Equivalence between Priority Queues and Sorting in External Memory

Zhewei Wei[1,2] and Ke Yi[3,⋆]

[1] School of Information, Renmin University of China
wzskytop@gmail.com
[2] MADALGO[⋆⋆], Department of Computer Science, Aarhus University, Denmark
zhewei@cs.au.dk
[3] The Hong Kong University of Science and Technology, Hong Kong
yike@cse.ust.hk

Abstract. A priority queue is a fundamental data structure that maintains a dynamic ordered set of keys and supports the followig basic operations: insertion of a key, deletion of a key, and finding the smallest key. The complexity of the priority queue is closely related to that of sorting: A priority queue can be used to implement a sorting algorithm trivially. Thorup [11] proved that the converse is also true in the RAM model. In particular, he designed a priority queue that uses the sorting algorithm as a black box, such that the per-operation cost of the priority queue is asymptotically the same as the per-key cost of sorting. In this paper, we prove an analogous result in the external memory model, showing that priority queues are computationally equivalent to sorting in external memory, under some mild assumptions. The reduction provides a possibility for proving lower bounds for external sorting via showing a lower bound for priority queues.

1 Introduction

The priority queue is an abstract data structure of fundamental importance. A priority queue maintains a set of keys and supports the following operations: insertion of a key, deletion of a key, and findmin, which returns the current minimum key in the priority queue. It is well known that a priority queue can be used to implement a sorting algorithm: we simply insert all keys to be sorted into the priority queue, and then repeatedly delete the minimum key to extract the keys in sorted order. Thorup [11] showed that the converse is also true in the RAM model. In particular, he showed that given a sorting algorithm that sorts N keys in $NS(N)$ time, there is a priority queue that uses the sorting algorithm as a black box, and supports insertion and deletion in $O(S(N))$ time, and findmin in constant time. The reduction uses linear space. The main implication of this reduction is that we can regard the complexity of internal priority queues

⋆ Supported by HKRGC under grant GRF-621413.
⋆⋆ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

as settled, and just focus on establishing the complexity of sorting. Algorithmically, it also gives new priority queue constructions by using the fastest (integer) sorting algorithms currently known: an $O(N \log\log N)$ deterministic algorithm by Han [7] and an $O(N\sqrt{\log\log N})$ randomized one by Han and Thorup [8].

In this paper, we prove an analogous result in the external memory model (the I/O model), showing that priority queues are almost computationally equivalent to sorting in external memory. We design a priority queue that uses the sorting algorithm as a black box, such that the cost of an insertion or deletion (given the key of the element to be deleted) in the priority queue is essentially the same as the per-key I/O cost of the sorting algorithm. The priority queue always has the current minimum key in memory so findmin can be handled without I/O cost. Our priority queue is a non-trivial generalization of Thorup's.

1.1 Our Results

Let us first recall the standard I/O model [1]: The machine consists of an internal memory of size M and an infinitely large external memory. Computation can only be carried out in internal memory. The external memory is divided into blocks of size B, and with one I/O, a block of B keys can be together moved from internal to external memory or vice versa. We measure the complexity of an algorithm by counting the number of I/Os it performs, while internal memory computation is free.

Let $\log^{()}$ denote the nested logarithmic function, i.e., $\log^{(0)} x = x$ and $\log^{(i)} = \log(\log^{(i-1)} x)$. Our main result is stated in the following theorem:

Theorem 1. *Suppose we can sort up to N keys in $NS(N)/B$ I/Os in external memory, where S is a non-decreasing function. Then there exists an external priority queue that uses linear space and supports a sequence of N insertion and deletion operations in $O(\frac{1}{B}\sum_{i\geq 0} S(B\log^{(i)}\frac{N}{B}))$ amortized I/Os per operation. Findmin can be supported without I/O cost. The reduction uses $O(B)$ internal memory and is deterministic.*

The first implication of Theorem 1 is that if the main memory has size $\Omega(B\log^{(c)}\frac{N}{B})$ for any constant c, then our priority queue supports insertion and deletion with $O(S(N)/B)$ amortized I/O cost. This is because $S(N) = 0$ when $N \leq M$. Even if $M = O(B)$, the reduction is still tight as long as the function S grows not too slowly. More precisely, we have the following corollary:

Corollary 1. *For $S(N) = \Omega(2^{\log^* \frac{N}{B}})$, the priority queue supports updates with $O(S(N)/B)$ amortized I/O cost; for $S(N) = o(2^{\log^* \frac{N}{B}})$, the priority queue supports updates with $O(S(N)\log^* \frac{N}{B}/B)$ amortized I/O cost.*

The first part can be verified by plugging $S(N) = 2^{\log^* \frac{N}{B}}$ into Theorem 1 and showing that the $S(B\log^{(i)}\frac{N}{B})$'s decrease exponentially with i. For the second part, we simply relax all the $S(B\log^{(i)}\frac{N}{B})$'s to $S(N)$. Note that $2^{\log^* \frac{N}{B}} = o(\log^{(c)}\frac{N}{B})$ for any constant c, so it is very unlikely that a sorting algorithm could achieve $S(N) = o(2^{\log^* \frac{N}{B}})$. No such algorithm is known, even in the RAM model. Therefore, we can essentially consider our reduction to be tight.

1.2 Related Work

Sorting and priority queues have been well studied in the comparison-based I/O model, in which the keys can only be accessed via comparisons. Aggarwal and Vitter [1] showed that $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os are sufficient and necessary to sort N keys in the comparison-based I/O model. This bound is often referred to as the *sorting bound*. If the comparison constraint is replaced by the weaker indivisibility constraint, there is an $\Omega(\min\{\frac{N}{B}\log_{M/B}\frac{N}{B}, N\})$ lower bound, known as the *permuting bound*. The two bounds are the same when $\frac{N}{B}\log_{M/B}\frac{N}{B} < N$; it is conjectured that for this parameter range, $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$ is still the sorting lower bound even without the indivisibility constraint. For $\frac{N}{B}\log_{M/B}\frac{N}{B} > N$, the current situation in the I/O model is the same as that in the RAM model, that is, the best upper bound is just to use the best RAM algorithm (which has $O(N\log\log N)$ time deterministically or $O(N\sqrt{\log\log N})$ time randomized) naively in external memory with one I/O only accessing one key in external memory, completely wasting the parallelism of the block accesses. There is no lower bound without the indivisibility assumption or when $\frac{N}{B}\log_{M/B}\frac{N}{B} > N$. It has been observed that when the block size is not too small, none of the RAM sorting algorithms works better than the comparison-based one, which makes the situation "cleaner". Thus, a sorting lower bound (without any restrictions) has been considered to be more hopeful in the I/O model (with B not too small) than in the RAM model, and it was posed as a major open problem in [1]. Thus, our result provides a way to approach a sorting lower bound via that of priority queues, while data structure lower bounds have been considered (relatively) easier to obtain than (concrete) algorithm lower bounds (except in restricted computation models), as witnessed by the many recent strong cell probe lower bounds for data structures, such as [10,9] among many others. However, our result does not offer any new bounds for priority queues because we do not know of a better sorting algorithm than the comparison-based ones in the I/O model.

Since a priority queue can be used to sort N keys with N insertion and N deletemin operations, it follows that $\Omega(\frac{1}{B}\log_{M/B}\frac{N}{B})$ is also a lower bound for the amortized I/O cost per operation for any external priority queue, in the comparison-based I/O model. There are many priority queue constructions that achieve this lower bound, such as the buffer tree [2], M/B-ary heaps [5], and array heaps [4]. See the survey [12] for more details. However, they do not use sorting as just a black box, and cannot be improved even if we have a faster external sorting algorithm. Thus they do not give a priority queue-to-sorting reduction. The extra $O(\log_{M/B}\frac{N}{B})$ factor comes from a tree structure with fanout $O(M/B)$ within the priority queue construction. and a key must be moved $\Omega(\log_{M/B}\frac{N}{B})$ times to "bubble up" or "bubble down".

Arge et al. [3] developed a cache-oblivious priority queue that achieves the sorting bound with the tall cache assumption, that is, M is assumed to be of size at least B^2. We note that their structure can serve as a priority queue-to-sorting reduction in the I/O model, by replacing the cache-oblivious sort with a sorting black box. The resulting priority queue supports all operations in $O(\frac{1}{B}\sum_{i\geq 0} S(N^{(2/3)^i})$ amortized I/Os if the sorting algorithm sorts N keys in $NS(N)/B$ I/Os. However, this reduction is not tight for $S(N) = O(\log\log\frac{N}{B})$, and there seems to be no easy way to get rid of the tall cache assumption, even if the algorithm has the knowledge of M and B.

1.3 Reduction in Internal Memory

In this section we describe some high level ideas of Thorup's priority queue in internal memory, which will provide some intuition for our reduction in external memory. Given N keys sorted in ascending order, the priority queue will divide the keys into exponentially increasing subsets called *levels*. The number of levels is $\Theta(\log N)$. The minimum key is contained in the smallest level called the *head*. We employ an atomic heap [6] of size $\Theta(\log N)$ to accomodate insertions before distributing them to corresponding levels. An atomic heap can support updates and predecessor queries in sets of $O(\log^2 N)$ size in constant time, so insertions can be distributed to the corresponding levels in constant time. However, the $O(\log N)$-level structure implies that a key may be moved $O(\log N)$ times in its lifetime. To overcome this, we group the keys in a level into base sets, each of size $\Theta(\log N)$. By moving the pointers to the base sets rather than the base sets themselves, we can move $\Omega(\log N)$ keys in constant time. Base sets are rebalanced by merging and splitting whenever their sizes change by a constant factor, which only adds $O(1)$ to the amortized update cost. We also associate each level with a buffer to accomodate keys before distributing them to the base sets. Finally, the head consists of a single base set, which contains $\Theta(\log N)$ keys. We employ an atomic heap on top of it so that the minimum key can be returned in constant time.

2 Structure

In this section, we describe the structure of our priority queue. In the next section, we show how this structure supports various operations. Finally we analyze the I/O costs of these operations.

Some intuitions. Before diving into technical details, we first give some intuitions on our structure and why it works in the I/O model. We follow Thorup's general framework in our reduction. However, in order to achieve I/O-efficiency, there are several challenges to be addressed. An obvious problem is how to integrate the block size B into the structure. The choice of the parameters appears to be rather important, as we don't know any external structures that are as powerful as atomic heaps, and therefore have to use a delicate recursive structure to get near optimal performance. The second challenge is that the buffer flush and rebalance operations of Thorup's priority queue are not designed to be I/O-efficient. We need to come up with new schemes of maintaining and flushing the buffers in order to achieve I/O efficiency. A third challenge concerns the different ways of handling deletions in internal and external memory. In an internal priority queue, each key is associated with a pointer, so given a deletion we can simply follow the pointer to the key and perform the deletion immediately. In external memory, however, this will incur an extra $\Omega(1)$ I/O cost for a deletion. Deletions are usually supported in a lazy fashion in external memory: we insert a "deleting signal" to the structure, and perform the actual deletions afterwards. However, combining the deleting signal techniques with Thorup's idea turns out to be a non-trivial task; Since we do not have direct access to the base sets, we cannot address the deleting signals when they hit the "lowest level" like buffer tree or any other external priority queues do. As we shall see later, this introduces some more subtle complications, and we have to carefully schedule the operations in order to maintain the shape of the priority queue.

The priority queue consists of multiple layers whose sizes vary from N to cB, where c is some constant to be determined later. The i'th layer from above has size $\Theta(B\log^{(i)}\frac{N}{B})$, for $i \geq 0$, and the priority queue has $O(\log^* N)$ layers. For the sake of simplicity we will refer to a layer by its size. Thus the layers from the largest to the smallest are layer N, layer $B\log\frac{N}{B}$, ..., layer cB. Layer cB is also called the head, and is stored in main memory. Given a layer X, its *upper layer* and *lower layer* are layer $B2^{\frac{X}{B}}$ and layer $B\log\frac{X}{B}$, respectively. We use Ψ_X to denote $B2^{\frac{X}{B}}$ and Φ_X to denote $B\log\frac{X}{B}$. The priority queue maintains the invariant that the keys in layer Φ_X are smaller than the keys in layer X. In particular, the minimum key is always stored in the head and can be accessed without I/O cost.

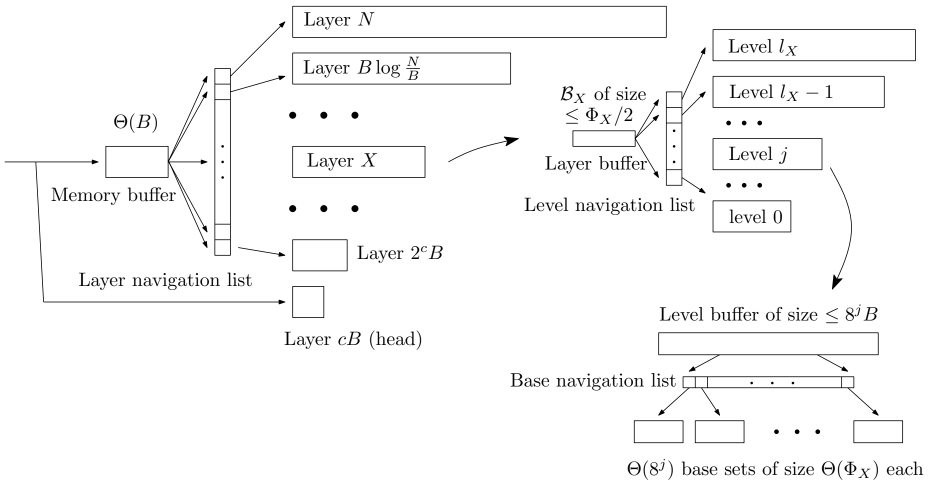

Fig. 1. The components of the priority queue

We maintain a main memory buffer of size $O(B)$ to accommodate incoming insertion and deletion operations. In order to distribute keys in the memory buffer to different layers I/O-efficiently, we maintain a structure called *layer navigation list*. Since this structure will also be used in other components of the priority queue, we define it in a unified way. Suppose we want to distribute the keys in a buffer $\mathcal{B}$ to t sub-structures $S_1, S_2, \dots S_t$. The keys in different sub-structures are sorted relative to each other, that is, the keys in S_i are less or equal to the keys in S_{i+1}. Each sub-structure S_i is associated with a buffer $\mathcal{B}_i$, which accommodates keys transferred from $\mathcal{B}$. The goal is to distribute the keys in $\mathcal{B}$ to each $\mathcal{B}_i$ I/O-efficiently, such that the keys that go to $\mathcal{B}_i$ have values between the minimum keys of S_i and S_{i+1}. A navigation list stores a set of t *representatives*, each representing a sub-structure. The representative of S_i, denoted r_i, is a triple that stores the minimum key of S_i, the number of keys stored in $\mathcal{B}_i$, and a pointer to the last non-full block of the buffer $\mathcal{B}_i$. The representatives are stored consecutively on the disk, and are sorted on the minimum keys. The layer navigation list is built for the $O(\log^* N)$ layers, so it has size $O(\log^* N)$. Please see Figure 1.

Now we will describe the structures inside a layer X except layer cB, which is always in the main memory. First we maintain a *layer buffer* of size $\Phi_X/2$ to store keys flushed from the memory buffer. The main structure of layer X consists of $O(\log \frac{X}{\Phi_X})$ levels with exponentially increasing sizes. The j'th level from the bottom, denoted level j, has size $\Theta(8^j \Phi_X)$. We also keep the invariant that the keys in level j are less or equal to the keys in level $j+1$. We maintain a *level navigation list* of size $\Theta(\log \frac{X}{\Phi_X})$, which represents the $\log \frac{X}{\Phi_X}$ levels. Most keys in level j are stored in $\Theta(8^j)$ disjoint *base sets*, each of size $\Theta(\Phi_X)$. The base sets, from left to right, are sorted relative to each other, but they are not internally sorted. Other than the base sets, there is a *level buffer* of size $8^j B$, which is used to temporarily accommodate keys before distributing them to the base set. We also maintain a *base navigation list* of size $\Theta(8^j)$ for the base sets. Note that we do not impose the level structures on layer cB since it can fit in the main memory. The components of the priority queue are illustrated in Figure 1.

Let l_X denote the top level of layer X. We use $\mathcal{B}_X$ to denote the layer buffer of layer X and $\mathcal{B}_j$ to denote the level buffer of level j when the layer is specified. Our priority queue maintains the following invariants for layer X:

Invariant 1. *The layer buffer $\mathcal{B}_X$ contains at most $\frac{1}{2}\Phi_X$ keys; the level buffer $\mathcal{B}_j$ at layer X contains at most $8^j B$ keys.*

Invariant 2. *The layer buffer $\mathcal{B}_X$ only contains keys between the minimum keys of layer X and its upper layer. The level buffer $\mathcal{B}_j$ only contains keys between the minimum keys of level j and its upper level.*

Invariant 3. *A base set in layer X has size between $\frac{1}{2}\Phi_X$ and $2\Phi_X$; level j of layer X, for $j = 0, 1, \dots, l_X - 1$, has size between $2 \cdot 8^j \Phi_X$ and $6 \cdot 8^j \Phi_X$, and level l_X has size between $2 \cdot 8^{l_X} \Phi_X$ and $40 \cdot 8^{l_X} \Phi_X$.*

Invariant 4. *The head contains at most $2cB$ keys and no delete signal.*

Note that when we talk about the size of a level, we only count the keys in its base sets and exclude the level buffer. The top level has a slightly different size range so that the construction works for any value of X.

We say a layer buffer, a level buffer, a base set, a level or the head overflows if its size exceeds its upper bound in Invariant 1, 3 or 4; we say a base set, a level underflows if its size gets below the lower bound in Invariant 3.

3 Operations

Recall that the priority queue supports three operations: insertion, deletion, and findmin. Since we always maintain the minimum key in the main memory (it is always in the head), the cost of a findmin operation is free. We process deletions in a lazy fashion, that is, when a deletion comes we generate a *delete signal* with the corresponding key and a time stamp, and insert the delete signal to the priority queue. In most cases we treat the delete signals as normal insertions. We only perform the actually delete in the head or during global rebuilding so that the current minimum key is always valid. To ensure linear space usage we perform a global rebuild after every $N/8$ updates.

Our priority queue is implemented by three general operations: *global rebuild*, *flush*, and *rebalance*. A global rebuild operation sorts all keys and processes all delete signals to maintain linear size. A flush operation distributes all keys in a buffer to the buffers of corresponding sub-structures to maintain Invariant 1. A rebalance operation moves keys between two adjacent sub-structures to maintain Invariant 3.

3.1 Global Rebuild

We conduct the first global rebuild when the internal memory buffer is full. Then, after each global rebuild, we set N to be the number of keys in the priority queue, and keep it fixed until the next global rebuild. A global rebuild is triggered whenever layer N (in fact, its top level) becomes unbalanced or the priority queue has received $N/8$ new updates since the last global rebuild. We show that it takes $O(NS(N)/B)$ I/Os to rebuild our priority queue. We first sort all keys in the priority queue and process the delete signals. Then we scan through the remaining keys and divide them into base sets of size Φ_N, except the last base set which may be smaller. This base set is merged to its predecessor if its size is less or equal to $\frac{1}{2}\Phi_N$. The first base set is used to construct the lower layers, and the rest are used to construct layer N. To rebuild the $O(\log \frac{N}{\Phi_N})$ levels of layer N, we scan through the base sets, and take the next $4 \cdot 8^j$ base sets to build level j, for $j = 0, 1, 2, \ldots$. Note that the base navigation list of these $4 \cdot 8^j$ base sets can be constructed when we scan through the keys in the base sets. The level rebuild process stops when we encounter an integer l_N such that the number of remaining base sets is more than $4 \cdot 8^{l_N}$, but less or equal to $4 \cdot (8^{l_N} + 8^{l_N+1}) = 36 \cdot 8^{l_N}$. Then we take these base sets to form the top level of layer N. After the global rebuild, level j has size $4^j\Phi_N$, and the top level l_N has size between $(4 \cdot 8^{l_N} - \frac{1}{2})\Phi_N$ and $(36 \cdot 8^{l_N} + \frac{1}{2})\Phi_N$. For $X = B \log \frac{N}{B}, B \log^{(2)} \frac{N}{B}, \ldots, cB$, layer X are constructed recursively using the same algorithm. All buffers are left empty.

Based on the global rebuild algorithm, the priority queue maintains the following invariant between two global rebuilds:

Invariant 5. *The top level l_X in layer X is determined by the maximum l_X such that*

$$1 + \sum_{j=0}^{l_X} 4 \cdot 8^j \leq \frac{X}{\Phi_X}.$$

The number of layers and the number of levels in each layer will not change between two global rebuilds.

As a result of Invariant 5, we have the following lemma:

Lemma 1. *Suppose the top level in layer X is level l_X. Then l_X is an integer that satisfies the following inequality:*

$$4 \cdot 8^{l_X} \Phi_X \leq X \leq 40 \cdot 8^{l_X} \Phi_X.$$

3.2 Flush

We define the flush operation in a unified way. Suppose we have a buffer $\mathcal{B}$ and k sub-structures $S_1, S_2, \ldots, S_k$. Each S_i is associated with a buffer $\mathcal{B}_i$, and a navigation list L of size k is maintained for the k sub-structures. To flush the buffer $\mathcal{B}$ we first sort the keys in it. Then we scan through the navigation list, and for each representative r_i in L, we read the last non-full block of $\mathcal{B}_i$ to the memory, and fill it with keys in $\mathcal{B}$. When the block is full, we write it back to disk, and allocate a new block. We do so until we encounter a key that is larger than the key in r_{i+1}. Then we update r_i, and advance to r_{i+1}. The I/O cost for a flush is the cost of sorting a buffer of size $|\mathcal{B}|$ plus one I/O for each sub-structure, so we have the following lemma:

Lemma 2. *The I/O cost for flushing keys in buffer $\mathcal{B}$ to k sub-structures is bounded by $O(\frac{|\mathcal{B}|S(|\mathcal{B}|)}{B} + k)$.*

There are three individual flush operations. A *memory flush* distributes keys in the internal memory buffer to $O(\log^* N)$ layer buffers; a *layer flush* on layer X distributes keys in the layer buffer to $O(\log \frac{X}{\Phi})$ level buffers in the layer; and a *level flush* on level j at layer X distributes keys in the level buffer to $\Theta(8^j)$ base sets in the level.

3.3 Rebalance

Rebalancing the base sets. Base rebalance is performed only after a level flush, since this is the only operation that causes a base set to be unbalanced. Consider a level flush in level j of layer X. Suppose the base set A overflows after the flush.

To rebalance A we sort and scan through the keys in it, and split it into base sets of size Φ_X. If the last base set has less than $\frac{1}{2}\Phi_X$ keys we merge it into its predecessor. Note that any base set coming out of a split has between $\frac{3}{2}\Phi_X$ and $\frac{1}{2}\Phi_X$ keys, so it takes at least $\frac{1}{2}\Phi_X$ new updates to any of them before it initiates a new split. Note that after the split we should update the representatives in the base navigation list. This can be done without additional I/Os asymptotically to the level flush operation: We store all new representatives in a temporary list and rebuild the navigation list after all overflowed base sets are rebalanced in level j. A base set never underflows so we do not have a join operation.

Rebalancing the levels. We define two level rebalance operations: *level push* and *level pull.* Consider level j at layer X. When the number of keys in level j (except the top level) gets to more than $6 \cdot 8^j \Phi_X$, a level push operation is performed to move some of its base sets to the upper level. More precisely, we scan through the navigation list of level j to find the first representative r_k such that the number of keys before r_k is larger than $4 \cdot 8^j \Phi_X$. Then we split the navigation list of level j around r_k and attach the second half to the navigation list of level $j+1$. Note that by moving the representatives we also move their corresponding base sets to level $j+1$. By Invariant 3, the number of keys in a base set is at most $2\Phi_X$, so the new level j has size between $4 \cdot 8^j \Phi_X$ and $(4 \cdot 8^j + 2)\Phi_X$. Finally, to maintain Invariant 2 we sort level buffer $\mathcal{B}_j$ and move keys larger than the r_k to the level buffer $\mathcal{B}_{j+1}$.

Conversely, if the number of keys in level j gets below $2 \cdot 8^j \Phi_X$ (except the top level), a level pull operation is performed. We cut a proportion of the navigation list of level $j+1$ and attach it to the navigation list of level j, such that the number of keys in level i becomes between $4 \cdot 8^j \Phi_X$ and $(4 \cdot 8^j - 2)\Phi_X$. We also sort $\mathcal{B}_{j+1}$, the buffer of level $j+1$, and move the corresponding keys to level buffer $\mathcal{B}_j$.

Observe that after a level push/pull, the number of keys in level j is between $(4 \cdot 8^j - 2)\Phi_X$ and $(4 \cdot 8^j + 2)\Phi_X$, so it takes at least $\Omega(8^j \Phi_X)$ new updates before the level needs to be rebalanced again. The main reason that we adopt this level rebalance strategy is that it does not touch all keys in the level; the rebalance only takes place on the base navigation lists and the keys in the level buffers.

Rebalancing the layers. When the top level l_X of layer X becomes unbalanced, we can no longer rebalance it only using the navigation list. Recall that its upper level is level 0 in layer Ψ_X. For simplicity we will refer to the two levels as level l_X and level 0, without specifying their layers. We also define two operations for rebalancing a layer: *layer push* and *layer pull.* A layer push is performed when the layer overflows, that is, the number of keys in level l_X gets more than $40 \cdot 8^{l_X} \Phi_X$. In this case we sort all keys in level l_X and level 0 together, then use the first $4 \cdot 8^{l_X} \Phi_X$ keys to rebuild level l_X and the rest to rebuild level 0. Recall that to rebuild a level we scan through the keys and divide them into base sets of size Φ_X, except the last one which has size between $\frac{1}{2}\Phi_X$ and $\frac{3}{2}\Phi_X$, and then we scan through the keys again to build the base navigation list. Note that the rebuild operation will change the minimum key in layer Ψ_X, so we update the layer navigation list accordingly. Finally we sort the keys in the layer buffer $\mathcal{B}_X$ and the level buffer $\mathcal{B}_{l_X}$, and move the keys larger than the new minimum key of layer Ψ_X to the level buffer $\mathcal{B}_0$.

A layer pull operation is performed when the layer underflows, that is, there are less than $2 \cdot 8^{l_X}\Phi_X$ keys in level l_X. A layer pull proceeds in the same way as a layer push does, except for the last step. Here we sort the layer buffer $\mathcal{B}_{\Psi_X}$ and the level buffer $\mathcal{B}_0$ and move the keys smaller than the new minimum key to the level buffer $\mathcal{B}_{l_X}$. After a layer push or pull, the number of keys in level l_X is $4 \cdot 8^{l_X}\Phi_X$. By lemma 1, we have $40 \cdot 8^{l_X}\Phi_X \geq X$, so it takes at least $2 \cdot 8^{l_X}\Phi_X = \Omega(X)$ new updates to layer X before we initiate a new push or a pull again.

Note that since we do not impose the level structure on the head layer cB, we need to design the layer push and layer pull operations specifically for it. A layer push is performed when the number of keys in the head gets to more than $2cB$. We sort all keys in it and level 0 of layer Ψ_{cB}, and use the first cB keys to rebuild the head and the rest to rebuild level 0. A layer pull is performed when the head becomes empty. The operation processes in the same way as a layer push does, except that after rebuilding both levels, we sort the layer buffer $\mathcal{B}_{\Psi_{cB}}$ and the level buffer $\mathcal{B}_0$ together, and move the keys smaller than the new minimum key of layer $\mathcal{B}_{\Psi_{cB}}$ to the head.

3.4 Scheduling Flush and Rebalance Operations

A key component in our priority queue construction is a schedule of the flush and rebalance operations. This is mainly due to the introduction of the deleting signals. Since we are only able to process deleting signals in the head, an update may cause the priority queue to shrink and expand multiple times. In order to maintain the shape of the priority queue, we need to schedule the operations delicately. When a new update comes, we insert it to the head if it is smaller than the maximum key in the head, and to the memory buffer if otherwise.If a delete signal is inserted to the head we process it so that invariant 4 is maintained. Whenever the memory buffer overflows or the head becomes empty or overflowed we start to update the priority queue. This process is divided into three stages: the flush stage, the push stage, and the pull stage. In the flush stage we flush all overflowed buffers and rebalance all unbalanced base sets; in the push stage we use push operations to rebalance all overflowed layers and levels. We treat delete signals as insertions in the flush stage and the push stage. In the pull stage we deal with delete signals and use pull operations to rebalance all underflowed layers and levels.

In the flush stage, we initialize a queue Q_o to keep track of all overflowed buffers and a doubly linked list L_o to keep track of all overflowed levels. The buffers are flushed in a BFS fashion. First we flush the memory buffer into $O(\log^* N)$ layer buffers. After flushing the memory buffer, we insert the representatives of the overflowed layer buffers into Q_o, from bottom to top. We also check whether the head overflows after the memory flush. If so, we insert its representatives to the beginning of L_o. Then we start to flush the layer buffers in Q_o. Again, when flushing a layer buffer we insert the representatives of the overflowed level buffers to Q_o from bottom to top. After all layer buffers are flushed, we begin to flush level buffers in Q_o. After each level flush, we rebalance all unbalanced base sets in this level, and if the level overflows we add the representative of this level to the end of L_o. Note that the representatives in L_o are sorted on the minimum keys of the levels.

After all overflowed level buffers are flushed, we enter the push stage and start to rebalance levels in L_o in a bottom-up fashion. In each step, we take out the first level in L_o (which is also the current lowest overflowed level) and rebalance it. Suppose this level is level j of layer X. If it is not the top level or the head layer we perform a level push; otherwise we perform a layer push. Then we delete the representative of this level from L_o. A level push may cause the level buffer of level $j+1$ to be overflowed, in which case we flush it and rebalance the overflowed base sets. Then we check whether level $j+1$ overflows. If so, we insert the representative of level $j+1$ to the head of L_o (unless it is already at the beginning of L_o) and perform a level push on level $j+1$. Otherwise we take out a new level in L_o and continue the process. When the top level of layer N becomes unbalanced we simply perform a global rebuild.

After rebalancing all levels, we enter the pull stage and start to process the delete signals. This is done as follows. We first process all delete signals in the head. If the head becomes empty we perform a layer pull to get more keys into the head. This may cause higher levels or layers to underflow, and we keep performing level pulls and layer pulls until all levels and layers are balanced. Consider a level pull or layer pull on level j of layer X. After the level pull or layer pull the level buffer $\mathcal{B}_j$ may overflow. If so, we flush it and rebalance the base sets when necessary. Note that this may cause the size of level j to grow, but it will not overflow, as we will show later, so that we do not need push operations in the pull stage. After all levels and layers are balanced, we process the delete signals in the head again. We repeat the pull process until there are no delete signals left in the head and the head is non-empty.

4 Correctness and I/O Cost Analysis

We note that with inappropriate scheduling, it is possible that the pull stage fails due to lack of keys or overflows in some levels. The following two lemmas guarantee that in our scheduling, the pull stage will succeed.

Lemma 3. *When we perform a level pull on level j, there are enough keys in level $j+1$ to rebalance level j; When we perform a layer pull on layer X, there are enough keys in level 0 of layer Ψ_X to rebalance level l_X.*

Lemma 4. *A level or a layer never overflows in the pull stage.*

The following lemma states that the I/O cost for each update is very close to $S(N)/B$, which directly implies Theorem 1.

Lemma 5. *The amortized I/O cost per update for the priority queue is bounded by $O(\frac{1}{B}\sum_{i=0} S(B\log^{(i)}\frac{N}{B}))$.*

Due to space limitations, we omit the proofs of above lemmas from this extended abstract; all missing proofs can be found in the full version of the paper [13].

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM 31(9), 1116–1127 (1988)
2. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica 37(1), 1–24 (2003)
3. Arge, L., Bender, M., Demaine, E., Holland-Minkley, B., Munro, J.: Cache-oblivious priority queue and graph algorithm applications. In: Proc. ACM Symposium on Theory of Computing, pp. 268–276. ACM (2002)
4. Brodal, G., Katajainen, J.: Worst-case efficient external-memory priority queues. In: Proc. Scandinavian Workshop on Algorithms Theory, pp. 107–118 (1998)
5. Fadel, R., Jakobsen, K., Katajainen, J., Teuhola, J.: Heaps and heapsort on secondary storage. Theoretical Computer Science 220(2), 345–362 (1999)
6. Fredman, F.W., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In: Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci., pp. 719–725 (1990)
7. Han, Y.: Deterministic sorting in $o(n \log \log n)$ time and linear space. Journal of Algorithms 50(1), 96–105 (2004)
8. Han, Y., Thorup, M.: Integer sorting in $o(n\sqrt{\log \log n})$ expected time and linear space. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 135–144. IEEE (2002)
9. Larsen, K.G.: The cell probe complexity of dynamic range counting. In: Proc. ACM Symposium on Theory of Computing (2012)
10. Pǎtraşcu, M.: Unifying the landscape of cell-probe lower bounds. SIAM Journal on Computing 40(3) (2011)
11. Thorup, M.: Equivalence between priority queues and sorting. Journal of the ACM 54(6), 28 (2007)
12. Vitter, J.: External memory algorithms and data structures: Dealing with massive data. ACM Computing Surveys 33(2), 209–271 (2001)
13. Wei, Z., Yi, K.: Equivalence between Priority Queues and Sorting in External Memory. arXiv: 1207.4383 (2014)