# External memory priority queues with decrease-key and applications to graph algorithms

John Iacono*[1], Riko Jacob†[2], and Konstantinos Tsakalidis‡[3]

[1]Department of Computer Science, Université Libre de Bruxelles, Belgium
[2]Computer Science Department, IT University of Copenhagen, Denmark
[3]Department of Computer Science, University of Liverpool, United Kingdom



**Abstract**

We present priority queues in the external memory model with block size $B$ and main memory size $M$ that support on $N$ elements, operation UPDATE (a combination of operations INSERT and DECREASEKEY) in $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ amortized I/Os and operations EXTRACT-MIN and DELETE in $O\left(\left\lceil\frac{M^{\varepsilon}}{B}\log_{\frac{M}{B}}\frac{N}{B}\right\rceil\log_{\frac{M}{B}}\frac{N}{B}\right)$ amortized I/Os, for any real $\varepsilon\in(0,1)$, using $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ blocks. Previous I/O-efficient priority queues either support these operations in $O\left(\frac{1}{B}\log_2\frac{N}{B}\right)$ amortized I/Os [Kumar and Schwabe, SPDP '96] or support only operations INSERT, DELETE and EXTRACTMIN in optimal $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ amortized I/Os, however without supporting DECREASEKEY [Fadel et al., TCS '99].

We also present buffered repository trees that support on a multi-set of $N$ elements, operation INSERT in $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os and operation EXTRACT on $K$ extracted elements in $O\left(M^{\varepsilon}\log_{\frac{M}{B}}\frac{N}{B}+K/B\right)$ amortized I/Os, using $O\left(\frac{N}{B}\right)$ blocks. Previous results achieve $O\left(\frac{1}{B}\log_2\frac{N}{B}\right)$ I/Os and $O\left(\log_2\frac{N}{B}+\frac{K}{B}\right)$ I/Os, respectively [Buchsbaum et al., SODA '00].

Our results imply improved $O\left(\frac{E}{B}\log_{\frac{M}{B}}\frac{E}{B}\right)$ I/Os for single-source shortest paths, depth-first search and breadth-first search algorithms on massive directed dense graphs $(V,E)$ with $E=\Omega\left(V^{1+\varepsilon}\right),\varepsilon>0$ and $V=\Omega\left(M\right)$, which is equal to the I/O-optimal bound for sorting $E$ values in external memory.

*Partially supported by NSF grants CCF-1319648 and CCF-1533564. Supported by Fonds de la Recherche Scientifique-FNRS under Grant no MISU F 6001 1. ulb@johniacono.com

†rikj@itu.dk

‡Partially supported by NSF grants CCF-1319648 and CCF-1533564. K.Tsakalidis@liverpool.ac.uk

# 1 Introduction

Priority queues are fundamental data structures with numerous applications across computer science, most prominently in the design of efficient graph algorithms. They support the following operations on a set of $N$ stored *elements* of the type (*key*, *priority*), where "key" serves as an identifier and "priority" is a value from a total order:

- Insert(element $e$): Insert element $e$ to the priority queue.
- Delete(key $k$): Remove all elements with key $k$ from the priority queue.
- element $e$ = ExtractMin(): Remove and return the element $e$ in the priority queue with the smallest priority.
- DecreaseKey(element $(k, p)$): Given that an element with key $k$ and priority $p'$ is stored in the priority queue, if priority $p < p'$, replace the element's priority $p'$ with $p$.

Operation Update(element $(k, p)$) is a combination of operations Insert and DecreaseKey, such that if the priority queue does not contain any element with key $k$, Insert($(k, p)$) is executed, otherwise DecreaseKey($(k, p)$) is executed.

We study the problem of designing priority queues that support all these operations in external memory. In the *external memory model* (also known as the *I/O model*) [1] the amount of input data is assumed to be much larger than the main memory size $M$. Thus, the data is stored in an external memory device (i.e. disk) that is divided into consecutive blocks of size $B$ elements. Time complexity is measured in terms of *I/O operations* (or *I/Os*), namely block transfers from external to main memory and vice versa, while computation in main memory is considered to be free. Space complexity is measured in the number of blocks occupied by the input data in external memory. Algorithms and data structures in this model are considered *cache-aware*, since they are paremeterized in terms of $M$ and $B$. In contrast, *cache-oblivious* algorithms and data structures [11] are oblivious to both these values, which allows them to be efficient along all levels of a memory hierarchy. I/O-optimally scanning and sorting $x$ consecutive elements in an array are commonly denoted to take $\text{Scan}(x) = O\left(\frac{x}{B}\right)$ I/Os and $\text{Sort}(x) = O\left(\frac{x}{B} \log_{\frac{M}{B}} \frac{x}{B}\right)$ I/Os, respectively [1, 11].

Priority queues are a basic component in several fundamental graph algorithms, including:

- The *single-source shortest paths (SSSP)* algorithm on directed graphs with positively weighted edges, which computes the minimum edge-weight paths from a given source node to all other nodes in the graph.
- The *depth-first search (DFS)* and *breadth-first search (BFS)* algorithms on directed unweighted graphs, which number all nodes of the graph according to a depth-first or a breadth-first exploration traversal starting from a given source node, respectively.

Another necessary component for these algorithms are I/O-efficient *buffered repository trees (BRTs)* [6, 2, 7]. They are used by external memory graph algorithms in order to confirm that a given node has already been visited by the algorithm. This avoids expensive random-access I/Os incurred by internal memory methods. In particular, BRTs support the following operations on a stored multi-set of $N$ (key, value) elements, where "key" serves as an identifier and "value" is from a total order:

- Insert(element $e$): Insert element $e$ to the BRT.
- element $e_i$ = Extract(key $k$): Remove and return all $K$ elements $e_i$ (for $i \in [1, K]$) in the BRT with key $k$.

| | Insert | Delete | ExtractMin | DecreaseKey |
|---|---|---|---|---|
| [10] | $\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}$ | $\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}$ | $\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}$ | – |
| [14] | $\frac{1}{B}\log_2\frac{N}{B}$ | $\frac{1}{B}\log_2\frac{N}{B}$ | $\frac{1}{B}\log_2\frac{N}{B}$ | $\frac{1}{B}\log_2\frac{N}{B}$ |
| [13]* | $\frac{1}{B}\log_{\log N}\frac{N}{B}$ | $\frac{1}{B}\log_{\log N}\frac{N}{B}$ | $\frac{1}{B}\log_{\log N}\frac{N}{B}$ | $\frac{1}{B}\log_{\log N}\frac{N}{B}$ |
| New | $\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}$ | $\lceil\frac{M^\varepsilon}{B}\log_{\frac{M}{B}}\frac{N}{B}\rceil\log_{\frac{M}{B}}\frac{N}{B}$ | $\lceil\frac{M^\varepsilon}{B}\log_{\frac{M}{B}}\frac{N}{B}\rceil\log_{\frac{M}{B}}\frac{N}{B}$ | $\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}$ |
| [2] | $\frac{1}{B}\log_{\frac{M}{B}}N$ | $\frac{1}{B}\log_{\frac{M}{B}}N$ | $\frac{1}{B}\log_{\frac{M}{B}}N$ | – |
| [5, 7] | $\frac{1}{B}\log_2 N$ | $\frac{1}{B}\log_2 N$ | $\frac{1}{B}\log_2 N$ | $\frac{1}{B}\log_2 N$ |

Table 1: Asymptotic amortized I/O-bounds of cache-aware and cache-oblivious priority queue operations (respectively, above and below the horizontal line) on $N$ elements and real $\varepsilon \in (0,1)$. *Expected I/Os.

### 1.1 Previous work

Designing efficient external memory priority queues able to support operation DecreaseKey (or at least operation Update) has been a long-standing open problem [14, 10, 16, 9, 7, 13]. I/O-efficient adaptations of the standard heap data structure [10] or other sorting-based approaches [16], despite achieving optimal base-$(M/B)$ logarithmic amortized I/O-complexity, fail to support operation DecreaseKey. (Nevertheless, we use these priority queues as subroutines in our structure.) Adaptations of the tournament tree data structure support all operations, albeit in not so efficient base-2 logarithmic amortized I/Os [14, 7]. Indeed, in the recent work of Eenberg, Larsen and Yu [9] it is shown that for a sequence of $N$ operations, any external-memory priority queue supporting DecreaseKey must spend max{Insert, Delete, ExtractMin, DecreaseKey} $= \Omega\left(\frac{1}{B}\log_{\log N} B\right)$ amortized I/Os. Randomized priority queues with matching complexity were recently presented by Jiang and Larsen [13].

The BRTs introduced by Buchsbaum et al. [6, Lemma 2.1] and their cache-oblivious counterparts [2] support Insert in $O\left(\frac{1}{B}\log_2 N\right)$ amortized I/Os and Extract on $K$ extracted elements in $O\left(\log_2 N + K/B\right)$ amortized I/Os on a multi-set of $N$ stored elements.

### 1.2 Our contributions

We present I/O-efficient priority queues that support on $N$ stored elements, operation Update in optimal $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ amortized I/Os and operations ExtractMin and Delete in $O\left(\lceil\frac{M^\varepsilon}{B}\log_{\frac{M}{B}}\frac{N}{B}\rceil\log_{\frac{M}{B}}\frac{N}{B}\right)$, for any real $\varepsilon \in (0,1)$. Our priority queues are the first to support operation Update (and thus DecreaseKey) in a cache-aware setting in optimal I/Os, while also I/O-optimally supporting operation Insert. These bounds improve upon previous priority queues supporting DecreaseKey [14], albeit at the expense of suboptimal I/O-efficiency for ExtractMin and Delete (respecting the lower bound of [9] for $M = \Omega\left(B\log_2 N\right)$). See Table 1 for a comparison with previous cache-aware and cache-oblivious I/O-efficient priority queues.

We also present I/O-efficient BRTs that support on a multi-set of $N$ elements, operation Insert in $O\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ amortized I/Os and operation Extract on $K$ extracted elements in $O\left(M^\varepsilon\log_{\frac{M}{B}}\frac{N}{B} + K/B\right)$ amortized I/Os. Previous cache-aware bounds were $O\left(\frac{1}{B}\log_2\frac{N}{B}\right)$ and $O\left(\log_2\frac{N}{B} + \frac{K}{B}\right)$, respectively [6]. Combined with our priority queues, for external memory SSSP, DFS and BFS algorithms on graphs with $V$ nodes and $E$ directed edges, we achieve $O\left(V\frac{M^{\frac{\alpha}{1+\alpha}}}{B}\log^2_{\frac{M}{B}}\frac{E}{B} + V\log_{\frac{M}{B}}\frac{E}{B} + \frac{E}{B}\log_{\frac{M}{B}}\frac{E}{B}\right)$ I/Os. This compares to previous $O\left(\left(V+\frac{E}{B}\right)\log_2 E\right)$ I/Os for directed SSSP [14, 15, 7] and $O\left(\left(V+\frac{E}{B}\right)\log_2\frac{V}{B} + \frac{E}{B}\log_{\frac{M}{B}}\frac{E}{B}\right)$ I/Os for directed DFS and BFS [6, 2]. Our bounds are I/O-optimal for dense graphs with $E = \Omega\left(V^{1+\varepsilon}\right)$ and $V = \Omega\left(M\right)$.

### 1.3 Our approach

The main component of our priority queues is the *x-treap*, a recursive structure inspired by similar cache-oblivious $x$-box [4] and cache-aware hashing data structures [12] that solve the dynamic dictionary problem in external memory (respectively, under predecessor and membership queries on a dynamic set of keys). To solve the priority queue problem, we adapt this recursive scheme to also handle priorities, inspired by the cache-oblivious priority queues of Brodal et al. [5] that support UPDATE, yet in suboptimal I/Os. Here we discuss informally these ideas, the rationale for combining them, and a back-of-the-envelope intuitive, but incomplete analysis. It is hoped that this will provide the intuition to more easily follow the full details in the sequel.

The idea behind the cache-oblivious priority queues of Brodal et al. [5] is simple. The structure has a logarithmic number of levels, where level $i$ has two arrays, or buffers, of size roughly $2^i$. These buffers are called the *front* and *rear* buffers. They contain key-priority pairs or a key-delete message (described later). The idea is that the front buffers are sorted, with everything in the $i$-th front buffer having smaller priorities than everything in the $(i+1)$-th front buffer. The items in the rear buffers do not have this rigorous ordering, but instead must be larger than the items in the rear buffer at the smaller levels. When an UPDATE operation occurs, the key-priority pair gets placed in the first rear buffer; when a EXTRACTMIN operation occurs, the key-priority pair with the smallest priority is removed from the first front buffer. Every time a level-$i$ buffer gets too full or empty relative to its target size of $2^i$, this is fixed by moving things up or down as needed, and moving things from the rear to front buffer if that respects the ordering of items in the front buffer. This resolution of problems is done efficiently using a scan of the affected and neighbouring levels. Thus, looking in a simplified manner at the lifetime of an UPDATEd item, it will be inserted in the smallest rear buffer, be pushed down to larger rear buffers as they overflow, be moved from a rear buffer to a front buffer once it has gone down to a level where its priority is compatible with those in the corresponding front buffer, then moves up from the front buffer to smaller front buffers as they underflow, and is finally removed from the smallest front buffer during an EXTRACTMIN. Thus, during its lifetime, it could be moved from one level to another a total of $O\left(\log_2 \frac{N}{B}\right)$ times at an I/O-cost of $O\left(\frac{1}{B}\right)$ per level, for a total cost of $O\left(\frac{1}{B}\log_2 \frac{N}{B}\right)$ I/Os. One detail is that when an item moves from a rear to a front buffer, we want to make sure that no items in larger levels with the same key and larger priority are ever removed. This is done through special delete messages, which stay in the rear buffers and percolate down, removing any key-priority pairs with the given key that they encounter in their buffer or the corresponding front buffer.

The problem with this approach is that the base-2 logarithm seems unavoidable, with the simple idea of a geometrically increasing buffer size. So here instead we use the more complicated recursion introduced with the cache-oblivious $x$-box [4] structure and also used in the cache-aware hashing data structures [12]. In its simplest form, used for a dictionary, an $x$-box has three buffers: top, middle and bottom (respectively of approximate size $x$, $x^{1.5}$ and $x^2$), as well as $\sqrt{x}$ recursive *upper-level* $\sqrt{x}$-boxes (ordered logically between the top and middle buffers) and $x$ recursive *lower-level* $\sqrt{x}$-boxes (ordered logically between the middle and bottom buffers). Data in each buffer is sorted, and all keys in a given recursive buffer are smaller than all keys in subsequent recursive buffers in the same level (upper or lower). There is no enforced order among keys in different buffers or in a recursive upper- or lower-level $\sqrt{x}$-box. The key feature of this construction is that the top/middle/bottom buffers have the same size as the neighbouring recursive buffers: the top buffer has size $x$, the top buffers of the upper-level recursive $\sqrt{x}$-boxes have total size $x$; the middle buffer, sum of the bottom buffers of the upper-level, and sum of the top buffers of the lower-level recursive structures all have size $x^{1.5}$; the sum of the bottom buffers of the lower-level recursive structures and the bottom buffer both have size $x^2$. Therefore, when for example a top buffer overflows, it can be fixed by moving excess items to the top buffers of the top recursive substructures. In a simplified view with only insertions, as buffers overflow, an item over its lifetime will percolate from the top buffer to the upper-level substructures, to the middle buffer, to the lower-level substructures, and to the bottom buffer, with each overflow handled only using scans. Assuming a base case of size $M$, there will be $O(\log_M N)$ times that an item will move from one buffer to another and an equal number of times that an item will pass through a base case. One major advantage of this recursive approach, is that an item will pass through a small base case not just once at the structure's top, as before, but many times.

We combine these ideas to form the $x$-treap, described at a high level as follows: Everywhere an $x$-box has a buffer, we replace it with front and rear buffers storing key-priority pairs. The order used by $x$-box is imposed on the keys, not the priorities. The order imposed on priorities in the Brodal et al. structure are carried over and imposed on the priorities in different levels of the $x$-treap; this is aided by the fact that the buffers in the $x$-treap form a DAG, thus the buffers where items with a given key can appear, form a natural total order. Hence, this forms a treap-like arrangement where we use the keys for order in one dimension and priorities for order in the other. We use a separate trivial base case structure which is invoked at a size smaller than the memory size; it stores items in no particular order and thus supports fast insertion of items when a neighbouring buffer adds them ($O\left(\frac{1}{B}\right)$), but slow ($O(M^\epsilon)$ amortized) removal of items with small priorities to fix the underflow of a front buffer above. Thus, again considering the typical hypothetical lifetime of an item, it will be inserted at the top in the rear buffer, percolate down $O\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$ levels and base cases at a cost of $O\left(\frac{1}{B}\right)$ amortized each, move over to a front buffer, then percolate up $O\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$ levels at a cost of $O\left(\frac{M^\epsilon}{B}\right)$ amortized each. Thus, the total amortized cost for an item that is eventually removed by an ExtractMin is $O\left(\frac{M^\epsilon}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$.

However, we want the amortized cost for an item that is inserted via Update to be much faster than this, i.e. $O\left(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$. This requires additional observations and tricks. The first is that, unlike Brodal et al., we do not use delete-type messages that percolate down to eliminate items with larger than minimum priority in order to prevent their removal from ExtractMin. Instead, we adopt a much simpler approach, and use a hash table to keep track of all keys that have been removed by an ExtractMin, and when an ExtractMin returns a key that has been seen before, it is discarded and ExtractMin is repeated. The second trick is to simply ensure that each buffer has as most one item with each key (and removes key-priority pairs other than the one with the minimum priority among those with the same key in the buffer). This has the effect that if there are a total of $u$ updates performed on a key before it is removed by an ExtractMin, the total cost will involve up to $O\left(u \log_{\frac{M}{B}} \frac{N}{B}\right)$ percolations down at a cost of $O\left(\frac{1}{B}\right)$, but only $O\left(\log^2_{\frac{M}{B}} \frac{N}{B}\right)$ percolations up at a cost of $O\left(\frac{M^\epsilon}{B}\right)$ amortized each. After the ExtractMin, some items may still remain in the structure and will be discarded when removed by ExtractMin however, due to the no-duplicates-per-level property there will only be $O\left(\log_{\frac{M}{B}} N\right)$ such items (called *ghosts*) which will incur a cost of at most $O\left(\left\lceil \frac{M^\epsilon}{B} \log_{\frac{M}{B}} \frac{N}{B} \right\rceil\right)$ amortized each, where the ceiling accounts for accessing the hash table. Thus the total amortized cost for the lifetime of the $u$ Updates and one ExtractMin involving a single key is $O\left(\frac{u}{B} \log_{\frac{M}{B}} \frac{N}{B} + \left\lceil \frac{M^\epsilon}{B} \log_{\frac{M}{B}} \frac{N}{B} \right\rceil \log_{\frac{M}{B}} \frac{N}{B}\right)$. This cost can be apportioned in the amortized sense by having the ExtractMin cost $O\left(\left\lceil \frac{M^\epsilon}{B} \log_{\frac{M}{B}} \frac{N}{B} \right\rceil \log_{\frac{M}{B}} \frac{N}{B}\right)$ amortized and the updates cost $O\left(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ amortized each, assuming that the treap finishes in an empty state and no item can be Updated after it has been ExtractMin'd.

The details that implement these rough ideas consume the rest of the paper. One complication that eludes the above discussion is that items don't just percolate down and then up; they could move up and down repeatedly and this can be handled through an appropriate potential function. The various layers of complexity needed for the $x$-treap recursion combined with the front/rear buffer idea, various types of over/underflows of buffers, a special base case, having the middle and bottom buffers be of size $x^{1+\frac{\alpha}{2}}$ and $x^{1+\alpha}$ for a suitable parameter $\alpha$ rather than

$x^{1.5}$ and $x^2$ as described above, and a duplicate-catching hash table, result in a complex structure with an involved potential analysis, but that follows naturally from the above high-level description.

## 2 $x$-Treap

Given real parameter $\alpha \in (0, 1]$ and *key range* $[k_{\min}, k_{\max}) \subseteq \mathbb{R}$, an *x-treap D* stores a set of at most $2\,(D.x)^{1+\alpha}$ *elements* $(*, k, p)$ associated with a *key* $k \in [D.k_{\min}, D.k_{\max})$ and a *priority p* from a totally ordered set. $D$ represents a set $D.rep$ of pairs (key, priority), such that a particular key $k$ contained in $D$ is represented to have the smallest priority $p$ of any element with key $k$ stored in $D$, unless an element with key $k$ and a smaller priority has been removed from the structure. In particular, we call the key and priority *represented*, when the pair (key, priority) $\in D.rep$. A *representative* element contains a represented key and its represented priority. More formally, we define:

$$D.rep := \bigcup_{\{k|(k,p)\in D\}} \left\{ \left( k, \min_p \,(k,p) \in D \right) \right\}$$

The proposed representation scheme works under the assumption that a key that is not represented by the structure anymore, cannot become represented again. In other words, a key returned by operation ExtractMin cannot be Inserted to the structure again.

The following *interface operations* are supported:

- Batched-Insert$(D, e_1, e_2, \ldots, e_b)$: For constant $c \in \left(0, \frac{1}{3}\right]$, insert $b \leq c \cdot D.x$ elements $e_1, e_2, \ldots, e_b$ to $D$, given they are key-sorted with keys $e_i.k \in [D.k_{\min}, D.k_{\max})\,, i \in [1, b]$.

  Batched-Insert adds the pairs $(e_i.k, e_i.p)$ to $D.rep$ with key $e_i.k$ that is not contained in $D$ already. Batched-Insert decreases the priority of a represented key $e_i.k$ to $e_i.p$, if its represented priority is larger than $e_i.p$ before the operation. More formally, let $X_{new}$ contain the inserted pairs $(e_i.k, e_i.p)$ with $e_i.k \notin D.rep$. Let $X_{old}$ contain the pairs in $D.rep$ with an inserted key, but with larger priority than the inserted one, and let $X_{dec}$ contain these inserted pairs. After Batched-Insert, a new $x$-treap $D'$ is created where $D'.rep = D.rep \cup X_{new} \cup X_{dec} \backslash X_{old}$.

- Batched-ExtractMin$(D)$: For constant $c \in \left(0, \frac{1}{4}\right]$, remove and return the at most $c \cdot D.x$ elements $(k, p)$ with the smallest priorities in $D$.

  Batched-ExtractMin removes the pairs $X_{\min}$ from $D.rep$ with the at most $c \cdot D.x$ smallest priorities. Let $X_{key}$ contain the pairs in $D$ with keys in $X_{\min}$. After Batched-ExtractMin, a new $x$-treap $D'$ is created where $D'.rep = D.rep \backslash X_{\min} \backslash X_{key}$.

The lemmata in this section prove the following theorem.

**Theorem 1.** *An x-treap D supports* Batched-ExtractMin *in* $O\left(M^{\frac{\alpha}{1+\alpha}} \frac{1+\alpha}{B} \log_M D.x\right)$ *amortized I/Os per element and* Batched-Insert *in* $O\left(\frac{1+\alpha}{B} \log_M D.x\right)$ *amortized I/Os per element, using* $O\left(\frac{(D.x)^{1+\alpha}}{B} \log_M D.x\right)$ *blocks, for any real* $\alpha \in (0, 1]$.

The structure is recursive. The base case is described separately in Subsection 2.3. The base case structure is used when $D.x \leq c' M^{\frac{1}{1+\alpha}}$ (for an appropriately chosen constant $c' > 0$). Thus assuming $D.x > c' M^{\frac{1}{1+\alpha}}$, we define an $x$-treap to contain three *buffers* (which are arrays that store elements) and many $\sqrt{x}$-treaps (called *subtreaps*). Specifically, the *top*, *middle* and *bottom* buffers have *sizes* $D.x, (D.x)^{1+\frac{\alpha}{2}}$ and $(D.x)^{1+\alpha}$, respectively. Each buffer is divided in the middle into a *front* and a *rear buffer*. The subtreaps are divided into the *upper* and the *lower level* that contain at most $\frac{1}{4}\,(D.x)^{\frac{1}{2}}$ and $\frac{1}{4}\,(D.x)^{\frac{1+\alpha}{2}}$ subtreaps, respectively. Let $|b|$ denote

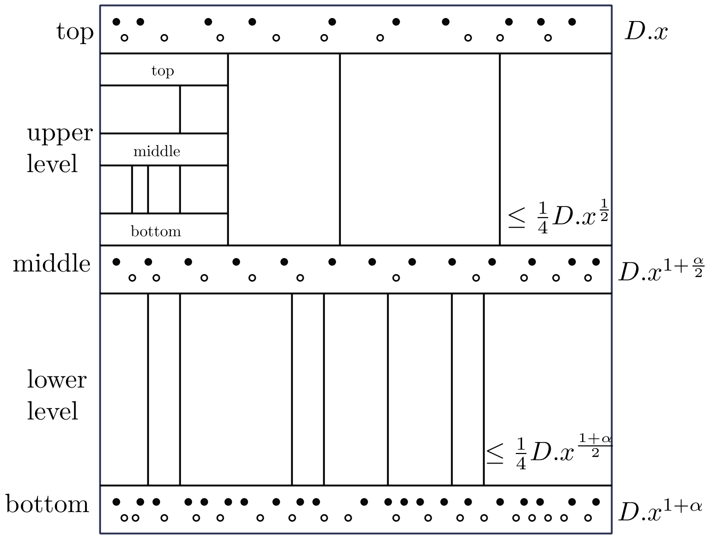


Figure 1: Overview of an $x$-treap $D$ on "key" $\times$ "partial order" space. Black/white dots represent elements in the front/rear buffers, respectively. All buffers are resolved. Buffer sizes and maximum number of subtreaps in a level are shown on the right-hand side.

the *size* of a buffer $b$. We define the *capacity* of an $x$-treap $D$ to be the maximum number of elements it can contain, which is $D.x + \frac{5}{4}(D.x)^{1+\frac{\alpha}{2}} + \frac{5}{4}(D.x)^{1+\alpha} < 2(D.x)^{1+\alpha}$.

We define a partial order ($\preceq$) using the terminology "above/below" among the buffers of an $x$-treap and all of the buffers in recursive subtreaps or base case structures. In this order we have top buffer $\preceq$ upper level recursive subtreaps $\preceq$ middle buffer $\preceq$ lower level recursive subtreaps $\preceq$ bottom buffer.

Along with all buffers of $D$, we also store several additional pieces of bookkeeping information: a counter with the total number of elements stored in $D$ and an index indicating which subtreap is stored in which space in memory.

## 2.1 Invariants

An $x$-treap $D$ maintains the following invariants with respect to every one of its top/middle/bottom buffers $b$. The invariants hold after the execution of each interface operation, but may be violated during the execution. They allow changes to $D$ that do not change $D.rep$.

1. The front and rear buffers of $b$ store elements sorted by key and left-justified.

2. The front buffer's elements' priorities are smaller than the rear buffer's elements' priorities.

3. The front buffer's elements' priorities are smaller than all elements' priorities in buffers below $b$ in $D$.

4. For a top or middle buffer $b$ with key range $[b.k_{\min}, b.k_{\max})$, the $r$ upper or lower subtreaps $D_i, i \in \{1, r\}$, respectively, have distinct key ranges $[D_i.k_{\min}, D_i.k_{\max})$, such that $b.k_{\min} = D_1.k_{\min} < D_1.k_{\max} = D_2.k_{\min} < \ldots < D_r.k_{\max} = b.k_{\max}$.

5. If the middle or bottom buffer $b$ is not empty, then at least one upper or lower subtreap is not empty, respectively.

## 2.2 Auxiliary operations

The operations Batched-Insert and Batched-ExtractMin make use of the following *auxiliary operations*:

- Operation RESOLVE$(D, b)$. We say that a buffer $b$ is *resolved*, when the front and rear buffers contain elements with pairs (key,priority) $(k, p)$, such that $k$ is a represented key, and when the front buffer contains those elements with smallest priorities in the buffer. To resolve $b$, operation RESOLVE assigns to the elements with represented keys, the key's minimum priority stored in $b$. Also, it removes any elements with non-represented keys from $b$. RESOLVE restores Invariant 2 in $b$, when it is temporarily violated by other (interface or auxiliary) operations that call it.

- Operation INITIALIZE$(D, e_1, e_2, \ldots, e_b)$ distributes to a new $x$-treap $D$, $\frac{1}{4}(D.x) \leq b \leq \frac{1}{2}(D.x)^{1+\alpha}$ elements $e_i, i \in [1, b]$ from a temporary array (divided in the middle into a front and a rear array, respecting Invariants 1 and 2).

- Operation FLUSH-UP$(D)$ ensures that the front top buffer of $D$ contains at least $\frac{1}{4}D.x$ elements (unless all buffers of $D$ contains less elements altogether, in which case FLUSH-UP moves them all to the top front buffer of $D$). By Invariants 2 and 3, these are the elements in $D$ with smallest priority.

- Operation FLUSH-DOWN$(D)$ is called by BATCHED-INSERT on an $x$-treap $D$ whose bottom buffer contains between $\frac{1}{2}(D.x)^{1+\alpha}$ and $(D.x)^{1+\alpha}$ elements. It moves to a new temporary array, at least $\frac{1}{6}(D.x)^{1+\alpha}$ and at most $\frac{2}{3}(D.x)^{1+\alpha}$ elements from the bottom buffer of $D$. It ensures that the largest priority elements are removed from $D$.

- Operation SPLIT$(D)$ is called by BATCHED-INSERT on an $x$-treap $D$ that contains between $\frac{1}{2}(D.x)^{1+\alpha}$ and $(D.x)^{1+\alpha}$ elements. It moves to a new temporary (front and rear) array, the at most $\frac{1}{3}(D.x)^{1+\alpha}$ elements with largest keys in $D$.

### 2.2.1 Resolving a buffer

**Algorithm.** Auxiliary operation RESOLVE on a buffer $b$ of an $x$-treap $D$ is called by the auxiliary operation FLUSH-UP and by the interface operation BATCHED-INSERT. It makes use of two temporary auxiliary arrays of size $|b|$. RESOLVE$(D, b)$ implemented as following:

1. Determine the maximum priority $p_{\max}$ in the front buffer (by a scan). Return, if the front buffer is empty.

2. 2-way merge the elements in the front and rear buffers into a temporary array (by simultaneous scans in increasing key-order). Empty the front and rear buffers.

3. Determine the representative elements in the temporary array (by a scan) and write them to a second temporary array (by another scan): specifically for each key, write only the element with the smallest priority to the secondary temporary array.

4. Scan the second temporary array, writing the elements with priority smaller or equal to $p_{\max}$ to the front buffer, and with priority larger than $p_{\max}$ to the rear buffer. Discard the temporary arrays.

5. Update the counter of $D$.

**Correctness.** After calling RESOLVE on a buffer $b$, elements from $b$ are allowed to be moved to other buffers, since Invariants 1 and 2 are maintained. This is because after Steps 2, 3 and 4, the front and rear buffers of $b$ contain a representative element for every represented key in $b$ separated by priority $p_{\max}$ (computed in Step 1). Step 5 accounts for the elements ignored in Step 3. See Figure 2 for an illustration of the operation.

| |
|---|
| (A,8) (B,4) (C,2) (E,6) (F,10) (H,4) |
| (A,3) (B,5) (D,1) (F,12) (G,14) |

| |
|---|
| (A,3) (B,4) (C,2) (D,1) (E,6) (F,10) (H,4) |
| (G,14) |

Figure 2: A buffer before and after operation Resolve (respectively, above and below).

#### 2.2.2 Initializing an $x$-treap

**Algorithm.** Auxiliary operation Initialize is called by the auxiliary operation Flush-Up and by the interface operation Batched-Insert. It allocates an empty $x$-treap $D$ and distributes the $b \in \left[\frac{1}{4}(D.x), \frac{1}{2}(D.x)^{1+\alpha}\right]$ elements $e_i$, $i \in [1, b]$ from a temporary key-sorted array (divided in the middle into a front and a rear array) to the buffers of $D$. Initialize$(D, e_1 \ldots, e_b)$ is implemented as following:

1. Create a new $x$-treap $D$ and move all elements in the temporary rear array to the rear bottom buffer of $D$.
2. Find the $\left(\frac{1}{2}(D.x)^{1+\alpha}\right)$-th smallest priority in the temporary front array (by an order-statistics algorithm [3]) and move all elements in the array with larger priority to the front bottom buffer of $D$.
3. Find the $\left(\frac{1}{2}(D.x)\right)$-th smallest priority in the temporary front array and move all elements in the array with smaller priority to the front top buffer of $D$.
4. Find the $\left(\frac{1}{2}(D.x)\right)$-th smallest priority in the temporary front array and until the maximum number of upper level subtreaps has been reached: Initialize a new upper subtreap with the $\frac{1}{2}(D.x)^{\frac{1+\alpha}{2}}$ key-next elements with smaller priority.
5. Find the $\left(\frac{1}{2}(D.x)^{1+\frac{\alpha}{2}}\right)$-th smallest priority in the temporary front array and move all elements in the array with smaller priority to the front middle buffer of $D$.
6. Find the $\left(\frac{1}{2}(D.x)^{1+\frac{\alpha}{2}}\right)$-th smallest priority in the temporary front array and until the maximum number of lower level subtreaps has been reached: Initialize a new lower subtreap with the $\frac{1}{2}(D.x)^{\frac{1+\alpha}{2}}$ key-next elements with smaller priority.
7. Discard the temporary array and update the counters of $D$.

**Correctness.** Operation Initialize moves the elements from the temporary array to a new $x$-treap in the following sequence: bottom rear buffer, top front buffer, upper subtreaps' front buffers, middle front buffer and lower subtreaps' front buffers, bottom front buffer. The recursive calls in Steps 4 and 6 ensure that the temporary array empties. All invariants are maintained. See Figure 3 for an illustration of the operation.

#### 2.2.3 Flushing up an $x$-treap

**Algorithm.** Auxiliary operation Flush-Up on an $x$-treap $D$ is called only by the interface operation Batched-ExtractMin. It is implemented by means of the recursive subroutine Flush-Up$(D, b)$ that also takes as argument a top or middle buffer $b$ of $D$ and moves to its front buffer, the elements with the (at most) $\frac{1}{4}|b|$ smallest represented priorities among the representative elements stored inside and below $b$ in $D$. The operation makes use of a temporary priority

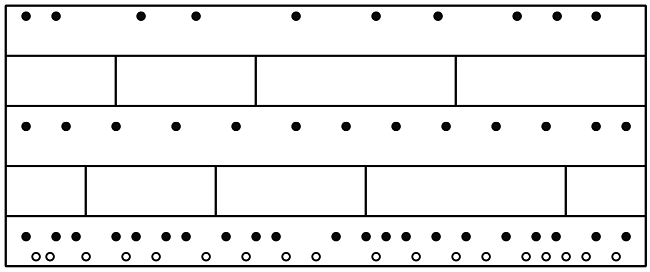

Figure 3: A new $x$-treap after operation Initialize.

queue that supports only operations Insert and ExtractMin [10, 16]. For a bottom buffer $b$, a non-recursive subroutine Flush-Up$(D, b)$ simply calls Resolve on $b$. Flush-Up$(D, b)$ is implemented as following:

1. Resolve $b$ and Flush-Up the top buffers of the subtreaps immediately below $b$.

2. If the front buffer of $b$ contains $k < \frac{1}{4}|b|$ elements: Allocate a temporary array of size $|b|$ and a temporary priority queue $Q$. For every subtreap immediately below $b$: Remove all elements from its front top buffer and Insert them to $Q$ (by simultaneous scans). While the temporary array contains no more than $\frac{1}{4}|b| - k$ elements, do:

   2.1. ExtractMin one element $e$ from $Q$ and write it in the temporary array.

   2.2. If $Q$ contains no more elements from the subtreap $D'$ that contained $e$: Flush-Up the top buffer of $D'$, remove all its elements and Insert them to $Q$.

   2.3. If $Q$ is empty and the temporary buffer contains $k' < \frac{1}{4}|b| - k$ elements: Flush-Up the buffer $b'$ immediately below $b$ in $D$, find the $\left(\frac{1}{4}|b| - k - k'\right)$-th smallest priority in the front buffer of $b'$ (by an external memory order-statistics algorithm [3]) and move its elements with smaller priority to the temporary array. Left-justify $b'$.

3. Sort the elements in the temporary array by key. 2-way merge into the front buffer of $b$, the elements in the front buffer of $b$ and the temporary array (by simultaneous scans in increasing key-order). Discard the temporary array.

4. If $Q$ is not empty: ExtractMin all elements from $Q$ into a new temporary array, sort the array by key, move the elements left-justified back to the front top buffers of the subtreaps they were taken from (by simultaneous scans in increasing key-order), update the subtreaps' counters and discard the array.

5. Discard $Q$, update the counters of $D$ and remove all empty subtreaps immediately below $b$ (i.e. whose counter is 0).

6. If there are no subtreaps immediately below $b$ and the front buffer $b'$ immediately below $b$ is not empty: Find the $\left(\frac{1}{4}|b|^{\frac{1+\alpha}{2}}\right)$-th smallest priority in the front buffer of $b'$ (by an external memory order-statistics algorithm [3]), move the elements with smaller priority to a new temporary front array (by a scan), left-justify the front buffer of $b$ and Initialize a new subtreap with the elements in the array. Discard the temporary array.

**Correctness.** Operation Flush-Up allows for accessing the representative elements with smallest represented priorities in $D$ by only accessing its front top buffer. Invariants 2 and 3 imply that the next-larger represented priorities with respect to the front top buffer's maximum represented priority are stored in the upper subtreaps' front top buffers and in turn the next-larger ones are stored in the front middle buffer. Similarly, this holds between the middle buffer

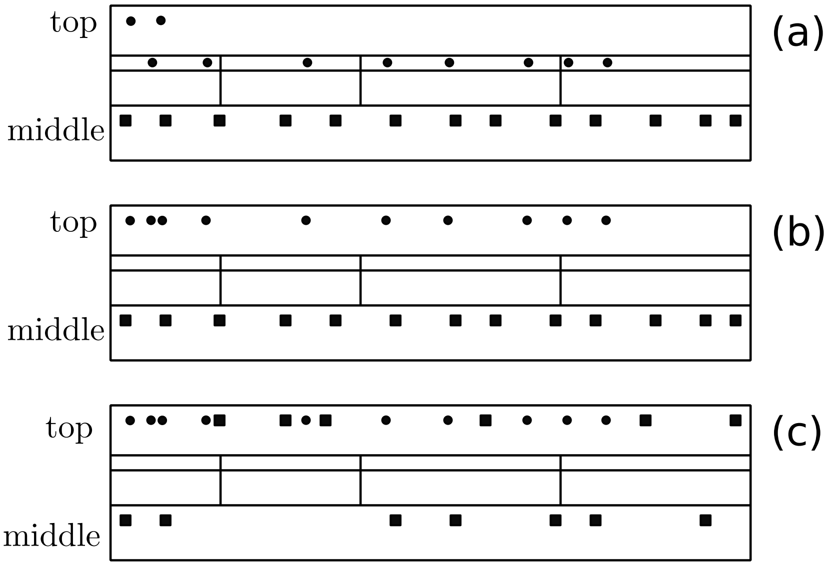


Figure 4: The top and middle buffers and the top upper level buffers, (a) before operation FLUSH-UP, (b) after Step 5 and (c) after Step 6 .

with respect to the lower subtreaps and the bottom buffer. The subroutine FLUSH-UP$(D, b)$ respects this sequence when it moves elements with minimum represented priorities from front buffers to the front buffer of $b$. Specifically, Step 1 ensures that the representative elements in $b$ with priority smaller than the $\left(\frac{1}{4}|b|\right)$-th largest priority in $b$ are stored in its front buffer. It also ensures this recursively for the front top buffers of the subtreaps immediately below $b$. If the front buffer of $b$ contains less than $\frac{1}{4}|b|$ such elements, Step 2 attempts to move into a temporary array enough elements from below $b$ in $D$. The temporary priority queue is used (at Steps 2 and 2.1) to ensure that indeed the smallest-priority representative elements are moved, first from the subtreaps (Step 2.2) and, if not enough elements have been moved, from the buffer immediately below $b$ in $D$ (at Step 2.3). At most two key-sorted runs are created in the temporary array (one from the subtreaps and one from the buffer immediatelly below $b$) which are merged to the front buffer of $b$ by Step 3, while maintaining Invariants 1 and 4. Step 4 revokes the effects of the temporary priority queue, allowing it to be discarded at Step 5, which also accounts for the moved elements. It also removes any empty subtreaps, which may violate Invariant 5. Step 6 restores Invariant 5. See Figure 4 for an illustration of the operation.

#### 2.2.4 Flushing down an $x$-treap

**Algorithm.** Auxiliary operation FLUSH-DOWN on an $x$-treap $D$ is called only by the interface operation BATCHED-INSERT and returns a new temporary key-sorted array with at most $\frac{2}{3}(D.x)^{1+\alpha}$ elements. FLUSH-DOWN$(D)$ is implemented as following:

1. Move all elements from the bottom rear buffer of $D$ the the temporary rear array (by a scan).

2. If Step 1 did not move more than $\left(\frac{1}{6}(D.x)^{1+\alpha}\right)$ elements: Find the $\left(\frac{1}{3}(D.x)^{1+\alpha}\right)$-th smallest priority in the bottom front buffer of $D$ (by an external memory order-statistics algorithm [3]). Move all elements in the bottom front buffer with larger priority to the temporary front array and left-justify the bottom front buffer (by a scan).

3. 2-merge the two runs created by Steps 1 and 2 in the temporary array (by a scan).

4. Update the counter of $D$.

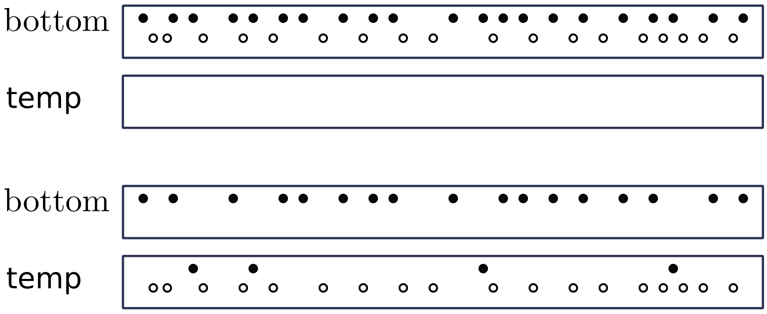


Figure 5: The bottom buffer and temporary array before and after operation FLUSH-DOWN (respectively, above and below).

**Correctness.** Operation FLUSH-DOWN leaves the bottom rear buffer of $D$ empty and the bottom front buffer with at most $\frac{1}{3}(D.x)^{1+\alpha}$ elements. By Invariants 2 and 3, the largest priority elements of $D$ are in the bottom rear buffer and they are removed at Steps 1. However, if they don't account for a constant fraction of $D$'s size, Step 2 removes such a fraction from the bottom front buffer, which contains the next-smaller elements. Invariant 1 is maintained. Step 4 accounts for the removed elements. See Figure 5 for an illustration of the operation.

### 2.2.5 Splitting an $x$-treap

**Algorithm.** Auxiliary operation SPLIT is called only by the interface operation BATCHED-INSERT. It moves to a new temporary key-sorted array (divided in the middle into a front and a rear array) at most $\frac{1}{4}|b_i|$ key-larger elements from all buffers $b_i$ in $D$. SPLIT($D$) is implemented as following:

1. Find the $\left(\frac{1}{4}(D.x)^{1+\alpha}\right)$-th smallest key in the front top/middle/bottom buffer of $D$ (by a scan). (Let this key be $k$ for the front buffer and $k'$ for the middle buffer.) Respectively, move all elements in the front buffers with larger key to three new front auxiliary arrays.
2. Repeat Step 1 with respect to rear buffers.
3. Update the counter of $D$.
4. SPLIT the upper subtreap whose key range contains $k$. SPLIT the lower subtreap whose key range contains $k'$.
5. Merge the elements in all front/rear auxiliary arrays to a new front/rear temporary array, respectively. Discard all auxiliary arrays.

**Correctness.** Operation SPLIT leaves all (front and rear) buffers of $D$ half-full. Since it operates on $x$-treaps that are more than half-full, whose bottom buffers contain a constant fraction of the total number of elements in the $x$-treap, the execution of Steps 1, 2 and 4 moves at most $\frac{1}{2}(D.x)^{1+\alpha}$ elements to the temporary array. Step 3 accounts for the removed elements. All invariants are maintained.

## 2.3 Base case

The $x$-treap is a recursive structure. When the $x$-treap stores few enough elements so that it can be stored in internal memory, we use simple arrays to support the interface operations and operation FLUSH-UP.

**Lemma 1.** *An* $O\left(M^{\frac{1}{1+\alpha}}\right)$*-treap fits in internal memory and supports operation* Batched-Insert *in* $O(1/B)$ *amortized I/Os per element and operations* Batched-ExtractMin *and* Flush-Up *in* Scan$\left(M^{\frac{\alpha}{1+\alpha}}\right)$ *amortized I/Os per element.*

*Proof.* For a universal positive constant $c_0$ and a constant parameter $c' < c_0^{\frac{1}{\alpha}+1}$, we allocate an array of size $\left(c'\left(M^{\frac{1}{1+\alpha}}\right)\right)^{\frac{\alpha}{1+\alpha}} \leq c_0 M$ and divide it in the middle into a front and a rear buffer that store elements and maintain only Invariants 1 and 2. To implement Batched-Insert on at most $\frac{c'}{2}M^{\frac{1}{1+\alpha}}$ elements, we simply add them to the rear buffer and update the counter. This costs $O\left(\frac{M^{\frac{1}{1+\alpha}}}{B}/\frac{1}{2}M^{\frac{1}{1+\alpha}}\right) = O\left(\frac{1}{B}\right)$ I/Os amortized per added element, since we only scan the part of the rear buffer where the elements are being added to. To implement Batched-ExtractMin on at most $\frac{c'}{2}M^{\frac{1}{1+\alpha}}$ extracted elements, we Resolve the array (as implemented for Theorem 1), remove and return all elements in the front buffer, and update the counter. By Lemma 4 (proven later in Subsection 2.5) this costs $O\left(\frac{M}{B}/\frac{1}{2}M^{\frac{1}{1+\alpha}}\right) = O\left(\frac{M^{\frac{\alpha}{1+\alpha}}}{B}\right)$ I/Os amortized per extracted element. Flush-Up is implemented like Batched-ExtractMin with the difference that the returned elements are not removed from the array. ☐

## 2.4 Interface operations

We proceed to the description of the interface operations supported by an $x$-treap.

### 2.4.1 Inserting elements to an $x$-treap

**Algorithm.** Interface operation Batched-Insert on an $x$-treap $D$ is implemented by means of the recursive subroutine Batched-Insert$\big(D, e_1, \ldots, e_{c\cdot|b|}, b\big)$ that also takes as argument a top or middle buffer $b$ of $D$ and inserts $c \cdot |b|$ elements $e_1, \ldots, e_{c\cdot|b|}$ (contained in a temporary array) inside and below $b$ in $D$, for constant $c \in \left(0, \frac{1}{3}\right]$. For a bottom buffer $b$, a non-recursive subroutine Batched-Insert$(D, b)$ simply executes Step 1 below and discards the temporary array. Batched-Insert$\big(D, e_1, \ldots, e_{c\cdot|b|}, b\big)$ is implemented as following:

1. If $D.x > c'M^{\frac{1}{1+\alpha}} + c|b|$:

    1.1. 2-way merge into the temporary array, the elements in the temporary array and in the rear buffer of $b$ (by simultaneous scans in increasing key-order). Resolve $b$ considering the temporary array as the rear buffer of $b$.

    1.2. Implicitly partition the front buffer of $b$ and the temporary array by the key ranges of the subtreaps immediately below $b$. Consider the subtreaps in increasing key-order by reading the index of $D$. For every key range (associated with subtreap $D'$) that contains at least $\frac{1}{3}(D.x)^{\frac{1}{2}}$ elements in either the front buffer of $b$ or the temporary array: While the key range in the front buffer of $b$ and in the temporary array contains at most $\frac{2}{3}(D.x)^{\frac{1}{2}}$ elements, do:

        1.2.1. Find the $\left(\frac{2}{3}(D.x)^{\frac{1}{2}}\right)$-th smallest priority within the key range in the front buffer of $b$ and in the temporary array (by an external memory order-statistics algorithm [3]) and move the elements in the key range with larger priority to a new auxiliary array (by simultaneous scans in increasing key-order).

        1.2.2. If the counter of $D'$ plus the auxiliary array's size does not exceed the capacity of $D'$: Batched-Insert the elements in the auxiliary array to the top buffer of $D'$. Discard the auxiliary array.

1.2.3. Else, if there are fewer than the maximum allowed number of subtreaps in the level immediately below $b$: SPLIT $D'$. Let $k$ be the smallest key in the array returned by SPLIT (determined by a constant number of random accesses to the leftmost elements in the returned front/rear array). Move the elements in the auxiliary array with key smaller than $k$ to a new temporary array (by a scan), BATCHED-INSERT these elements to $D'$ and discard this temporary array. 2-way merge the remaining elements in the auxiliary array into the returned rear array and discard the auxiliary array. INITIALIZE a new subtreap with the elements in the returned array. Discard the returned array.

1.2.4. Else, FLUSH-DOWN all subtreaps immediately below $b$, which writes them to many returned arrays. 2-way merge into a new temporary array, all elements in $b$ and in all returned arrays (by simultaneous scans in increasing key-order). (When the scan on a subtreap's temporary array is over, determine the subtreap with the key-next elements in the level by reading the index of $D$.) BATCHED-INSERT the elements in the new temporary array to the buffer $b'$ immediately below $b$. Discard the new temporary array and all returned arrays.

1.3. Discard the temporary array and update the counter of $D$.

2. Else if $D.x \leq c'M^{\frac{1}{1+\alpha}} + c|b|$: BATCHED-INSERT the elements to the base case structure.

**Correctness.** Operation BATCHED-INSERT accommodates the insertion of at most $\frac{1}{3}|b|$ elements by allocating recursively extra space within $D$. Step 1 considers the recursive structure. Specifically, Step 1.1 allows for moving representative elements from $b$ and inserted elements by resolving $b$ with respect to the temporary array. Step 1.2 identifies the subtreaps immediately below $b$ (repeatedly in increasing key-order) whose associated key range contains too many keys (stored in $b$ and the temporary array, but not in the considered subtreap) and attempts to move the largest-priority elements within this key range into the subtreap. Step 1.2.1 identifies the at most $\frac{1}{3}|b|^{\frac{1}{2}}$ elements to be moved and Step 1.2.2 recursively inserts them to the subtreap. However if the subtreap is full, Step 1.2.3 splits it into two subtreaps with enough space. Nonetheless if the level cannot contain a new subtreap, Step 1.2.4 essentially moves all elements in $b$ and in all subtreaps immediately below $b$, to the buffer $b'$ immediately below $b$. Step 1.3 accounts for the number of inserted elements and the changes in number of subtreaps. Step 2 allows for recursing down to the base case.

### 2.4.2 Extracting minimum-priority elements from an $x$-treap

**Algorithm.** Interface operation BATCHED-EXTRACTMIN on an $x$-treap $D$ is implemented as following:

1. If $D.x > c'M^{\frac{1}{1+\alpha}}$:

   1.1 If the front top buffer contains less than $\frac{1}{4}D.x$ elements: FLUSH-UP the top buffer.

   1.2 Remove and return all the elements $(e_i.k, e_i.p)$ from the front top buffer.

   1.3 Update the counter of $D$.

2. Else if $D.x \leq c'M^{\frac{1}{1+\alpha}}$: BATCHED-EXTRACTMIN the base case structure.

**Correctness.** Operation BATCHED-EXTRACTMIN considers only the top buffer of $D$. Step 1.1 ensures that there are enough minimum-priority representative elements in the front top buffer of $D$ to be extracted by Step 1.2. Step 1.3 accounts from the extracted elements. Step 2 allows for recursing down to the base case. All Invariants are maintained.

### 2.5 Analysis

**Lemma 2.** *An $x$-treap $D$ has $O\left(\log_{\frac{M}{B}} D.x\right)$ levels and occupies $O\left((D.x)^{1+\alpha}\log_{\frac{M}{B}} D.x\right)$ blocks of space.*

*Proof.* We number the levels of the structure sequentially, following the defined "above/below" order from top to bottom, where a base case structure counts for one level. Hence, the total number of levels is given by $L(D.x) = 3 + 2L\left((D.x)^{\frac{1}{2}}\right)$ with $L\left(cM^{\frac{1}{1+\alpha}}\right) = 1$, which solves to the stated bound. Since, RESOLVE leaves in the operated buffer at most one element with a given key, the space bound follows. $\square$

**Lemma 3.** *By the tall-cache assumption, scanning the buffers of an $x$-treap $D$ and randomly accessing $O\left((D.x)^{\frac{1+\alpha}{2}}\right)$ subtreaps takes Scan$\left((D.x)^{1+\alpha}\right)$ I/Os, for any real $\alpha \in (0,1]$.*

*Proof.* We show that $O\left(\frac{(D.x)^{1+\alpha}}{B} + (D.x)^{\frac{1+\alpha}{2}}\right) = O\left(\frac{(D.x)^{1+\alpha}}{B}\right)$. Indeed this holds when $(D.x)^{\frac{1+\alpha}{2}} = O\left(\frac{(D.x)^{1+\alpha}}{B}\right)$. Otherwise $(D.x)^{\frac{1+\alpha}{2}} = \Omega\left(\frac{(D.x)^{1+\alpha}}{B}\right) \Rightarrow (D.x)^{1+\alpha} = O\left(B^2\right)$ and by the tall-cache assumption that $M \geq B^2$, we get that $D.x = O\left(M^{\frac{1}{1+\alpha}}\right) = O(M)$. Hence $D$ fits into main memory and thus randomly accessing its subtreaps incurs no I/Os, meaning that the I/O-complexity is dominated by $O\left(\frac{(D.x)^{1+\alpha}}{B}\right)$. $\square$

#### 2.5.1 Amortization

A buffer $b_i$ at level $i \leq h = O\left(\log_{\frac{M}{B}} D.x\right)$ with current number of elements in the front and rear buffers $b_f, b_r$, respectively, has potential $\Phi(b_i) = \Phi_f(b_i) + \Phi_r(b_i)$, such that (for constants $\varepsilon := \frac{\alpha}{1+\alpha}$ and $c_0 \geq 1$):

- $\Phi_f(b_i) = \begin{cases} 0, & \text{if } \frac{1}{4}|b_i| \leq b_f \leq \frac{1}{3}|b_i|, \\ \frac{c_0}{B}M^\varepsilon \cdot \left(\frac{|b_i|}{4} - b_f\right) \cdot (h-i), & \text{if } b_f < \frac{1}{4}|b_i|, \\ \frac{c_0}{B} \cdot \left(b_f - \frac{|b_i|}{3}\right) \cdot (h-i), & \text{if } b_f > \frac{1}{3}|b_i|, \end{cases}$
- $\Phi_r(b_i) = \left\{ 2\frac{c_0}{B} \cdot \left(b_r - \frac{|b_i|}{2}\right) \cdot (h-i), \quad \text{if } b_r > 0. \right.$

In general, a particular element will be added to a rear buffer and will be moved down the levels of the structure over rear buffers by operation FLUSH-DOWN. A RESOLVE operation will move the element from the rear to the front buffer, if it is a representative element. From this point, it will be moved up the levels over front buffers by operation FLUSH-UP. If it is not representative, it will either get discarded by RESOLVE(when there is an element with the same key and with smaller priority in the same buffer) or it will keep going down the structure. Since RESOLVE leaves only one element per key at the level it operates, $O\left(\log_{\frac{M}{B}} D.x\right)$ elements with the same key (i.e. at most one per level) will remain in the structure after the extraction of the representative element for this key.

The $M^\varepsilon$-factor accounts for the extra cost of FLUSH-UP and BATCHED-EXTRACTMIN, the $(h-i)$-factor allows for moving elements up or down a level by FLUSH-UP and FLUSH-DOWN and the 2-factor accounts for moving elements from the rear to the front buffer.

**Lemma 4.** RESOLVE *on a buffer $b_i$ takes Scan$(|b_i|) + O(1)$ amortized I/Os.*

*Proof.* All steps of operation RESOLVE are implemented by a constant number of scans over buffers of size at most $|b_i|$. Since elements can only be added to the front buffer and only be removed from the rear buffer, the maximum difference in potential occurs, when $\frac{|b_i|}{2} - \frac{|b_i|}{3} = \frac{|b_i|}{6}$

elements are moved from a full rear buffer to a front buffer that contains $\frac{|b_i|}{3}$ elements. We have that:

$$\Delta\Phi(b_i) \leq \Delta\Phi_f(b_i) + \Delta\Phi_r(b_i) \leq \frac{c_0}{B} \cdot \frac{|b_i|}{6} \cdot i - 2\frac{c_0}{B} \cdot \frac{|b_i|}{6} \cdot i \leq -\frac{c_0}{B} \cdot \frac{|b_i|}{6} \cdot i \leq 0,$$

for $|b_i|, i \geq 0$. $\square$

**Lemma 5.** Batched-Insert *on an x-treap D takes* $O\left(\frac{1+\alpha}{B} \log_{\frac{M}{B}} D.x\right)$ *amortized I/Os per element, for any real* $\alpha \in (0, 1]$.

*Proof.* Excluding all recursive calls (to Batched-Insert at Steps 1.2.2, 1.2.3 and 1.2.4 and to Initialize at Step 1.2.3), the worst-case cost of Batched-Insert on a buffer $b_i$ is Scan $\left(|b_i|^{1+\alpha}\right)+ O(1)$ I/Os by Lemmata 3 and 4, by [3] and because this is also the worst-case I/O-cost of Flush-Down and Initialize. The base case (Step 2) charges an extra $O\left(\frac{1}{B}\right)$ I/Os per element by Lemma 1.

In Step 1.2.2, at most $\frac{1}{3}|b_{i+1}|$ elements are moved from a buffer $b_i$ at level $i$ to a buffer $b_{i+1}$ at level $i+1$, where $|b_i| = |b_{i+1}|^{1+\frac{\alpha}{2}}$. The maximum difference in potential occurs when all these elements are moved from a full rear buffer of $b_i$ to a rear buffer of $b_{i+1}$. We have that:

$$\sum_{j=i}^{i+1} \Delta\Phi_r(b_j) \leq -2\frac{c_0}{B}\frac{|b_{i+1}|}{3}(h-i) + 2\frac{c_0}{B}\frac{|b_{i+1}|}{3}(h-i-1) \leq -\frac{2}{3}\frac{c_0}{B}|b_{i+1}| \leq 0$$

for $|b_{i+1}| \geq 0$. If the same scenario occurs between front buffers, we have that:

$$\sum_{j=i}^{i+1} \Delta\Phi_f(b_j) \leq -\frac{1}{3}\frac{c_0}{B}|b_{i+1}| \leq 0$$

for $|b_{i+1}| \geq 0$. Hence $\sum_{j=i}^{i+1} \Delta\Phi(b_j) \leq 0$, given every newly inserted element is charged with an $O(h) = O(\log_M D.x)$ initial amount of potential.

In Step 1.2.3, operation Split removes at most $\frac{1}{2}(D.x)^{1+\alpha}$ from a subtreap $D$, whose bottom buffer $b_i$ we assume to be at level $i$. Operation Initialize will add these elements to a new subtreap, whose bottom buffer is also at level $i$. Without loss of generality, we focus only on these bottom buffers, since $D$ is more than half-full when Split is called on it and since the bottom buffer's size is constant fraction of the subtreap's total size. The maximum difference in potential occurs when $\frac{1}{4}|b_i|$ elements are removed from a full bottom front/rear buffer of $b_i$ and added to an empty bottom front/rear buffer, respectively. We have that:

$$\Delta\Phi_f(b_i) \leq -\frac{c_0}{B}\frac{|b_i|}{6}(h-i) - \frac{c_0}{B}M^{\varepsilon}\frac{|b_i|}{6}(h-i) \leq 0$$

for $|b_{i+1}| \geq 0$.

$$\Delta\Phi_r(b_i) \leq -2\frac{c_0}{B}\frac{|b_i|}{2}(h-i) + 2\frac{c_0}{B}\frac{|b_i|}{2}(h-i) = 0$$

$\Delta\Phi(b_i) \leq 0$, given that subtreaps created by Initialize are charged with an $O\left((D.x)^{1+\alpha} \cdot h\right) = O\left((D.x)^{1+\alpha} \log_M D.x\right)$ initial amount of potential. This extra charge does not exceed by more than a constant factor, the initial potential charged to every newly inserted element, and is amortized over the $O\left((D.x)^{1+\alpha}\right)$ elements that are inserted to the created subtreap.

In Step 1.2.4, operation Flush-Down removes elements from a bottom buffer $b_i$ at level $i$ and inserts them to a middle or bottom buffer $b_{i+1}$ at level $i+1$, where $|b_{i+1}| = |b_i|^{1+\frac{\alpha}{2}}$. The maximum difference in potential occurs when $\frac{1}{2}|b_i|$ elements are removed from a full bottom rear buffer of $b_i$ and added to a rear buffer at level $i+1$. We have that:

$$\sum_{j=i}^{i+1} \Delta\Phi_r(b_j) \leq -2\frac{c_0}{B}\frac{|b_i|}{2}(h-i) + 2\frac{c_0}{B}\frac{|b_i|}{2}(h-i-1) \leq -\frac{c_0}{B}|b_i| \leq 0$$

for $|b_{i+1}| \geq 0$. For the case where Step 2 is executed, in the worst case at most $\frac{1}{6}|b_i|$ elements are removed from a full bottom front buffer of $b_i$. By Invariants 2 and 3, operation Resolve at Step 1.1 of the recursive call to Batched-Insert will move these elements to the middle/bottom front buffer at level $i+1$. We have that:

$$\sum_{j=i}^{i+1} \Delta\Phi_f(b_j) \leq -\frac{1}{6}\frac{c_0}{B}|b_i| \leq 0$$

for $|b_i| \geq 0$. Hence $\sum_{j=i}^{i+1} \Delta\Phi(b_j) \leq 0$. □

**Lemma 6.** Batched-ExtractMin *on an x-treap $D$ takes $O\left(M^{\frac{\alpha}{1+\alpha}}\frac{1+\alpha}{B}\log_{\frac{M}{B}} D.x\right)$ amortized I/Os per element, for any real $\alpha \in (0,1]$.*

*Proof.* The cost of Batched-ExtractMin is dominated by the call to Flush-Up on a buffer $b_i$, where $|b_i| = O(D.x)$. Steps 1.2 (extraction) and 2 (base case) cost an extra $O\left(M^{\frac{1}{1+\alpha}}/B\right)$ amortized I/Os per element, due to the potential's definition and by Lemma 1, respectively.

Excluding all recursive calls (to Flush-Up at Steps 1, 2.2 and 2.3 and to Initialize at Step 6), the worst-case cost of Flush-Up on a buffer $b_i$ is $\text{Sort}(|b_i|) + O(1)$ I/Os. Specifically, Flush-Up executes $O(|b_i|)$ Inserts and ExtractMins to the temporary priority queue (that does not support DecreaseKey) and a constant number of scans and merges. The priority queue operations [16] and Step 3 take $O(\text{Sort}(|b_i|))$ I/Os, which dominates the cost of incurred random accesses and of calls to Resolve (Lemmata 3 and 4, respectively). The I/O-cost is amortized over the $\Theta(|b_i|)$ elements returned by Batched-ExtractMin, resulting in $O\left(\frac{1}{B}\log_{\frac{M}{B}}|b_i|\right)$ amortized I/Os per element.

To prove the negative cost of the recursive calls (Steps 1 and 2.3), it suffices to argue that there is a release in potential when buffers in consecutive levels are being processed. Without loss of generality, we assume that elements are moved from the middle front buffer at level $i$ to the top front buffer at level $i-2$ ($|b_i| = |b_{i-2}|^{1+\frac{\alpha}{2}}$), where the upper level subtreaps at level $i-1$ are base case structures (hence they do not affect the potential). This assumption charges an extra $O\left(\frac{M^{\varepsilon}}{B}\right)$ I/Os per element by Lemma 1. The case between the bottom and middle buffers is analogous. The maximum difference in potential occurs when $\frac{|b_{i-2}|}{4}$ elements are removed from a front middle buffer at level $i$ with less than $\frac{|b_i|}{4}$ elements, and added to an empty front top buffer at level $i-2$. Since the rear buffers do not change, we have that:

$$\sum_{j=i-2}^{i} \Delta\Phi(b_j) \leq \sum_{j=i-2}^{i} (\Delta\Phi_f(b_j) + \Delta\Phi_r(b_j)) \leq \Delta\Phi_f(b_{i-2}) + \Delta\Phi_f(b_i) \leq$$

$$-\frac{c_0}{B}M^{\varepsilon}\frac{|b_i|}{4}(h-i+2) + \frac{c_0}{B}M^{\varepsilon}\frac{|b_i|}{4}(h-i) \leq -\frac{c_0}{B}M^{\varepsilon}\frac{|b_i|}{2} \leq 0,$$

for $|b_i| \geq 0$. □

# 3 Priority queues

Priority queues support operations Update and ExtractMin that are defined similarly to Batched-Insert and Batched-ExtractMin, respectively, but on a *single* element.

### 3.1 Data structure

To support these operations, we compose a priority queue out of its batched counterpart in Theorem 1. The data structure on $N$ elements consists of $1 + \log_{1+\alpha} \log_2 N$ $x$-treaps of doubly increasing size with parameter $\alpha$ being set the same in all of them. Specifically, for $i \in \{0, \log_{1+\alpha} \log_2 N\}$, the $i$-th $x$-treap $D_i$ has $D_i.x = 2^{(1+\alpha)^i}$. We store all keys returned by ExtractMin in a hash table $X$ [12, 8].

For $i \in \{0, \log_{1+\alpha} \log_2 N - 1\}$, we define the top buffer of $D_i$ to be "below" the bottom buffer of $D_{i-1}$ and the bottom buffer of $D_i$ to be "above" the top buffer of $D_{i+1}$. We define the set of represented pairs (key, priority) $rep = \bigcup_{i=0}^{\log_{1+\alpha} \log_2 N} D_i.rep \backslash \{(k,p)|k \in X\}$ and call *represented* the keys and priorities in $rep$. We maintain the invariant that the maximum represented priority in $D_i.rep$ is smaller than the smallest represented priority below it.

### 3.2 Algorithms

To implement Update on a pair (key,priority) $\in rep$, we Batched-Insert the corresponding element to $D_0$. $D_0$ handles single-element batches, since for $i = 0 \Rightarrow x = \Theta(1)$. When $D_i$ reaches capacity (i.e. contains $(D_i.x)^{1+\alpha}$ elements), we call Flush-Down on it, Batched-Insert the elements in the returned temporary array to $D_{i+1}$ and discard the array. This process terminates at the first $x$-treap that can accomodate these elements without reaching capacity.

To implement ExtractMin, we call Batched-ExtractMin to the first $x$-treap $D_i$ with a positive counter, add the extracted elements to the (empty) bottom front buffer of $D_{i-1}$ and repeat this process on $D_{i-1}$, until $D_0$ returns at least one element. If the returned key does not belong to $X$, we insert it. Else, we discard the element and repeat ExtractMin.

To implement Delete of a key, we add the key to $X$.

**Theorem 2.** *There exist priority queues on $N$ elements that support operation* Update *in $O\left(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ amortized I/Os per element, operations* ExtractMin *and* Delete *in amortized $O\left(\lceil \frac{M^{\frac{\alpha}{1+\alpha}}}{B} \log_{\frac{M}{B}} \frac{N}{B} \rceil \log_{\frac{M}{B}} \frac{N}{B}\right)$ I/Os per element, using $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ blocks, for any real $\alpha \in (0, 1]$.*

*Proof.* To each element in $D_i$ that has been Updated since the last time that $D_i$ has undergone a Flush-Down operation, we define an *update potential* of:

$$\frac{1+\alpha}{B} \log_{\frac{M}{B}} \frac{2^{(1+\alpha)^i}}{B} = (1+\alpha)^{i+1} \frac{1}{B \log \frac{M}{B}} - \frac{1+\alpha}{B} \log_{\frac{M}{B}} B$$

When $D_i$ reaches capacity and Flush-Down is called on it, the number of elements that have been Updated in the priority queue is a constant fraction of the $D_i$'s capacity. This is because the only way element could appear in $D_i$ is from:

- The elements that were already in the priority queue at $D_j$ for $j < i$.
- The elements that were newly inserted.
- The elements inserted from $D_j$ for $j > i$ during ExtractMin process. However, these elements will never bring the number in $D_i$ above a constant fraction of the capacity.

Hence, an Insert operation, before any needed Flush-Down operation, increases the update potential by:

$$\sum_{i=0}^{\log_{1+\alpha}\log_2 N}\left((1+\alpha)^{i+1}\frac{1}{B\log\frac{M}{B}}-\frac{1+\alpha}{B}\log_{\frac{M}{B}}B\right)\leq\sum_{i=0}^{\log_{1+\alpha}\log_2 N}(1+\alpha)^{i+1}\frac{1}{B\log\frac{M}{B}}$$
$$=O\left(\frac{\log N}{B\log\frac{M}{B}}\right)$$
$$=O\left(\frac{\log_{\frac{M}{B}}N}{B}\right)$$

To every key $k$ that has been returned by operation ExtractMin and occurs in $d$ elements in $D$, we define a potential of $\Phi(k)=d\cdot\frac{c_d}{B}M^{\varepsilon}\cdot\log_{\frac{M}{B}}D.x$ (for constant $c_d\geq 1$). Insert does not affect this potential by the assumption that an extracted key is not reinserted to the structure.

To $D_i$ we give an *extract-min potential* of:

$$M^{\frac{\alpha}{1+\alpha}}\frac{1+\alpha}{B}\log_{\frac{M}{B}}\frac{2^{(1+\alpha)^i}}{B}=(1+\alpha)^{i+1}\frac{M^{\frac{\alpha}{1+\alpha}}}{B\log\frac{M}{B}}-M^{\frac{\alpha}{1+\alpha}}\frac{1+\alpha}{B}\log_{\frac{M}{B}}B$$

for the number of ExtractMin operations performed since the last time a Batched-ExtractMin operation was performed on $D_i$. When $D_i$ gets empty and ExtractMin is needed to fill it, the number of elements ExtractMin'd from the priority queue since the last time that happened is a constant fraction of $D_i$'s capacity. This is true, since the only way elements could be removed from $D_i$ between Batched-ExtractMins is because $D_i$ reached its capacity from Updates and thus Flush-Down needed to be called on $D_{i-1}$. This, however, only reduces the number of elements to a constant fraction of $D_i$'s capacity.
Hence, an ExtractMin operation, before any needed Flush-Down operation, increases the extract-min potential by:

$$\sum_{i=0}^{\log_{1+\alpha}\log_2 N}\left((1+\alpha)^{i+1}\frac{M^{\frac{\alpha}{1+\alpha}}}{B\log\frac{M}{B}}-M^{\frac{\alpha}{1+\alpha}}\frac{1+\alpha}{B}\log_{\frac{M}{B}}B\right)\leq\sum_{i=0}^{\log_{1+\alpha}\log_2 N}(1+\alpha)^{i+1}\frac{M^{\frac{\alpha}{1+\alpha}}}{B\log\frac{M}{B}}$$
$$=O\left(M^{\frac{\alpha}{1+\alpha}}\frac{\log N}{B\log\frac{M}{B}}\right)$$
$$=O\left(M^{\frac{\alpha}{1+\alpha}}\frac{\log_{\frac{M}{B}}N}{B}\right)$$

Finally, the potential is constructed to exactly pay for the cost of Flush-Down between $x$-treaps.

The worst-case cost of operation Delete is $O(1)$ I/Os [12]. However, its amortized cost includes the potential necessary to remove all the elements with the deleted key from the structure. We introduce an extra "ghost potential" of $O\left(\frac{M^{\frac{\alpha}{1+\alpha}}}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ to every key in $X$ that has been Deleted or ExtractMin'd. The stated I/O-cost for Delete follows from the fact that the first time a key is Deleted or ExtractMin'd from the structure $O\left(\log_{\frac{M}{B}}\frac{N}{B}\right)$ elements in the structure get charged by this potential (by Lemma 2). Whenever an ExtractMin returns an element with ghost potential, this potential is released in order to ExtractMin the next element at no extra amortized cost.

The priority queue may contain an $N$-treap that occupies $O\left(\left(\frac{N}{B}\right)^{1+\alpha}\log_{\frac{M}{B}}\frac{N}{B}\right)$ blocks of space (by Theorem 1). However, with carefull manipulation of the unaddressed empty space, this space usage (and thus of the whole priority queue) can be reduced back to $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ blocks. □

# 4 Buffered repository trees

A *buffered repository tree (BRT)* [6, 2, 7] stores a multi-set of at most $N$ *elements*, each associated with a *key* in the range $[1 \dots k_{\max}]$. It supports the operations Insert and Extract that, respectively, insert a new element to the structure and remove and report all elements in the structure with a given key. To implement a BRT, we make use of the $x$-box [4]. Given positive real $\alpha \leq 1$ and key range $[k_{\min}, k_{\max}) \subseteq \Re$, an *$x$-box* $D$ stores a set of at most $\frac{1}{2}(D.x)^{1+\alpha}$ elements associated with a key $k \in [D.k_{\min}, D.k_{\max})$. An $x$-box supports the following operations:

- Batched-Insert$(D, e_1, e_2, \dots, e_b)$: For constant $c \in \left(0, \frac{1}{2}\right]$, insert $b \leq c \cdot D.x$ elements $e_1, e_2, \dots, e_b$ to $D$, given they are key-sorted with keys $e_i.k \in [D.k_{\min}, D.k_{\max}), i \in [1, b]$.
- Search$(D, \kappa)$: Return pointers to all elements in $D$ with key $\kappa$, given they exist in $D$ and $\kappa \in [D.k_{\min}, D.k_{\max})$.

To implement operation Extract$(D, \kappa)$ that extracts all elements with key $\kappa$ from an $x$-box $D$, we Search$(D, \kappa)$ and remove from $D$ all returned pointed elements.

The BRT on $N$ elements consists of $1 + \log_{1+\alpha} \log_2 N$ $x$-boxes of doubly increasing size with parameter $\alpha$ being set the same in all of them. We obtain the stated bounds by modifying the proof of the $x$-box [4, Theorem 5.1] to account for Lemmata 7 and 8.

**Lemma 7.** *For $D.x = \Omega\left(M^{\frac{1}{1+\alpha}}\right)$, an $x$-box supports operation* Batched-Insert *in amortized $O\left(\frac{1+\alpha}{B} \log_{\frac{M}{B}} \frac{D.x}{B}\right)$ I/Os and operation* Extract *on $K$ extracted elements in amortized $O\left((1+\alpha) \log_{\frac{M}{B}} \frac{D.x}{B} + \frac{K}{B}\right)$ I/Os, using $O\left(\frac{(D.x)^{1+\alpha}}{B}\right)$ blocks of space.*

*Proof.* Regarding Batched-Insert on a cache-aware $x$-box, we obtain $O\left(\frac{1+\alpha}{B} \log_{\frac{M}{B}} \frac{D.x}{B}\right)$ amortized I/Os by modifying the proof of Batched-Insert [4, Theorem 4.1] according the proof of Lemma 5. Specifically, every element is charged $O(1/B)$ amortized I/Os, instead of $O\left(1/B^{\frac{1}{1+\alpha}}\right)$, and the recursion stops when $D.x = O\left(M^{\frac{1}{1+\alpha}}\right)$, instead of $D.x = O\left(B^{\frac{1}{1+\alpha}}\right)$.

Regarding Searching for the first occurrence of a key in a cache-aware $x$-box, we obtain $O\left(\log_{\frac{M}{B}} \frac{D.x}{B}\right)$ amortized I/Os by modifying the proof of Search [4, Lemma 4.1], such that the recursion stops when $D.x = O\left(M^{\frac{1}{1+\alpha}}\right)$, instead of $D.x = O\left(B^{\frac{1}{1+\alpha}}\right)$. To Extract all $K$ occurrences of the searched key, we access them by scanning the $x$-box and by following fractional cascading pointers, which incurs an extra $O(K/B)$ I/Os. □

**Lemma 8.** *An $O\left(M^{\frac{1}{1+\alpha}}\right)$-box fits in internal memory and supports operations* Batched-Insert *in $O(1/B)$ amortized I/Os per element and operation* Extract *on $K$ extracted elements in Scan$\left(M^{\frac{\alpha}{1+\alpha}}\right)$ amortized I/Os per element.*

*Proof.* We allocate an array of size $O(M)$ and implement Batched-Insert by simply appending the inserted element to the array and Extract by scanning the array and removing and returning all occurrences of the searched key. □

**Theorem 3.** *There exist buffered priority trees on a multi-set of $N$ elements and of $K$ extracted elements that support operations* Insert *and* Extract *in amortized $O\left(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ and $O\left(\frac{M^{\frac{\alpha}{1+\alpha}}}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{K}{B}\right)$ I/Os per element, using $O\left(\frac{N}{B}\right)$ blocks, for any real $\alpha \in (0, 1]$.*

## 5 Graph algorithms

### 5.1 Directed single-source shortest paths

**Theorem 4.** *Single source shortest paths on a directed graph with $V$ nodes and $E$ edges can be computed in* $O\left(V\frac{M^{\frac{\alpha}{1+\alpha}}}{B}\log^2_{\frac{M}{B}}\frac{E}{B}+V\log_{\frac{M}{B}}\frac{E}{B}+\frac{E}{B}\log_{\frac{M}{B}}\frac{E}{B}\right)$ *I/Os, for any real* $\alpha\in(0,1]$.

*Proof.* The algorithm of Vitter [15] (described in detail in [7, Lemma 4.1] for the cache-oblivious model) makes use of a priority queue that supports the UPDATE operation and of a BRT on $O(E)$ elements. Specifically, it makes $V$ calls to EXTRACTMIN and $E$ calls to UPDATE on the priority queue and $V$ calls to EXTRACT and $E$ calls to INSERT on the BRT. Hence, we obtain the stated bounds, by using Theorems 2 and 3. □

### 5.2 Directed depth- and breadth-first search

**Theorem 5.** *Depth-first search and breadth-first search numbers can be assigned to a directed graph with $V$ nodes and $E$ edges in* $O\left(V\frac{M^{\frac{\alpha}{1+\alpha}}}{B}\log^2_{\frac{M}{B}}\frac{E}{B}+V\log_{\frac{M}{B}}\frac{E}{B}+\frac{E}{B}\log_{\frac{M}{B}}\frac{E}{B}\right)$ *I/Os, for any real* $\alpha\in(0,1]$.

*Proof.* The algorithm of Buchsbaum et al. [6] makes use of a priority queue and of a BRT on $O(E)$ elements. Specifically, it makes $2V$ calls to EXTRACTMIN and $E$ calls to INSERT on the priority queue and $2V$ calls to EXTRACT and $E$ calls to INSERT on the BRT [6, Theorem 3.1]. Hence, we obtain the stated bounds, by using Theorems 2 and 3. □

## References


[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[2] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan HollandMinkley, and J. Ian Munro. An optimal cacheoblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007.

[3] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, August 1973.

[4] Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1448–1456, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[5] 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*, pages 480–492, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[6] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, pages 859–860, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[7] Rezaul A. Chowdhury and Vijaya Ramachandran. Cache-oblivious buffer heap and cache-efficient computation of shortest paths in graphs. *ACM Trans. Algorithms*, 14(1):1:1–1:33, January 2018.

[8] Alex Conway, Martín Farach-Colton, and Philip Shilane. Optimal Hashing in External Memory. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[9] Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. Decreasekeys are expensive for external memory priority queues. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 1081–1093, New York, NY, USA, 2017. ACM.

[10] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theor. Comput. Sci.*, 220(2):345–362, June 1999.

[11] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[12] John Iacono and Mihai Pătraşcu. Using hashing to solve the dictionary problem. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 570–582, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[13] 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.

[14] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, SPDP '96, pages 169–, Washington, DC, USA, 1996. IEEE Computer Society.

[15] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, June 2001.

[16] Zhewei Wei and Ke Yi. Equivalence between priority queues and sorting in external memory. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014*, pages 830–841, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.