Quickheaps: Simple, Efficient, and Cache-Oblivious *

Gonzalo Navarro and Rodrigo Paredes

Dept. of Computer Science, University of Chile.
{gnavarro,raparede}@dcc.uchile.cl

Abstract. We present the *Quickheap*, a simple and efficient data structure for implementing priority queues in main and secondary memory. *Quickheaps* are comparable with classical binary heaps in simplicity, but are more cache-friendly. This makes them an excellent alternative for a secondary memory implementation. We show that the average amortized CPU cost per operation over a *Quickheap* of m elements is $O(\log m)$, and this translates into $O((1/B)\log(m/M))$ I/O cost with block size B, in a cache-oblivious fashion. Our experimental results show that *Quickheaps* are very competitive with the best alternative external memory heaps.

1 Introduction

A *priority queue* is a data structure which allows maintaining a set of elements in a partially ordered way. Assume the elements are of the form $(key, item)$, where key is the priority. Let us focus on min-priority queues, which support the following basic operations: **insert**$(key, item)$, which inserts element $(key, item)$ in the queue; **findMin**(), which returns an element of the queue with lowest key value; and **extractMin**(), which in addition removes that lowest key value element.

The set of operations can be extended to construct a priority queue from a given array A (**heapify**), increment or decrement the key of an element in the queue (**increaseKey** and **decreaseKey**, respectively), answer whether an arbitrary element belongs to the queue (**find**), delete an arbitrary element from the queue (**delete**), and so on.

Inspired by the IncrementalQuicksort algorithm (**IQS**) [13], we develop a novel data structure for implementing priority queues, coined *Quickheaps*. *Quickheaps* enable efficient element insertion, minimum extraction, deletion of arbitrary elements and modification of the priority of elements within the heap. They are as simple to implement as classical binary heaps, and require only $O(\log m)$ extra integers for a queue of m elements. Furthermore, they exhibit a local access pattern, which makes them excellent alternatives for a secondary memory implementation.

Interestingly, our algorithms are unaware of the disk transfers, so the result is cache-oblivious. Cache obliviousness [9, 5] means that the algorithm is designed for the RAM model but analyzed under the I/O model, assuming an optimal offline page replacement strategy. Cache-oblivious algorithms for secondary memory are not only easier to program than their cache-aware counterparts, but they adapt better to arbitrary memory hierarchies.

We prove that the average amortized complexity of the basic operations is $O(\log m)$ CPU time and $O((1/B)\log(m/M))$ I/O time, being B the disk block size, if $M = \Omega(B \log m)$. This is close to the lower bound $O((1/B)\log_{M/B}(m/B))$, where M is the main memory size, and matches some cache-oblivious lower bounds for sorting [5].

We experimentally compare *Quickheaps* with state-of-the-art external priority queue implementations, showing that they are extremely competitive.

For space limitations, we do not give results on operations **delete**, **increaseKey** and **decreaseKey**, which can be included in *Quickheaps* without altering our complexity results.

* This work has been funded by a grant of Yahoo! Research Latin America.

2 Related Work

2.1 Priority Queues

The classical implementation of a priority queue uses a binary heap [16]. It allows operations **insert**, **findMin**, **extractMin**, and **heapify**. Other operations such as **delete**, **decreaseKey** and **increaseKey** can be added if we have a dictionary to know the position of the element to modify. The associated algorithms can be found in most textbooks [7]. There are dozens of other priority queue implementations described in the literature.

Many classical data structures have been adapted to work efficiently on secondary memory [15], and priority queues are not an exception. Some examples are *buffer trees* [2, 11], *M/B-ary heaps* [12, 8], and *Array Heaps* [6], which achieve the lower bound of $O((1/B)\log_{M/B}(m/B))$ amortized I/Os per operation [15]. Those structures, however, are rather complex to implement and heavyweight in practice (in extra space and time) [4]. Other techniques are simple but do not perform so well (in theory or in practice), for example those that use B-trees [3].

A practical comparison of existing secondary memory priority queues was carried out by Brengel et al. [4], where in addition they adapt two-level radix heaps [1] to secondary memory (*R-Heaps*), and also simplify *Array-Heaps* [6]. The latter stays optimal in the amortized sense and becomes simple to implement. The experiments in [4] show that *R-Heaps* and *Array-Heaps* are by far the best choices for secondary memory. In the same issue, Sanders introduced *sequence heaps* [14], which can be seen as a simplification of the improved *Array-Heaps* of [4]. Sanders reports that *sequence heaps* are faster than the improved *Array-Heaps*, yet the experiments only consider caching in main memory.

2.2 Incremental Sorting

The incremental sorting problem can be stated as follows: Given a set A of m numbers, output the elements of A from smallest to largest, so that the process can be stopped after k elements have been output, for any k that is unknown to the algorithm. In [13] we introduced the IncrementalQuicksort algorithm, which can solve this problem in $O(m + k \log k)$ optimal expected time, and performs better in practice than previous approaches.

To output the k smallest elements, **IQS** calls Quickselect [10] to find the smallest element of arrays $A[0, m-1]$, $A[1, m-1]$, ..., $A[k-1, m-1]$. This leaves the k smallest elements sorted in $A[0, k-1]$. **IQS** avoids the $O(kn)$ complexity by reusing the work across calls to Quickselect.

Note that when we call Quickselect on $A[1, m-1]$, a decreasing sequence of pivots has already been used to partially sort A in the previous call on $A[0, m-1]$. **IQS** uses this sequence to reuse previous work: It uses a stack S of decreasing pivot positions that are relevant for the next calls to Quickselect. Fig. 1 shows how **IQS** searches for the smallest element (12) of an array by using a stack initialized with a single value $m = 16$. To find the next minimum, we first check whether p, the top value in S, is the index of the element sought, in which case we pop it and return $A[p]$. Otherwise, because of previous partitionings, it holds that elements in $A[1, p-1]$ are smaller than all the rest, so we run Quickselect on that portion of the array, pushing new pivots into S. As can be seen in Fig. 1, the second minimum (18) is the pivot on the top of S, so we pop it and return $A[1]$. Later, to extract the third element (whose index is 2), it is enough to work on the first chunk ($\{29, 25\}$), which is the chunk of elements in $A[2, S.\mathbf{top}()-1] = A[2, 3]$. Fig. 2 shows algorithm **IQS**.

The next lemma shows that it is correct to search for the minimum just within $A[i, S.\mathbf{top}()-1]$, from which the correctness of **IQS** immediately follows.

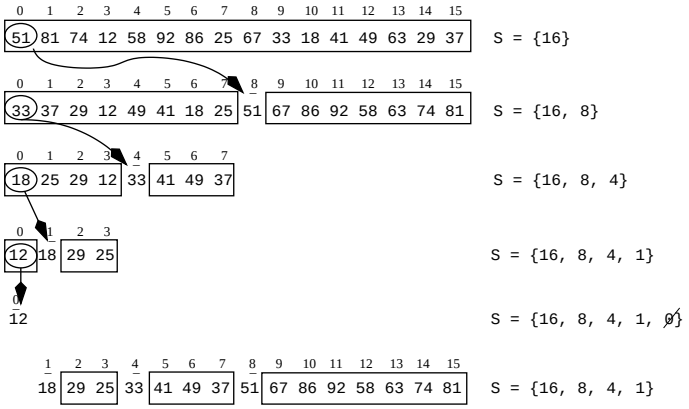

Fig. 1. Example of how **IQS** finds the first element of an array. Each line corresponds to a new partition of a subarray. Note that all the pivot positions are stored in stack S. In the example we use the first element as the pivot but it could be any other element of the first partition. The bottom line shows the array with the three partitions generated by the first call to **IQS**, and the pivot positions stored in S.

IQS (Set A, Index idx, Stack S)
1. **If** $idx = S.\mathbf{top}()$ **Then** $S.\mathbf{pop}()$, **Return** $A[idx]$
2. $pidx \leftarrow \mathbf{random}[idx,\ S.\mathbf{top}()-1]$
3. $pidx' \leftarrow \mathbf{partition}(A,\ A[pidx],\ idx,\ S.\mathbf{top}()-1)$
4. $S.\mathbf{push}(pidx')$
5. **Return IQS**$(A,\ idx,\ S)$

Fig. 2. Algorithm Incremental Quicksort (**IQS**). Stack S is initialized as $S \leftarrow \{|A|\}$. Both S and A are modified and rearranged during the algorithm. Note that the search range is limited to the array segment $A[idx, S.\mathbf{top}()-1]$. Procedure **partition** returns the position of pivot $A[pidx]$ after the partition completes. Note that the tail recursion can be easily removed.

Lemma 1 (pivot invariant [13]) *After i minima have been obtained in $A[0, i-1]$, (1) the pivot indices in S are decreasing bottom to top, (2) for each pivot position $p \neq m$ in S, $A[p]$ is not smaller than any element in $A[i, p-1]$ and not larger than any element in $A[p+1, m-1]$.*

3 Quickheaps

Fig. 3 shows the last line of Fig. 1, where pivots are enclosed in ovals, and we have added an extra ∞ mark signaling a fictitious pivot at the end of the array. By virtue of the pivot invariant, we see the following structure in the array: If we read the array from right to left, we start with a pivot (the fictitious pivot ∞ at position 16) and at its left side there is a chunk of elements smaller than it. Next, we have another pivot (pivot 51 at position 8) and another chunk, and so on, until we reach the last pivot (pivot 18 at position 1) and a last chunk (in this case, without elements).

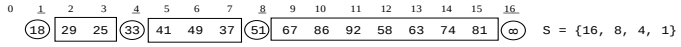

Fig. 3. Last line of Fig. 1.

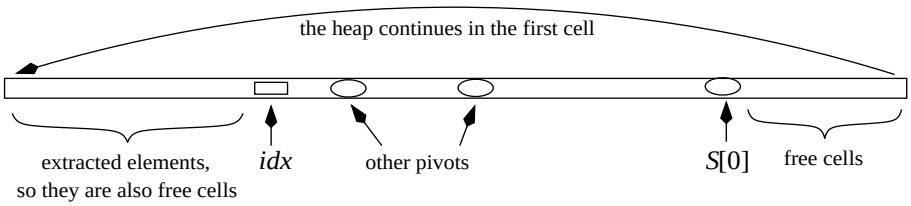

Fig. 4. A quickheap example. The quickheap is placed over an array *heap* of size *capacity*. The quickheap starts at cell *idx*, there are three pivots, and the last cell of the heap is marked by the fictitious pivot $S[0]$. There are some cells after $S[0]$, which are free cells to store new elements. There are also free cells that correspond to extracted elements, which will be used when the quickheap turns around the circular array.

This resembles a heap structure, in the sense that objects in the array are partially ordered. From now on we explain how to implement a min-priority queue, which we call *Quickheap* (**QH**), over an array processed with algorithm **IQS**. Naturally, we can symmetrically obtain a max-quickheap.

3.1 Data Structures for Quickheaps

To implement a quickheap we need the following structures:

1. An array *heap* to store elements. In Fig. 3, *heap* is $\{18, 29, \ldots, 81, \infty\}$.
2. A stack S to store the positions of pivots partitioning *heap*. Thus $S[0]$ is the fictitious pivot ∞, and $S.\mathbf{top}()$ is the smallest pivot. In Fig. 3, S is $\{16, 8, 4, 1\}$.
3. An integer *idx* to indicate the first cell of the quickheap. In Fig. 3, $idx = 1$.
4. An integer *capacity* to indicate the size of *heap*. As we need a cell for the fictitious pivot ∞ we can store up to $capacity - 1$ elements in the quickheap.

Fig. 4 illustrates the structure. We add elements at the tail of the quickheap (the cell $heap[S[0] \bmod capacity]$), and perform min-extractions from the head of the quickheap (the cell $heap[idx \bmod capacity]$). So, the quickheap *slides* from left to right over array *heap* as the operation progresses. In order to handle arbitrarily long sequences of insertions and deletions, we need to use *heap* as a circular array. So, we slightly modify **IQS** so as to take into account that an object whose position is *pos* is actually located at cell *pos* mod *capacity* of the circular array *heap*.

3.2 Quickheap Operations

We explain now how to implement the basic quickheap operations. We omit the expression mod *capacity* in order to simplify the reading, but keep it in the pseudocodes given in Fig. 5.

Creation of Empty Quickheaps. We create the array *heap* of size *capacity* with no elements, and initialize both $S = \{0\}$ and $idx = 0$. The value of *capacity* must be sufficient to store simultaneously all the elements we need in the heap. Note that it is not necessary to actually place the ∞ mark in $heap[S[0]]$, as we never access the $S[0]$-th cell.

Quick-heapifying an Array. We copy the array A to *heap*, and initialize both $S = |A|$ and $idx = 0$. The value of *capacity* must be at least $|A| + 1$. Note that this operation can be done in time $O(1)$ if we can take array A and use it as array *heap*.

```
Quickheap(Integer N) // constructor of an empty quickheap
1.  capacity ← N + 1, heap ← new Array[capacity], S ← {0}, idx ← 0

Quickheap(Array A, Integer N) // constructor of a quickheap from an array A
1.  capacity ← max{N, |A|} + 1, heap ← new Array[capacity], S ← {|A|}
2.  idx ← 0, heap.copy(A)

findMin()
1.  IQS(heap, idx, S)
2.  Return heap[idx mod capacity]

extractMin()
1.  IQS(heap, idx, S), idx ← idx + 1, S.pop()
2.  Return heap[(idx − 1) mod capacity]

insert(Elem x)
1.  pidx ← 0
2.  While TRUE Do // moving pivots, starting from pivot S[pidx]
3.      heap[(S[pidx] + 1) mod capacity] ← heap[S[pidx] mod capacity]
4.      S[pidx] ← S[pidx] + 1
5.      If (|S| = pidx + 1) OR // we are in the first chunk
          (heap[S[pidx + 1] mod capacity] ≤ x) Then // we found the chunk
6.          heap[(S[pidx] − 1) mod capacity] ← x, Return
7.      Else
8.          heap[(S[pidx]−1) mod capacity] ← heap[(S[pidx+1]+1) mod capacity]
9.          pidx ← pidx + 1 // go to next chunk
```

Fig. 5. Basic quickheap operations. N is an integer number giving the desired capacity of the heap. In operations **findMin** and **extractMin** we use a variant of **IQS**, that takes into account that array *heap* is circular and does not perform operation **pop** of line 1 of Fig. 2.

Finding the Minimum. Note that *idx* indicates the first cell used by the quickheap, and the pivots stored in S delimit chunks of partially ordered elements. Thus, the minimum of the heap must be placed within the first chunk ($heap[idx, S.\mathbf{top}() - 1]$). So, to find the minimum, we simply call a variant of $\mathbf{IQS}(heap, idx, S)$ and then return the element $heap[idx]$. This variant of **IQS** takes into account that the array is circular, and does not perform the **pop**() in line 1 of Fig. 2.

Extracting the Minimum. We call the same **IQS** variant of above to make sure that the minimum is located at cell $heap[idx]$. Next, we increase idx, pop S, and return $heap[idx - 1]$.

Inserting Elements. To insert a new element x into the quickheap we need to find the chunk where we can insert x in fulfillment of the pivot invariant. Then, we need to create an empty cell within this chunk in the array *heap*. A naive strategy will move every element in the array one position to the right, with an $O(m)$ worst-case complexity. Note, however, that it is enough to move only some pivots and elements to create an empty cell in the appropriate chunk.

We first move the fictitious pivot, updating its position in S, without comparing it with the new element x, so we have a free cell in the last chunk. Next, we compare x with the pivot at cell $S[1]$. If the pivot is smaller than or equal to x we place x in the free place left by pivot $S[0]$. Otherwise, we move the element at the right of pivot $S[1]$ to the free place left by pivot $S[0]$, and move pivot $S[1]$ one place to the right, updating its position in S. We repeat the process with the pivot at $S[2]$, and so on until we find the place where x has to be inserted, or we reach the first chunk.

4 Analysis of Quickheaps

We can prove that, along *any* sequence of operations including **insert**, **extractMin**, and **findMin**, the amortized cost of each of those operations is $O(\log m)$ on average, where m is the current size of the quickheap. Although the sequence of operations can be arbitrary, the analysis assumes that the elements inserted distribute uniformly across the set. The analysis also shows that the size of the stack (and hence the number of chunks) is $O(\log m)$ on average. For lack of space we will omit the proof of Lemma 2 [1].

This analysis is based on a key observation: quickheaps follow a *self-similar structure*, which means that the distribution of elements within a quickheap seen from the last chunk towards the first chunk is the same as the distribution within such quickheap seen from the second last chunk towards the first chunk, and so on. We start by proving that self-similarity property. Then, we introduce the *potential debt method* for amortized analysis. Finally, exploiting the self-similarity property, we analyze quickheaps using the potential debt method.

4.1 The Quickheap's Self-Similarity Property

In this section we introduce a formal notion of self-similarity for quickheaps. We show that this property is true at the beginning, and that it holds after extractions of minima, as well as insertions of elements that fall at independent and uniformly distributed positions in the heap. It follows that the property holds after arbitrary sequences of those operations, yet the positions of insertions cannot be arbitrary but uniformly distributed.

From now on, we consider that *array segments* are delimited by idx and the cell just before each pivot position $S[pidx]$ ($heap[idx \dots S[pidx]-1]$, thus segments overlap), and *array chunks* are composed by the elements between two consecutive pivot positions ($heap[S[pidx]+1 \dots S[pidx-1]-1]$) or between idx and the cell preceding the pivot on top of S ($heap[idx \dots S.\mathbf{top}()-1]$). We call $heap[idx \dots S.\mathbf{top}()-1]$ the first chunk, and $heap[S[1]+1 \dots S[0]-1]$ the last chunk. Analogously, we call $heap[idx \dots S.\mathbf{top}()-1]$ the first segment, and $heap[idx \dots S[0]-1]$ the last segment. The *pivot of a segment* will be the rightmost pivot within such segment (this is the one used to split the segment at the time **partition** was called on it). Thus, the pivot of the last segment is $S[1]$, whereas the first segment is the only one not having a pivot.

Using the traditional definition of the median of an n-element set —if n is odd the median is the $\frac{n+1}{2}$-th element, else it is the average of the values at positions $\frac{n}{2}$ and $\frac{n}{2}+1$—, let us call an element not smaller than the median of the array segment $heap[idx \dots S[pidx]-1]$ a *large element* of such segment. Analogously, let us call an element smaller than the median a *small element*.

The self-similarity property is the following:

Theorem 1 (quickheap's self-similarity property) *Given a segment $heap[idx \dots S[pidx]-1]$, the probability of its pivot being large is smaller than or equal to $\frac{1}{2}$, that is, $\mathbb{P}(pivot\ is\ large) \leq \frac{1}{2}$.*

To prove the property we need some notation. Let $\mathbb{P}_{i,j,n}$, $1 \leq i \leq n$, $j \geq 0$, $n > 0$, be the probability that the i-th element of a given segment of size n is the pivot of the segment after the j-th operation ($\mathbb{P}_{i,j,n} = 0$ outside bounds). Then, in the following we prove by induction on j that $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$, for all $2 \leq i \leq n$, j, n, after performing any sequence of operations **insert**, **findMin** and **extractMin**. That is, the probability of the element at cell i being the pivot is

[1] This proof of Lemma 2 is left in an appendix to this submission in case the referee wishes to check them.

non-increasing from left to right. Later, we use this to prove the self-similar property and some consequences of it.

Note that new segments with pivots are created when operations **extractMin** or **findMin** split the first segment. Note also that, just after a partitioned segment is created, the probabilities are $\mathbb{P}_{i,0,n} = \frac{1}{n}$, because the pivot is chosen at random from it, so we have proved the base case.

Lemma 2 *For each segment, the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after inserting a new element x at a uniformly chosen positition $[1, n]$.*

In order to analyze whether the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ is preserved after operations **findMin** and **extractMin** we need consider how **IQS** operates on the first segment. For this sake we introduce operation **pivoting**, which partitions the first segment with a pivot and pushes it into stack S. We also introduce operation **takeMin**, which increments idx, pops stack S and returns element $heap[idx - 1]$. Using these operations, we rewrite operation **extractMin** as: execute **pivoting** as many times as we need to push idx in stack S and next perform operation **takeMin**. Likewise, we rewrite operation **findMin** as: execute **pivoting** as many times as we need to push idx in stack S and next return element $heap[idx]$.

Operation **pivoting** creates a new segment and converts the previous first segment (with no pivot) into a segment with pivot, and where all the probabilities $\mathbb{P}_{i,0,n} = \frac{1}{n}$. The next lemma shows that after taking the minimum the property holds.

Lemma 3 *For each segment, the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after taking the minimum element of the quickheap.*

Proof. Due to previous calls to operation **pivoting**, the minimum is the pivot placed in idx. Once we pick it, the first segment vanishes. After that, the new first segment may be empty, but all the others have elements. For the empty segment the property is trivially true. Else, within each segment the probabilities change as follows: $\mathbb{P}_{i,j,n} = \mathbb{P}_{i+1,j-1,n+1}\frac{n+1}{n}$.

This gives us Theorem 1: When the segment is created, all the probabilities $\mathbb{P}_{i,j,n} = \frac{1}{n}$. Lemmas 2 and 3 guarantee that the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after inserting elements or taking the minimum. So, the property is preserved after any sequence of operations **insert**, **findMin** and **extractMin**. Therefore, adding up the probabilities $\mathbb{P}_{i,j,n}$ for the large elements, that is, for the $\left(\lceil \frac{n}{2} \rceil + 1\right)$-th to the n-th element, we obtain that $\mathbb{P}(\textit{pivot is large}) = \sum_{i=\lceil \frac{n}{2} \rceil + 1}^{n} \mathbb{P}_{i,j,n} \leq \frac{1}{2}$.

In the following, we use the self-similarity property to show two additional facts we use in the analysis of quickheaps. They are (i) the height of stack S is $O(\log m)$, and (ii) the sum of the size of the array segments is $\Theta(m)$.

Lemma 4 *The expected value of the height $\mathcal{H}$ of stack S is $O(\log m)$.*

Proof. Assume that the quickheap has m elements. Also, notice that the number $\mathcal{H}$ of pivots in the stack is monotonically non decreasing with m. Let us make some pessimistic simplifications. Let us take the largest value of the probability $\mathbb{P}(\textit{pivot is large})$, which is $\frac{1}{2}$. Furthermore, let us assume that if the pivot is taken from the large elements then it is the maximum element. Likewise, if it is taken from the small elements, then it is the element immediately previous to the median.

With these simplifications we have the following. When partitioning, we add one pivot to stack S. Then, with probabilities $\frac{1}{2}$ and $\frac{1}{2}$ the left partition has $m - 1$ or $\lfloor \frac{m}{2} \rfloor$ elements. So, we write the

following recurrence: $\mathcal{H} = T(m) = 1 + \frac{1}{2}T(m-1) + \frac{1}{2}T\left(\left\lfloor\frac{m}{2}\right\rfloor\right)$, $T(1) = 1$. Once again, using the monotonicity on the number of pivots, the recurrence is simplified to $T(m) \leq 1 + \frac{1}{2}T(m) + \frac{1}{2}T\left(\frac{m}{2}\right)$, which can be rewritten as $T(m) \leq 2 + T\left(\frac{m}{2}\right) \leq \ldots \leq 2j + T\left(\frac{m}{2^j}\right)$. As $T(1) = 1$, choosing $j = \log_2(m)$ we obtain that $\mathcal{H} = T(m) \leq 2\log_2 m + 1$. Finally, adding the fictitious pivot we have that $\mathcal{H} = 2(\log_2 m + 1) = O(\log m)$.

Lemma 5 *The expected value of the sum of the sizes of array segments is $\Theta(m)$.*

Proof. Using the same reasoning of Lemma 4, but considering that, when partitioning, we perform $m-1$ key comparisons, we write the following recurrence: $T(m) = m - 1 + \frac{1}{2}T(m-1) + \frac{1}{2}T\left(\left\lfloor\frac{m}{2}\right\rfloor\right)$, $T(1) = 0$. Using the monotonicity of $T(m)$ and neglecting the term -1, the recurrence is simplified to $T(m) \leq m + \frac{1}{2}T(m) + \frac{1}{2}T\left(\frac{m}{2}\right)$, which can be rewritten as $T(m) \leq 2m + T\left(\frac{m}{2}\right) \leq \ldots \leq 2m + m + \frac{m}{2} + \frac{m}{2^2} + \ldots + \frac{m}{2^{j-2}} + T\left(\frac{m}{2^j}\right)$. As $T(1) = 0$, choosing $j = \log_2(m)$ we obtain that $T(m) \leq 3m + m\sum_{i=1}^{\infty}\frac{1}{2^i} \leq 4m = \Theta(m)$. Therefore, the expected value of the sum of the array segment sizes is $\Theta(m)$.

4.2 The Potential Debt Method

To perform the amortized analysis of quickheaps we use a variant of the potential method, which we call the *potential debt method*. In this case, the potential function represents a total cost that has not yet been paid. At the end, this total debt must be split among all the performed operations. The potential debt is associated with the data structure as a whole.

The potential debt method works as follows. It starts with an initial data structure D_0 on which operations are performed. Let c_i be the actual cost of the i-th operation and D_i the data structure that results after applying the i-th operation to D_{i-1}. A *potential debt function* Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential debt associated with data structure D_i up to then. The *amortized cost* $\widetilde{c_i}$ of the i-th operation with respect to potential debt function Φ is defined by

$$\widetilde{c_i} \;=\; c_i - \Phi(D_i) + \Phi(D_{i-1}) \;. \tag{1}$$

Therefore, the amortized cost of i-th operation is the actual cost minus the potential debt variation due to the operation. Thus, the total amortized cost for N operations is

$$\sum_{i=1}^{N}\widetilde{c_i} \;=\; \sum_{i=1}^{N}(c_i - \Phi(D_i) + \Phi(D_{i-1})) \;=\; \sum_{i=1}^{N} c_i - \Phi(D_n) + \Phi(D_0) \;. \tag{2}$$

If we define a potential function Φ so that $\Phi(D_N) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{N}\widetilde{c_i}$ is a lower bound on the total actual cost $\sum_{i=1}^{N} c_i$. However, if we sum a positive cost $\Phi(D_N) - \Phi(D_0)$ to the amortized cost $\sum_{i=1}^{N}\widetilde{c_i}$, we compensate for the debt and obtain an upper bound on the actual cost $\sum_{i=1}^{N} c_i$. Thus, in Eq. (3) we write an amortized cost $\widehat{c_i}$ considering the potential debt, by assuming that we perform N operations during the process, and the potential due to these operations is $\Phi(D_N)$.

$$\widehat{c_i} \;=\; \widetilde{c_i} + \frac{\Phi(D_N) - \Phi(D_0)}{N} \;=\; c_i - \Phi(D_i) + \Phi(D_{i-1}) + \frac{\Phi(D_N) - \Phi(D_0)}{N} \tag{3}$$

This way, adding up for all the N operations, we obtain that

$$\sum_{i=1}^{N}\widehat{c_i} \;=\; \sum_{i=1}^{N}\left(c_i - \Phi(D_i) + \Phi(D_{i-1}) + \frac{\Phi(D_N) - \Phi(D_0)}{N}\right) \;=\; \sum_{i=1}^{N} c_i.$$

4.3 Average-case Amortized Analysis of Quickheaps

In this section, we consider that we operate over a quickheap qh with m elements in its *heap* and a pivot stack S of average height $\mathcal{H} = O(\log m)$, see Lemma 4.

We define the quickheap potential debt function as the sum of the sizes of the partitioned segments delimited by idx and pivots in $S[0]$ to $S[\mathcal{H}-1]$ (note that the last pivot is not counted). Eq. (4) shows the potential function $\Phi(qh)$.

$$\Phi(qh) = \sum_{i=0}^{\mathcal{H}-1} (S[i] - idx) = \Theta(m) \text{ on average, by Lemma 5} \tag{4}$$

Thus, the potential debt of an empty quickheap $\Phi(qh_0)$ is 0, and the average potential debt of an m-elements quickheap is $\Theta(m)$, see Lemma 5. Note that if we start from an empty quickheap qh, for each element within qh we have performed at least operation **insert**, so we can assume that there are more operations than elements in the quickheap, $N \geq m$. Therefore, in the case of quickheaps, the term $\frac{\Phi(qh_N)-\Phi(qh_0)}{N}$ is $O(1)$ on average. So, we can omit this term, writing the amortized costs directly as $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1})$.

Operation insert. The amortized cost of operation **insert** is defined by $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1})$. The difference of the potential debt $\Phi(qh_{i-1}) - \Phi(qh_i)$ (< 0) depends on how many segments are extended due to the insertion. Note that for each segment we extend —which increases by 1 the potential debt—, we also pay one key comparison, but in the first segment there is no increase. Thus, it holds $c_i - \Phi(qh_i) + \Phi(qh_{i-1}) \leq 1$. Then, the amortized cost of operation **insert** is $O(1)$.

Creation of a quickheap. The amortized cost of constructing a quickheap from scratch is $O(1)$. Instead, the amortized cost of constructing a quickheap from an array A of size m is $O(m)$, as we can see this as an initialization plus a sequence of m $O(1)$ amortized cost element insertions. Note that the potential debt of the quickheap is 0, as there is only one pivot in S.

Operation extractMin. To analyze this operation, we again use auxiliary operations **pivoting** and **takeMin** (see Section 4.1). Thus, we consider that operation **extractMin** is a sequence of zero or more calls to **pivoting**, until pushing idx in stack S, and then a single call to **takeMin**.

Each time we call operation **pivoting**, the actual cost corresponds to the size of the first segment, which is not yet accounted in the potential debt. On the other hand, once we push the pivot, the potential debt increases by an amount which is exactly the size of the partitioned segment. Thus, the amortized cost of operation **pivoting** is zero. With respect to operation **takeMin**, its actual cost is $O(1)$, and the potential debt decreases by $\mathcal{H}-1$, as all the segments considered in the potential are reduced in one cell after taking the minimum. As the expected value of $\mathcal{H} = O(\log m)$, see Lemma 4, the amortized cost of operation **takeMin** is $O(\log m)$. Therefore, adding the amortized cost of **pivoting** and **takeMin** we obtain that the amortized cost of operation **extractMin** is $O(\log m)$.

Operation findMin. Using operation **pivoting**, we rewrite operation **findMin** as: execute **pivoting** as many times as we need to push idx in stack S (with amortized cost zero) and then return element $heap[idx]$ (also with cost zero). Then, the amortized cost of operation **findMin** is $O(1)$.

5 Quickheaps in Secondary Memory

Quickheaps exhibit high locality of reference. On one hand, we have the stack S, which is small and accessed sequentially. On the other hand, each pivot in S points to a position in the array *heap*. Array *heap* is only modified at those positions, and the positions themselves increase at most by one at each insertion. The only remaining operation is **IQS**, which sequentially accesses the elements of the first chunk.

Under the cache-oblivious assumption, we will consider that we keep in main memory: (i) the stack S; (ii) for each pivot in S, the disk block containing its current position in *heap*; and (iii) the longest possible prefix of $heap[idx, N]$, containing at least one disk block. According to Lemma 4, all this requires on average to hold $M = \Omega(B \log m)$ integers in main memory. Say that we have twice the main memory required for (i) and (ii), so that we still have $\Theta(M)$ cells for (iii).

Let us first consider operation **insert**. Assume that entry $heap[i]$ is stored at disk block $\lceil i/B \rceil$. Note that once a disk page is loaded because a pivot position is incremented from $i = B \cdot j$ to $i+1 = B \cdot j+1$, we have disk page $j+1$ in main memory. From then, at least B increments of pivot position i are necessary to load another disk page. Therefore, the amortized cost of an element insertion is $\mathcal{H}/B$. According to the results of the previous section, this is $O(\log(m)/B)$ on average.

The other operations are **findMin** and **extractMin**, which essentially translate into a sequence of **pivoting** actions. Each such action sequentially traverses $heap[idx, S.\mathbf{top}()-1]$. Let $\ell = S.\mathbf{top}() - idx$ be the length of the area to traverse. The area to traverse spans $\lceil \ell/B \rceil$ disk blocks. As we have in main memory at least the first block of $heap[idx, N]$, we have to load at most $\lceil \ell/B \rceil - 1 \leq \ell/B$ disk blocks. On the other hand, the CPU cost of such traversal is $O(\ell)$. According to the previous section, all those traversals cost $O(\log m)$ amortized CPU time on average. Hence, the amortized I/O cost is $O(\log(m)/B)$ on average. Maintaining a prefix of a given size in main memory is easily done in $O(1/B)$ amortized time per operation, since *idx* grows by one per **extractMin**.

However, the result is better. As we have $M' = \Theta(M)$ integers of main memory to store the prefix of array *heap*, it is not hard to see, by the self-similarity property (Theorem 1), that on average the first $\Theta(\log M)$ pivots will always be in main memory, and therefore accessing them will be I/O-free. Similarly, **pivoting** is I/O-free over the first M' elements of *heap*. If in the previous section we had defined the potential function Φ as the sum of of segment sizes minus M' for those longer than M', and assumed that the accesses were free over the first M' positions, the analysis would have been very similar except that **takeMin** would have costed $O(\log m - \log M') = O(\log(m/M))$. Still the argument that at most one out of B of all those operations can force a disk access applies.

Theorem 2 *If the* Quickheap *is operated in external memory as described, so that we maintain* $M = \Omega(\log m)$ *disk blocks in main memory, operations* **findMin**, **extractMin** *and* **insert** *have an average amortized I/O cost of* $O((1/B)\log(m/M))$, *where* m *is the maximum heap size along the process.*

We note that this result is similar to the lower bounds given in [5] for cache-oblivious sorting.

6 Experimental Results

We carry out a brief experimental validation of our data structure, and also compare it with the results presented in [4], which report the number of blocks read/written for different sequences of

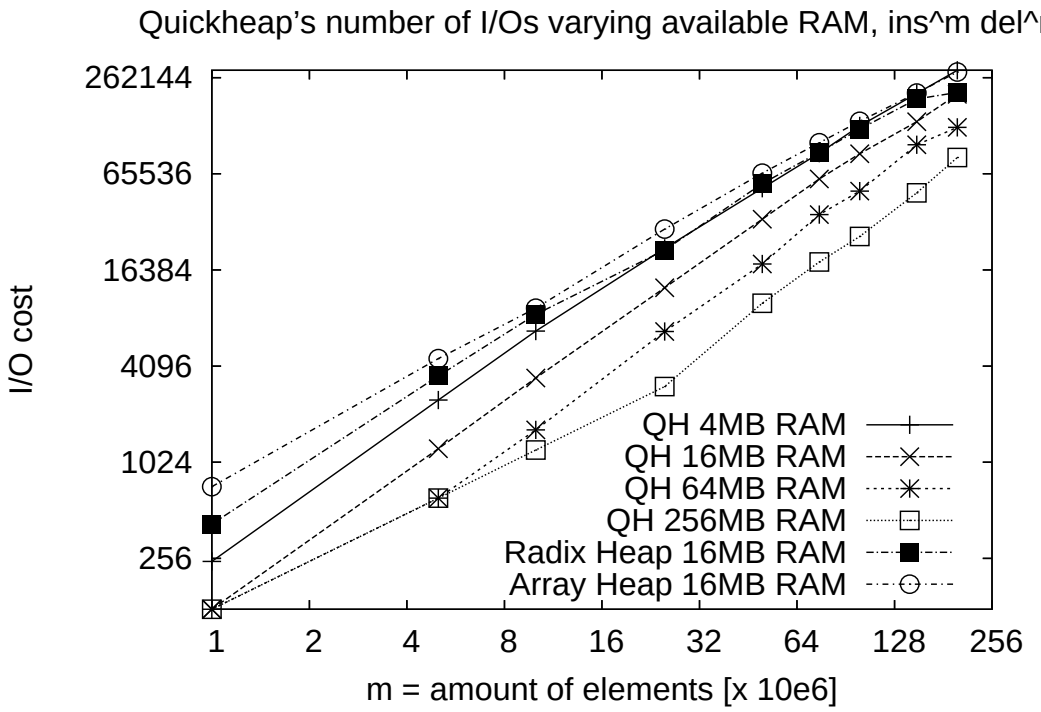

Fig. 6. I/O cost comparison of performing m random insertions followed by m minima extractions.

operations on the most promising secondary memory implementations. We use their same parameters $M = 16$ megabytes and $B = 32$ kilobytes, and run a sequence of N random insertions followed by N minima extractions. We consider different M values to show how M affects the performance of quickheaps.

The results are shown in Fig. 6. As it can be seen, quickheaps achieve a performance comparable with the best structures in [4] already with 4 megabytes of RAM. When using the same 16 megabytes, our structure performs 1.03 to 3.4 times fewer I/O accesses. Moreover, the best alternative structure is the *R-Heap*, which only works if the priorities of extracted elements from a nondecreasing sequence. If we consider the best alternative that works with no restriction (*Array-Heaps*), *QuickHeaps* perform 1.38 to 5.8 fewer I/Os. Other tests in [4] are harder to reproduce [2].

We also notice the logarithmic dependence on $m = \Theta(N)$ and on M (the plots are log-log), as expected from our analysis.

In addition, we note that a good part of the accesses carried out by *Quickheaps* are indeed local, as they come from **partition**, which sequentially traverses the first chunk. Currently our simulator does not give separate accounting for bulk and random I/Os, but this should be ready for the final version.

[2] For example, they also report real times, but those should be rerun in our machine and we do not have access to LEDA, which is mandatory to run their code.

7 Conclusions

We have introduced *Quickheaps*, a simple and efficient data structure implementing priority queues. *Quickheaps* are as simple to implement as classical binary heaps, need almost no extra space, are efficient in practice, and exhibit high locality of reference. We exploit this last property to design a cache-oblivious version that performs nearly optimally on secondary memory. We prove that the average amortized cost per operation is $O(\log m)$ in main memory and $O((1/B)\log(m/M))$ on disk, where m is the maximum heap size achieved, B the block size, and M the main memory size.

Our experimental results show that *Quickheaps* are extremely competitive in practice: using the same amount of memory, they perform 1.03 to 3.4 fewer I/O accesses than the best alternatives tested in the survey by Brengel et al. [4].

Future work considers more thorough experiments, including other sequences of accesses and real CPU and I/O times. We also plan to achieve amortized worst-case guarantees for the data structure, for example by replacing the randomized pivoting by an order statistic on the first chunk.

References

1. R. Ahuja, K. Mehlhorn, J. Orlin, and R. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.
2. L. Arge. The buffer tree: A new technique for optimal i/o-algorithms (extended abstract). In *Proc. 4th International Workshop on Algorithms and Data Structures (WADS'95)*, LNCS 995, pages 334–345, 1995.
3. R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.
4. K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues in external memory. *ACM Journal of Experimental Algorithmics*, 5(17), 2000.
5. G. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing (STOC'03)*, pages 307–315, 2003.
6. G. Brodal and J. Katajainen. Worst-case external-memory priority queues. In *Proc. 6th Scandinavian Workshop on Algorithm Theory (SWAT'98)*, LNCS 1432, pages 107–118, 1998.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
8. R. Fadel, K. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.
9. M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations on Computer Science (FOCS'99)*, pages 285–297, 1999.
10. C. A. R. Hoare. Algorithm 65 (FIND). *Comm. of the ACM*, 4(7):321–322, 1961.
11. D. Hutchinson, A. Maheshwari, J. Sack, and R. Velicescu. Early experiences in implementing buffer trees. In *Proc. 2nd International Workshop on Algorithmic Engineering (WAE'97)*, pages 92–103, 1997.
12. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE Symposium on Parallel and Distributed Processing (SPDP'96)*, page 169, 1996.
13. R. Paredes and G. Navarro. Optimal incremental sorting. In *Proc. 8th Workshop on Algorithm Engineering and Experiments and 3rd Workshop on Analytic Algorithmics and Combinatorics (ALENEX-ANALCO'06)*, pages 171–182. SIAM Press, 2006.
14. P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5(7), 2000.
15. J. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001. Version revised at 2007 from `http://www.cs.duke.edu/~jsv/Papers/Vit.IO_survey.pdf`.
16. J. Williams. Algorithm 232 (HEAPSORT). *Comm. of the ACM*, 7(6):347–348, 1964.

A Proof of Lemma 2 — to be read at the discretion of the reviewer

This appendix is included in case the referee wishes to check the omitted proof, but it is not necessary to follow the paper. The proof will probably not fit in a final version, but will be included in a techreport referenced from the paper.

Proof of Lemma 2

We suppose that after the $(j-1)$-th operation the segment has $n-1$ elements. As we insert x in the j-th operation, the resulting segment contains n elements. The probability that after the insertion the pivot p is at cell i depends on whether p was at cell $i-1$ and we have inserted x in any of first $i-1$ positions $1,\ldots,i-1$, so the pivot moved to the right; or the pivot already was at cell i and we have inserted x in any of last $n-i$ positions $i+1,\ldots,n$. So, we have the recurrence of Eq. (5).

$$\mathbb{P}_{i,j,n} \;=\; \mathbb{P}_{i-1,j-1,n-1}\frac{i-1}{n} \;+\; \mathbb{P}_{i,j-1,n-1}\frac{n-i}{n} \tag{5}$$

From the inductive hypothesis we have that $\mathbb{P}_{i,j-1,n-1} \leq \mathbb{P}_{i-1,j-1,n-1}$. Multiplying both sides by $\frac{n-i}{n}$, adding $\mathbb{P}_{i-1,j-1,n-1}\frac{i-1}{n}$ and rearranging terms we obtain the inequality of Eq. (6), whose left side corresponds to the recurrence of $\mathbb{P}_{i,j,n}$.

$$\mathbb{P}_{i-1,j-1,n-1}\frac{i-1}{n} + \mathbb{P}_{i,j-1,n-1}\frac{n-i}{n} \;\leq\; \mathbb{P}_{i-1,j-1,n-1}\frac{i-2}{n} + \mathbb{P}_{i-1,j-1,n-1}\frac{n+1-i}{n} \tag{6}$$

By the inductive hypothesis again, $\mathbb{P}_{i-1,j-1,n-1} \leq \mathbb{P}_{i-2,j-1,n-1}$, for $i > 2$. So, replacing on the right side above we obtain the inequality of Eq. (7), where, in the right side we have the recurrence for $\mathbb{P}_{i-1,j,n}$.

$$\mathbb{P}_{i,j,n} \;\leq\; \mathbb{P}_{i-2,j-1,n-1}\frac{i-2}{n} + \mathbb{P}_{i-1,j-1,n-1}\frac{n+1-i}{n} \;=\; \mathbb{P}_{i-1,j,n} \tag{7}$$

With respect to $i = 2$, note that the term $\mathbb{P}_{i-2,j-1,n-1}\frac{i-2}{n}$ from Eq. (7) vanishes. Thus, this equation can be rewritten as $\mathbb{P}_{2,j,n} \leq \mathbb{P}_{1,j-1,n-1}\frac{n-1}{n}$. Note that the right side is exactly $\mathbb{P}_{1,j,n}$ according to the recurrence Eq. (5) evaluated for $i = 1$.