elements, we recursively pull $\frac{\mathcal{M}}{2}$ elements from its children, to the extent possible.

Note that the total number of elements in external memory is always a multiple of $\frac{\mathcal{M}}{2}$, but this is not necessarily true for each Δ-heap due to the extractions of the smallest $\frac{\mathcal{M}}{2}$ elements across all Δ-heaps.

5 Semi-sorted lists

Given two sequences X and Y, we say $X \preceq Y$, if for every $x \in X$ and $y \in Y$: $x \leq y$. Note that $X \preceq Y$ implies no specific order within each individual sequence X or Y.

A buffer in external memory is a linked list of blocks $b_1, b_2, \ldots, b_\delta$, where each block contains B elements, except possibly the first and last blocks, which contain $\leq B$ elements. In the following, we simply denote such a linked list as a *list*. We define a list to be *semi-sorted* if $b_i \preceq b_{i+1}$ for every $1 \leq i < \delta$. All buffers of non-leaf nodes in a Δ-heap are stored as semi-sorted lists. The following are useful tools for working with semi-sorted buffers.

Blum, Floyd, Pratt, Rivest, and Tarjan [5] proved that the kth smallest element in a list with N elements can be found using $\mathcal{O}(N)$ comparisons using a double recursion making repeated use of scanning lists. Analyzed in the external-memory model, this immediately

implies that selection can be performed in a linear number of I/Os, and we have the following corollary.

▶ **Corollary 3** (Blum et al. [5]). *Given a list of N elements and an integer $1 \leq k \leq N$, the list can be partitioned into two lists X and Y, with $|X| = k$ and $X \preceq Y$, using $\mathcal{O}(N)$ comparisons and $\mathcal{O}\left(\frac{N}{B}\right)$ I/Os.*

▶ **Lemma 4** (Constructing semi-sorted buffer). *An unsorted list of $B\delta \leq \mathcal{M}$ elements can be converted into a semi-sorted list using $\mathcal{O}(B\delta \lg \delta)$ comparisons and $\mathcal{O}(\delta)$ I/Os.*

Proof. Recursively apply [5] to partition an unsorted list with δB elements into two subproblems with $\lceil \delta/2 \rceil B$ and $\lfloor \delta/2 \rfloor B$ elements, until the subproblems are of size exactly B. The result follows since there are $\lceil \lg \delta \rceil$ levels of recursion with a linear number of comparisons per level, and we only perform I/Os to read the initial list into internal memory and to write the final list to external memory, since the problem fits in internal memory. ◀

▶ **Lemma 5** (Pulling from semi-sorted lists). *Given at most $\Delta = \frac{\mathcal{M}}{B}$ semi-sorted lists, a semi-sorted list of the $\frac{\mathcal{M}}{2}$ smallest elements of their union can be computed in $\mathcal{O}(\mathcal{M} \lg \Delta)$ comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os.*

Proof. Read the first block from each semi-sorted list into internal memory and insert the largest element of each block into a binary min-heap Q [32]. Then, delete the smallest element from Q, read the next block from the corresponding semi-sorted list, and insert the largest element of this block into Q. Let X denote the elements in blocks whose maximums have been deleted from Q, and Y the elements in blocks whose maximums are in Q. Repeat this process of reading blocks until $|X| \geq \frac{\mathcal{M}}{2}$. In total, $\Theta\left(\frac{\mathcal{M}}{B}\right)$ blocks are read, where the maximum is computed of each block and each block requires $\mathcal{O}(1)$ heap operations on Q, where $|Q| \leq \Delta$. This initial phase requires a total of $\mathcal{O}\left(\mathcal{M} + \frac{\mathcal{M}}{B} \cdot \lg \Delta\right)$ comparisons and $\Theta\left(\frac{\mathcal{M}}{B}\right)$ I/Os.

Let x be the last element deleted from Q. We have $\max(X) \leq x$, since when the lists are semi-sorted, elements will be extracted in non-decreasing order from Q. From each list, we have a block represented in Q with maximum $\geq x$, so all remaining unread elements in the lists are also $\geq x$. Thus, the $\frac{\mathcal{M}}{2}$ smallest elements can be found by a single partition of $X \cup Y$ using $\mathcal{O}(\mathcal{M})$ comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os per Corollary 3. They can then be converted to a semi-sorted list using additional $\mathcal{O}(\mathcal{M} \lg \Delta)$ comparisons per Lemma 4. ◀

6 Lazy semi-sorted lists

Since I insertions can create $\Theta(I/\mathcal{M})$ leaf buffers of size $\mathcal{M}$, they cannot all be semi-sorted, as that would require $\Theta(I \lg \Delta)$ comparisons. The lower bound of $\Omega(\mathcal{M} \lg \Delta)$ comparisons for the multiple selection problem for each buffer is due to Dobkin and Munro [2, Theorem 1]. Instead, a leaf buffer with $\mathcal{M}$ elements is stored as a *lazy semi-sorted* list, see Figure 2. A lazy semi-sorted list stores the $\mathcal{M}$ elements as a sequence of *chunks* $c_1 \preceq c_2 \preceq \cdots \preceq c_d$, where $d = \left\lceil \lg \frac{\mathcal{M}}{B} \right\rceil + 1$ and, initially, the elements inside a chunk are stored in arbitrary order. Chunk c_1 stores B elements, c_i stores $B2^{i-2}$ elements for $2 \leq i < d$, and c_d stores the largest $\mathcal{M} - B2^{d-2}$ elements. If each chunk is converted into a semi-sorted list, and all chunks are concatenated, we have a semi-sorted list for the buffer. The basic idea is to exploit that a Δ-heap always accesses the blocks of a semi-sorted buffer left-to-right, implying that we can delay expanding a chunk into a semi-sorted sequence of blocks using Lemma 4 until the first block of the (semi-sorted version of the) chunk is accessed, hence the name lazy semi-sorted.

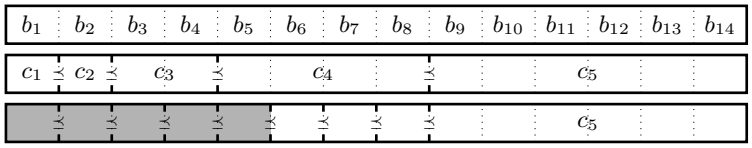

Figure 2 A lazy semi-sorted list. Top: Initial unsorted list of blocks $b_1, \ldots, b_{\mathcal{M}/B}$. Middle: Initial chunks $c_1 \preceq \cdots \preceq c_5$. Bottom: The state after the first five blocks $b_1, \ldots, b_5$ in the semi-sorted list have been accessed, i.e., the chunks $c_1, \ldots, c_4$ have been made into semi-sorted sublists.

The initial partitioning of a buffer into chunks is done by recursively partitioning the half with the smaller elements using Corollary 3.

▶ **Lemma 6** (Lazy access cost). *Initializing a lazy semi-sorted list of $\mathcal{M}$ elements requires $\mathcal{O}(\mathcal{M})$ comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os. The cost of accessing the first $k \leq \frac{\mathcal{M}}{B}$ blocks of the lazy semi-sorted list requires $\mathcal{O}(Bk \lg k)$ comparisons and $\mathcal{O}(k)$ I/Os.*

Proof. To initialize the lazy semi-sorted list, all $\mathcal{M}$ elements are first read into internal memory using $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os. No further I/Os are performed, except for eventually writing the output to external memory. The initial chunks, with elements in arbitrary order, are constructed one at a time by repeatedly applying Corollary 3 to find the appropriate number of largest elements. By constructing the chunks in the order $c_d, c_{d-1}, \ldots c_0$ from the largest to the smallest, the number of remaining elements decreases geometrically, leading to a total of $\mathcal{O}(\mathcal{M})$ comparisons.

If the k first blocks of elements have been accessed, then chunks $c_1, \ldots, c_{\lceil \lg k \rceil + 1}$ have been semi-sorted, i.e., fewer than $2k$ blocks. Note that c_1 and c_2 only consist of a single block, i.e., they are already semi-sorted at initialization. By applying Lemma 4, semi-sorting the remaining $\lceil \lg k \rceil - 1$ accessed chunks requires $\mathcal{O}\left(\sum_{i=1}^{\lceil \lg k \rceil - 1} B 2^i \lg 2^i\right) = \mathcal{O}(Bk \lg k)$ comparisons and $\mathcal{O}\left(\sum_{i=1}^{\lceil \lg k \rceil - 1} 2^i\right) = \mathcal{O}(k)$ I/Os. ◀

The idea of lazy semi-sorted buffers is that they allow us to charge the non-linear comparison cost of constructing a semi-sorted buffer to the pull operations on the buffer, in particular, the pull operations triggered by deletions.

7 Analysis

When Insert and DeleteMin can be performed entirely within the internal part, no I/Os are performed. The number of comparisons is $\mathcal{O}(1)$ for Insert and $\mathcal{O}(\lg \min(\mathcal{M}, I))$ for DeleteMin, plus the cost of moving elements from the min-buffer to the insert-buffer. If an insertion causes the min-buffer to overflow, we spend $\mathcal{O}(\mathcal{M})$ comparisons moving elements to the insert-buffer and building a new min-buffer. This can only happen once every $\frac{\mathcal{M}}{2}$ insertion, so the total cost for this is $\mathcal{O}(I)$ comparisons and no I/Os.

A transfer either moves $\mathcal{M}$ elements from the internal to the external part, or $\frac{\mathcal{M}}{2}$ elements from the external to the internal part. The following lemma bounds the number of transfers in each direction in terms of I and D.

▶ **Lemma 7** (Bounding transfers). *The number of transfers from the internal to the external part is $\leq \frac{I}{\mathcal{M}}$. The number of transfers from the external to the internal part is $\leq \frac{2D}{\mathcal{M}}$.*

Proof. We first consider the number of transfers from the internal to the external part. Let $\phi \geq 0$ be the number of elements in the min-buffer beyond the first $\frac{\mathcal{M}}{2}$ elements plus the

number of elements in the insert-buffer. Insertions cause ϕ to increase by at most one, and deletions do not cause ϕ to increase. A transfer to the external part decreases ϕ by $\mathcal{M}$, whereas a transfer to the internal part of $\frac{\mathcal{M}}{2}$ elements leaves ϕ unchanged, since the min-buffer gets size $\frac{\mathcal{M}}{2} - 1$ and the size of the insert-buffer does not change. It follows that ϕ only increases during the I insertions, where the total increase is $\leq I$, and each transfer to the external part decreases ϕ by $\mathcal{M}$, i.e., the number of transfers to the external part is at most $\frac{I}{\mathcal{M}}$.

The size of the min-buffer can only decrease below $\frac{\mathcal{M}}{2}$ due to a deletion, where it decreases by one. (When the size of the min-buffer decreases because half of the elements are moved to the insert-buffer, the min-buffer changes size from $\mathcal{M}+1$ to $\frac{\mathcal{M}}{2}$.) A transfer to the internal part occurs when the size of the min-buffer is zero before the deletion, but then the min-buffer has size $\frac{\mathcal{M}}{2} - 1$ after the deletion and the next $\frac{\mathcal{M}}{2} - 1$ deletions do not cause a transfer to the internal part. It follows that at most every $\frac{\mathcal{M}}{2}$ deletion can cause a transfer to the internal part. The first transfer to the internal part can only happen after a transfer to the external part has occurred, which first can happen after the pivot $p \neq +\infty$, where the size of the min-buffer is $\frac{\mathcal{M}}{2}$, i.e., there must have been at least $\frac{\mathcal{M}}{2}$ deletions before the first transfer to the internal part. It follows that the number of transfers to the internal part is at most $\frac{2D}{\mathcal{M}}$. ◀

The key part of the analysis is threefold. First, we bound the cost to restore invariants **I1** and **I2** whenever elements are removed from the internal nodes of the external part. Second, we bound the cost of *accessing* the lazy semi-sorted leaves. The access cost of a leaf refers to the cost of semi-sorting its chunks. Third, we bound the cost involved with transfers between internal and external memory. We define the *height* of a node in a Δ-heap to be the height of the subtree rooted at the node (leaves have height zero).

▶ **Lemma 8** (Total pull cost excluding leaf access). *The total number of pulls is $\mathcal{O}\big(\frac{I}{\mathcal{M}\Delta} + \frac{DH}{\mathcal{M}}\big)$ and their total cost is $\mathcal{O}\Big(I\frac{\lg\Delta}{\Delta} + D\lg\frac{I}{\mathcal{M}}\Big)$ comparisons and $\mathcal{O}\big(\frac{I}{\mathcal{M}} + \frac{DH}{B}\big)$ I/Os, excluding the cost of accessing the leaves.*

Proof. A pull at a node increases the height of $\frac{\mathcal{M}}{2}$ elements from the buffers of the children by one, except the last pull at a node that exhausts all subtrees at the children, where at most $\frac{\mathcal{M}}{2}$ elements have their height increased by one. By Lemma 7, at most $\frac{I}{\mathcal{M}}$ leaves are created, i.e., the maximum height of a Δ-heap is $H \leq \log_\Delta \frac{I}{\mathcal{M}}$. For a height h, $1 \leq h \leq H$, we bound the number of pulls P_h to nodes with height h by bounding the number of elements that could have been moved to a height $\geq h$ or transferred to the internal memory, the latter being bounded by D by Lemma 7. Since there are $\leq \frac{I}{\mathcal{M}}$ leaves, the number of nodes at height h is $\leq \frac{I}{\mathcal{M}\Delta^h}$, implying that the number of elements that could ever have been pulled to height h (and possibly further up) is at most $D + \mathcal{M}\sum_{j=h}^{H}\frac{I}{\mathcal{M}\Delta^j}$. Since each non-exhausting pull moves $\frac{\mathcal{M}}{2}$ elements one level up, and each node with height h can have one final pull exhausting all children, we get that the final number of pulls to height h is $P_h \leq \Big(D + \mathcal{M}\sum_{j=h}^{H}\frac{I}{\mathcal{M}\Delta^j}\Big) \,/\, \frac{\mathcal{M}}{2} + \frac{I}{\mathcal{M}\Delta^h} = \mathcal{O}\big(\frac{I}{\mathcal{M}\Delta^h} + \frac{D}{\mathcal{M}}\big)$, as $\Delta \geq 2$.

The total number of pulls is then $\sum_{h=1}^{H} P_h = \mathcal{O}\big(\frac{I}{\mathcal{M}\Delta} + \frac{DH}{\mathcal{M}}\big)$. Excluding the cost of accessing the leaves, we may treat each pull as being from semi-sorted lists. Applying Lemma 5, the total number of comparisons is $\mathcal{O}\big(\big(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\log_\Delta\frac{I}{\mathcal{M}}\big)\,\mathcal{M}\lg\Delta\big) = \mathcal{O}\Big(I\frac{\lg\Delta}{\Delta} + D\lg\frac{I}{\mathcal{M}}\Big)$ and the number of I/Os is $\mathcal{O}\big(\big(\frac{I}{\mathcal{M}\Delta} + \frac{DH}{\mathcal{M}}\big)\,\frac{\mathcal{M}}{B}\big) = \mathcal{O}\big(\frac{I}{\mathcal{M}} + \frac{DH}{B}\big)$ I/Os. ◀

▶ **Lemma 9** (Total leaf access cost). *The total cost of constructing the lazy semi-sorted leaves and accessing them is $\mathcal{O}(I + D\lg\Delta)$ comparisons and $\mathcal{O}\big(\frac{I}{B} + \frac{D}{B}\big)$ I/Os.*

Proof. As described in the proof of Lemma 8, the number of pulls from the leaves is $P_1 = \mathcal{O}\left(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\right)$. Virtual pulls may, however, also access elements in the leaves. Since each transfer to the internal part causes one virtual pull from $\mathcal{C}_0$, the number of virtual pulls from the leaves is bounded by $\frac{2D}{\mathcal{M}}$ by Lemma 7. The total number of pulls and virtual pulls from the leaves is $\mathcal{O}\left(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\right)$.

By Lemma 6, the cost for the initial construction of the at most $\frac{I}{\mathcal{M}}$ leaves as lazy semi-sorted lists is $\mathcal{O}(I)$ comparisons and $\mathcal{O}\left(\frac{I}{B}\right)$ I/Os. Since, by Lemma 5, each pull and virtual pull performs $\mathcal{O}(\mathcal{M} \lg \Delta)$ comparisons and accesses $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ blocks, we have, by Lemma 6, that at the leaves, the total number of comparisons is $\mathcal{O}\left(I + \left(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\right)\mathcal{M} \lg \Delta\right) = \mathcal{O}(I + D \lg \Delta)$ comparisons and the total number of I/Os is $\mathcal{O}\left(\frac{I}{B} + \left(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\right)\frac{\mathcal{M}}{B}\right) = \mathcal{O}\left(\frac{I}{B} + \frac{D}{B}\right)$ I/Os. ◀

▶ **Lemma 10** (Cost of transfers excluding pulls and leaf accessing). *The total cost of transfers between the internal and external part is $\mathcal{O}\left(D \lg \frac{I}{\mathcal{M}}\right)$ comparisons and $\mathcal{O}\left(\frac{I}{B} + \frac{D}{B} \log_\Delta \frac{I}{\mathcal{M}}\right)$ I/Os, excluding the cost of pulling to restore the invariants and to access the leaves.*

Proof. By Lemma 7, there are at most $\frac{I}{\mathcal{M}}$ transfers to the external part, each moving $\mathcal{M}$ elements from the input-buffer to a new leaf, with no comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os.

By Lemma 7, there are at most $\frac{2D}{\mathcal{M}}$ transfers to the internal part. Each transfer involves finding the $\mathcal{M}$ smallest elements among the elements in the buffers of all roots in all collections. First, for each collection, we perform a virtual pull from the semi-sorted roots. We exclude the access cost for the leaves and treat them as semi-sorted. Applying Lemma 5 for each of the $\mathcal{O}(H) = \mathcal{O}\left(\log_\Delta \frac{I}{\mathcal{M}}\right)$ collections leads to a cost of $\mathcal{O}\left(\mathcal{M} \lg \Delta \log_\Delta \frac{I}{\mathcal{M}}\right) = \mathcal{O}\left(\mathcal{M} \lg \frac{I}{\mathcal{M}}\right)$ comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B} \log_\Delta \frac{I}{\mathcal{M}}\right)$ I/Os. Then among these $\mathcal{O}\left(\mathcal{M} \log_\Delta \frac{I}{\mathcal{M}}\right)$ elements, we find the $\frac{\mathcal{M}}{2}$ smallest elements by applying Corollary 3. These elements are then transferred to the insert-buffer and removed from their respective roots. This may trigger pulls to restore the invariants, the cost of which we exclude here and is accounted for in Lemma 8. These operations do not asymptotically increase the total cost, which is $\mathcal{O}\left(D \lg \frac{I}{\mathcal{M}}\right)$ comparisons and $\mathcal{O}\left(\frac{D}{B} \log_\Delta \frac{I}{\mathcal{M}}\right)$ I/Os. Finally, a transfer to the internal part causes $\mathcal{O}(\mathcal{M})$ comparisons to build the new min-buffer and insert-buffer, which sums up to $\mathcal{O}(D)$ comparisons for all transfers to the internal part. ◀

Proof of Lemma 2. Summarizing the number of comparisons and I/Os performed, the total number of comparisons is

$$\mathcal{O}\Bigg(\underbrace{(I + D \lg \mathcal{M})}_{\text{internal memory}} + \underbrace{\left(I \frac{\lg \Delta}{\Delta} + D \lg \frac{I}{\mathcal{M}}\right)}_{\text{pulls}} + \underbrace{(I + D \lg \Delta)}_{\text{leaves}} + \underbrace{\left(D \lg \frac{I}{\mathcal{M}}\right)}_{\text{transfers}}\Bigg) = \mathcal{O}(I + D \lg I) \ ,$$

and the total number of I/Os is

$$\mathcal{O}\Bigg(\underbrace{\left(\frac{I}{\mathcal{M}} + \frac{DH}{B}\right)}_{\text{pulls}} + \underbrace{\left(\frac{I}{B} + \frac{D}{B}\right)}_{\text{leaves}} + \underbrace{\left(\frac{I}{B} + \frac{D}{B} \log_\Delta \frac{I}{\mathcal{M}}\right)}_{\text{transfers}}\Bigg) = \mathcal{O}\left(\frac{I}{B} + \frac{D}{B} \log_\Delta \frac{I}{\mathcal{M}}\right) \ .$$

Lemma 2 follows from $\mathcal{M} = \Theta(M)$ and $\Delta = \frac{\mathcal{M}}{B}$. ◀

8 Dependence on current priority queue size

In Sections 3–7, I denotes the total number of insertions performed, allowing the current size N to be arbitrarily smaller than I. Since the comparison and I/O bounds of Lemma 2 are dependent on I, the bounds are not a function of the current number of elements N in

the priority queue. To achieve this, we apply the standard technique of *global rebuilding* [28, Chapter 5].

Proof of Theorem 1. Let N_0 denote the size of the priority queue at the last global rebuilding of the priority queue, and let I_0 and D_0 denote the number of insertions and deletions since the last global rebuilding, respectively. We rebuild the priority queue whenever $I_0 + D_0 > \frac{N_0}{2}$. Rebuilding a priority queue containing N elements is performed by scanning over the structure, collecting the N elements into a list, and then repeatedly inserting these into a new empty priority queue using Insert. By Lemma 2, the resulting priority queue is constructed using $\mathcal{O}(N)$ comparisons and $\mathcal{O}\left(\frac{N}{B}\right)$ I/Os (the previous structure can be scanned in $\mathcal{O}\left(\frac{N}{B}\right)$ I/Os, since we argue below that $N = \Theta(N_0)$ and at most $2\frac{N_0+I_0}{\mathcal{M}} = \mathcal{O}\left(\frac{N}{\mathcal{M}}\right)$ buffers have been created in external memory since the last global rebuilding).

Together with Lemma 2, the previous rebuilding and the subsequent I_0 insertions and D_0 deletions, causing a total of $N_0 + I_0$ insertions and D_0 deletions, imply a total comparison cost of $\mathcal{O}(N_0 + I_0 + D_0 \lg(N_0 + I_0)) = \mathcal{O}(N_0 + D_0 \lg N_0)$ and a total I/O cost of $\mathcal{O}\left(\frac{N_0}{B} + \frac{1}{B}\left((N_0 + I_0) + D_0 \log_{M/B} \frac{N_0+I_0}{B}\right)\right) = \mathcal{O}\left(\frac{N_0}{B} + \frac{D_0}{B} \log_{M/B} \frac{N_0}{B}\right)$, since $I_0 \leq \frac{N_0}{2}$. We charge the linear cost to the at least $\frac{N_0}{3}$ priority queue operations between the last two global rebuildings, and the remaining cost to the deletions since the last global rebuilding. While no global rebuilding is triggered, we have $\frac{1}{2}N_0 \leq N \leq \frac{3}{2}N_0$, i.e., we have $N = \Theta(N_0)$ between two global rebuildings. Theorem 1 follows. ◀

9 Conclusion and open problems

The external-memory priority queue presented in this paper supports insertions with asymptotically fewer comparisons and I/Os than previous results. This is particularly beneficial in settings such as event-driven simulations or other incremental computations where execution may terminate before all elements have been extracted from the queue and the cost of insertions dominates. It should be noted that the amortized internal computation time required by our construction is asymptotically the same as the number of comparisons performed (which also holds for the black boxes used by the construction [5, 24, 31]).

The presented data structure is inherently cache-aware. This raises a natural question: does there exist a cache-oblivious priority queue that achieves the same amortized bounds on comparisons and I/Os?

The structure is also highly amortized (e.g., due to global rebuilding, transfers between internal and external memory, recursive pulling, lazy semi-sorted buffers), and a single Insert or DeleteMin operation might require $\Theta(N/B)$ I/Os in the worst-case. Another natural question, is whether it is possible to achieve a solution with worst-case guarantees, e.g., such that a window of B priority queue operations requires worst-case $\mathcal{O}(B \lg N)$ comparisons and $\mathcal{O}\left(\log_{M/B} \frac{N}{B}\right)$ I/Os, while maintaining the improved amortized bounds for Insert.

References

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

2 David P. Dobkin andJ. Ian Munro. Optimal time minimal space selection algorithms. *Journal of the ACM*, 28(3):454–461, 1981. doi:10.1145/322261.322264.

3 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.

4 Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal of Computing*, 36(6):1672–1695, 2007. doi:10.1137/S0097539703428324.

5 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

6 Klaus Brengel, Andreas Crauser, Paolo Ferragina, and Ulrich Meyer. An experimental study of priority queues in external memory. In Jeffrey Scott Vitter and Christos D. Zaroliagis, editors, *Algorithm Engineering, 3rd International Workshop, WAE '99, London, UK, July 19-21, 1999, Proceedings*, volume 1668 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1999. doi:10.1007/3-540-48318-7_27.

7 Gerth Stølting Brodal. A survey on priority queues. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 150–163. Springer, 2013. doi:10.1007/978-3-642-40273-9_11.

8 Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan J. Eidenbenz, and Ricardo Conejo, editors, *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2002. doi:10.1007/3-540-45465-9_37.

9 Gerth Stølting Brodal and Rolf Fagerberg. Funnel heap — A cache oblivious priority queue. In Prosenjit Bose and Pat Morin, editors, *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November 21-23, 2002, Proceedings*, volume 2518 of *Lecture Notes in Computer Science*, pages 219–228. Springer, 2002. doi:10.1007/3-540-36136-7_20.

10 Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In Lawrence L. Larmore and Michel X. Goemans, editors, *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 307–315. ACM, 2003. doi:10.1145/780542.780589.

11 Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. Springer, 2004. doi:10.1007/978-3-540-27810-8_41.

12 Gerth Stølting Brodal and Jyrki Katajainen. Worst-case external-memory priority queues. In Stefan Arnborg and Lars Ivansson, editors, *Algorithm Theory - SWAT '98, 6th Scandinavian Workshop on Algorithm Theory, Stockholm, Sweden, July, 8-10, 1998, Proceedings*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 1998. doi:10.1007/BFB0054359.

13 Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal of Computing*, 7(3):298–319, 1978. doi:10.1137/0207026.

14 Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT 88, 1st Scandinavian Workshop on Algorithm Theory, Halmstad, Sweden, July 5-8, 1988, Proceedings*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1988. doi:10.1007/3-540-19487-8_1.

15 Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In Phillip B. Gibbons and Micah Adler, editors, *SPAA 2004: Proceedings of*

the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain, pages 245–254. ACM, 2004. doi:10.1145/1007912.1007949.

16 James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988. doi:10.1145/50087.50096.

17 Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. Decreasekeys are expensive for external memory priority queues. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1081–1093. ACM, 2017. doi:10.1145/3055399.3055437.

18 R. Fadel, K. V. Jakobsen, Jyrki Katajainen, and Jukka Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999. doi:10.1016/S0304-3975(99)00006-7.

19 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, 1987. doi:10.1145/28869.28874.

20 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814600.

21 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22, 2012. doi:10.1145/2071379.2071383.

22 Bernhard Haeupler, Richard Hladík, Václav Rozhon, Robert E. Tarjan, and Jakub Tetek. Universal optimality of Dijkstra via beyond-worst-case heaps. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 2099–2130. IEEE, 2024. doi:10.1109/FOCS61266.2024.00125.

23 Bernhard Haeupler, Richard Hladík, Václav Rozhon, Robert E. Tarjan, and Jakub Tetek. Bidirectional Dijkstra's algorithm is instance-optimal. In Ioana Oriana Bercea and Rasmus Pagh, editors, *2025 Symposium on Simplicity in Algorithms, SOSA 2025, New Orleans, LA, USA, January 13-15, 2025*, pages 202–215. SIAM, 2025. doi:10.1137/1.9781611978315.16.

24 Nicholas J. A. Harvey and Kevin C. Zatloukal. The post-order heap. In *Proceedings Third International Conference on Fun with Algorithms (FUN 2004), Elba, Italy, May 2004*, 2004. URL: http://people.csail.mit.edu/nickh/Publications/PostOrderHeap/FUN04-PostOrderHeap.pdf.

25 John Iacono, Riko Jacob, and Konstantinos Tsakalidis. External memory priority queues with decrease-key and applications to graph algorithms. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 60:1–60:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ESA.2019.60.

26 Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with decreasekeys. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1331–1343. SIAM, 2019. doi:10.1137/1.9781611975482.81.

27 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.

28 Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. doi:10.1007/BFB0014927.

29 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 263–268. IEEE, 2000. doi:10.1109/IPDPS.2000.845994.

30 Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000. doi:10.1137/S0097539795288246.

31 Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978. doi:10.1145/359460.359478.

32 John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964. doi:10.1145/512274.512284.