On Sorting, Heaps, and Minimum Spanning Trees *

Gonzalo Navarro† Rodrigo Paredes‡

† Dept. of Computer Science, University of Chile, Santiago, Chile.
gnavarro@dcc.uchile.cl
‡ Depto. de Ciencias de la Computación, Universidad de Talca, Curico, Chile.
raparede@utalca.cl

Abstract

Let A be a set of size m. Obtaining the first $k \leq m$ elements of A in ascending order can be done in optimal $O(m + k \log k)$ time. We present *Incremental Quicksort* (**IQS**), an algorithm (online on k) which incrementally gives the next smallest element of the set, so that the first k elements are obtained in optimal expected time for any k. Based on **IQS**, we present the *Quickheap* (**QH**), a simple and efficient priority queue for main and secondary memory. *Quickheaps* are comparable with classical binary heaps in simplicity, yet are more cache-friendly. This makes them an excellent alternative for a secondary memory implementation. We show that the expected 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 main memory size M and block size B, in a cache-oblivious fashion. As a direct application, we use our techniques to implement classical Minimum Spanning Tree (MST) algorithms. We use **IQS** to implement Kruskal's MST algorithm and **QH**s to implement Prim's. Experimental results show that **IQS**, **QH**s, external **QH**s, and our Kruskal's and Prim's MST variants are competitive, and in many cases better in practice than current state-of-the-art alternative (and much more sophisticated) implementations.

Keywords: Kruskal's MST algorithm, Prim's MST algorithm, Incremental sorting, Priority Queues, External Priority Queues.

1 Introduction

There are cases where we need to obtain the smallest elements from a fixed set without knowing how many elements we will end up needing. Prominent examples are Kruskal's Minimum Spanning Tree (MST) algorithm [24] and ranking by Web search engines [3]. Given a graph, Kruskal's MST algorithm processes the edges one by one, from smallest to largest, until it forms the MST. At this point, remaining edges are not considered. Web search engines display a very small sorted subset of the most relevant documents among all those satisfying the query. Later, if the user wants more results, the search engine displays the next group of most relevant documents, and so on. In both cases, we could sort the whole set and later return the desired objects, but obviously this is more work than necessary.

*Supported in part by the Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile; Yahoo! Research grant "Compact Data Structures"; and Fondecyt grant 1-080019, Chile. Early parts of this work appeared in ALENEX 2006 [29].

This problem can be called *Incremental Sorting*. It 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. Therefore, *Incremental Sorting* is the online version of a variant of the *Partial Sorting* problem: Given a set A of m numbers and an integer $k \leq m$, output the smallest k elements of A in ascending order. This can be easily solved by first finding the k-th smallest element of A using $O(m)$ time *Select* algorithm [5], and then collecting and sorting the elements smaller than the k-th element. The resulting complexity, $O(m + k \log k)$, is optimal under the comparison model, as every cell must be inspected and there are $\Pi_{0 \leq j < k}(m - j) = \frac{m!}{(m-k)!}$ possible answers, thus a lower bound is $m + \log \frac{m!}{(m-k)!} = \Omega(m + k \log k)$.

A practical version of the above method uses Quickselect and Quicksort as the selection and sorting algorithms, obtaining $O(m + k \log k)$ expected complexity. Recently, it has been shown that the selection and sorting steps can be interleaved, which improves the constant terms [26].

To solve the online problem (incremental sort), we have to select the smallest element, then the second smallest, and so on until the process finishes at some unknown value $k \in [0, m-1]$. One can do this by using Select to find each of the first k elements, for an overall cost of $O(km)$. This can be improved by transforming A into a min-heap [42] in time $O(m)$ [15] and then performing k extractions. This premature cut-off of the heapsort algorithm [42] has $O(m + k \log m)$ worst-case complexity. Note that $m + k \log m = O(m + k \log k)$, as they can differ only if $k = o(m^c)$ for any $c > 0$, in which case m dominates $k \log m$. However, according to experiments this scheme is much slower than the offline practical algorithm [26] if a classical heap is used.

Sanders [32] proposes *sequence heaps*, a cache-aware priority queue, to solve the online problem. Sequence heaps are optimized to insert and extract *all the elements* in the priority queue at a small amortized cost. Even though the total CPU time used for this algorithm in the whole process of inserting and extracting all the m elements is pretty close to the time of running Quicksort, this scheme is not so efficient when we want to sort just a small fraction of the set. Then the quest for a practical online algorithm for partial sorting is raised.

In this paper we present *Incremental Quicksort* (**IQS**), a practical and efficient algorithm for solving the online problem, within $O(m + k \log k)$ expected time. Based on **IQS**, we present the *Quickheap* (**QH**), a simple and efficient data structure for implementing priority queues in main and secondary memory. *Quickheaps* are comparable with classical binary heaps in simplicity, yet are more cache-friendly. This makes them an excellent alternative for a secondary memory implementation. **QH**s achieve $O(\log m)$ expected amortized time per operation when they fit in main memory, and $O((1/B) \log(m/M))$ I/O cost when there are M bytes of main memory and the block size is B in secondary memory, working in a cache-oblivious fashion. **IQS** and **QH**s can be used to improve upon the current state of the art on many algorithmic scenarios. In fact, we plug them in the classic Minimum Spanning Tree (MST) techniques: We use incremental quicksort to boost Kruskal's MST algorithm [24], and a quickheap to boost Prim's MST algorithm [31]. Given a random graph $G(V, E)$, we compute its MST in $O(|E| + |V| \log^2 |V|)$ average time.

Experimental results show that **IQS**, **QH**s, external **QH**s and our Kruskal's and Prim's MST variants are extremely competitive, and in many cases better in practice than current state-of-the-art (and much more sophisticated) alternative implementations. **IQS** is approximately four times faster than the classic alternative to solve the online problem. **QH**s are competitive with pairing heaps [16] and up to four times faster than binary heaps [42] (according to Moret and Shapiro

[27], these are the fastest priority queue implementations in practice). Using the same amount of memory, our external **QH** performs up to 3 times fewer I/O accesses than *R-Heaps* [1] and up to 5 times fewer than *Array-Heaps* [8], which are the best alternatives tested in the survey by Brengel et al. [6]. External-memory Sequence Heaps [32], however, are faster than **QH**s, yet these are much more sophisticated and not cache-oblivious. Finally, our Kruskal's version is much faster than any other Kruskal's implementation we could program or find for any graph density. As a matter of fact, it is faster than Prim's algorithm [31], even as optimized by Moret and Shapiro [27], and also competitive with the best alternative implementations we could find [22, 23]. On the other hand, our Prim's version is rather similar to our Kruskal's one, yet it is resistant to some Kruskal's worst cases, such as the *lollipop graph*.

The rest of this paper is organized as follows. In the rest of the Introduction we briefly review some of the related work. In Section 2 we present our incremental sorting algorithm. Then, in Sections 3 and 4 we build on it to design Quickheaps in main memory. Next, in Section 5 we show how to adapt our priority queue to work in secondary memory. In Section 6 we apply our basic algorithms and structures to boost the construction of the MST of a graph. Section 7 gives our experimental results. Finally, in Section 8 we give our conclusions and some directions for further work. Pseudo-codes and more experiments are available [28].

1.1 Priority Queues

A *priority queue* (PQ) is a data structure which allows maintaining a set of elements in a partially ordered way, enabling efficient element insertion (**insert**), minimum finding (**findMin**) and minimum extraction (**extractMin**) —or alternatively, maximum finding and extraction. In the following we focus on obtaining the minima, that is in min-order PQs. The set of operations can be extended to construct a priority queue from a given array A (**heapify**), increase or decrease the priority of an arbitrary element (**increaseKey** and **decreaseKey**, respectively), delete an arbitrary element from the priority queue (**delete**), and a long so on.

The classic PQ implementation uses a binary heap [42, 11]. Wegener [41] proposes a *bottom-up deletion* algorithm, which addresses operation **extractMin** performing only $\log_2 m + O(1)$ key comparisons per extraction on average, in heaps of m elements. Other well-known priority queues are *sequence heaps* [32], *binomial queues* [40], *Fibonacci heaps* [17], *pairing heaps* [16], *skew heaps* [34], and *van Emde Boas queues* [38]. All are based on binary comparisons, except the latter which handles an integer universe $[0, m]$.

1.2 External Memory Priority Queues

When working in the secondary memory scenario, we assume that we have M bytes of fast-access internal memory and an arbitrarily large slow-access external memory located in one or more independent disks. Data between the internal memory and the disks is transferred in blocks of size B, called *disk pages*. In this model, the algorithmic performance is usually measured by counting the number of disk access performed, which we call I/Os. Thus, to improve the I/O performance, external memory techniques focus on guaranteeing good locality of reference. Therefore, external memory PQs usually offer just the basic operations, namely, **insert**, **findMin** and **extractMin**. This is because others, like **delete** or **decreaseKey**, need at least one random access to the queue.

Some external memory PQs are *buffer trees* [2, 20], *M/B-ary heaps* [25, 14], and *Array Heaps* [8], all of which achieve the lower bound of $\Theta((1/B)\log_{M/B}(m/B))$ amortized I/Os per operation [39]. Those structures, however, are rather complex to implement and heavyweight in practice (in extra space and time) [6]. Other techniques are simple but do not perform so well (in theory or in practice), for example those using B-trees [4]. A practical comparison of existing secondary memory PQs was carried out by Brengel et al. [6], where in addition they adapt two-level radix heaps [1] to secondary memory (*R-Heaps*), and also simplify *Array-Heaps* [8]. The latter stays optimal in the amortized sense and becomes simple to implement. Experiments [6] show that *R-Heaps* and *Array-Heaps* were the best choices for secondary memory. In the same issue, Sanders introduced *sequence heaps* [32], which can be seen as a simplification of the improved *Array-Heaps* [6]. Sanders reports that *sequence heaps* are faster than the improved *Array-Heaps* [12, 13].

1.3 Minimum Spanning Trees

Assume that $G(V, E)$ is a connected undirected graph with a nonnegative cost function $weight_e$ assigned to its edges $e \in E$. A minimum spanning tree mst of the graph $G(V, E)$ is a tree composed of $n-1$ edges of E connecting all the vertices of V at the lowest total cost $\sum_{e \in mst} weight_e$.

The most popular algorithms to solve this problem are Kruskal's [24] and Prim's [31], whose basic versions have complexity $O(m \log m)$ and $O\big(n^2\big)$, respectively. There are several other MST algorithms compiled by Tarjan [35]. Recently, Chazelle [9] gave an $O(m\alpha(m, n))$ time algorithm. Later, Pettie and Ramachandran [30] proposed an algorithm that runs in optimal time $O(\mathcal{T}^*(m, n))$, where $\mathcal{T}^*(m, n)$ is the minimum number of edge-weight comparisons needed to determine the MST of any graph $G(V, E)$ with m edges and n vertices. Its best known upper bound is also $O(m\alpha(m, n))$. These algorithms almost reach the lower bound $\Omega(m)$, yet they are so complicated that their interest is mainly theoretical.

Several experimental studies on MST exist [27, 22, 23]. Moret and Shapiro [27] compare several versions of Kruskal's, Prim's and Tarjan's algorithms, concluding that the best in practice (albeit not in theory) is Prim's using pairing heaps [16]. Their experiments show that neither Cheriton and Tarjan's [10] nor Fredman and Tarjan's algorithm [17] ever approach the speed of Prim's algorithm using pairing heaps. Moreover, they show that it is possible to use heaps to improve Kruskal's algorithm. The idea is to min-heapify the set E, and then to perform as many min-extractions of the lowest-cost edge as needed (they do this in their Kruskal's demand-sorting version [27]). The result is a rather efficient MST version with complexity $O(m + k \log m)$, being $k \leq m$ the number of edges reviewed by Kruskal. However, they also show that the worst-case behavior of Kruskal's algorithm stays poor: If the graph has two distinct components connected by a single, very costly edge, incremental sorting is forced to process the whole edge set. Katriel et al. [22, 23] present the algorithm *iMax*, whose expected complexity is $O(m + n \log n)$.

Final remarks on Kruskal's and Prim's algorithms are in order. If we are using either full or random graphs whose edge costs are assigned at random independently of the rest (using any continuous distribution), the subgraph composed by V with the edges reviewed by the Kruskal's algorithm is a random graph [21]. Based on that analysis [21, p. 349], we expect to finish the MST construction (that is, to connect the random subgraph) upon reviewing $\frac{1}{2}n \ln n + \frac{1}{2}\gamma n + \frac{1}{4} + O\left(\frac{1}{n}\right)$ edges, which can be much smaller than m. For each of these edges, we use $O(\log m)$ time to select and extract the minimum element of the heap. So, the average complexity of Kruskal with

incremental (or demand) sorting is $O(m + n \log n \log m) = O\left(m + n \log^2 n\right)$ (as $n - 1 \leq m \leq n^2$).

On the other hand, a practical, fast implementation of Prim's algorithm uses binary heaps, reducing the time to $O(m \log n)$, which is relevant when $m = o\big(n^2 / \log n\big)$. Alternatively, Prim's can be implemented using Fibonacci Heaps [17] to obtain $O(m + n \log n)$ complexity.

2 Optimal Incremental Sorting

Let A be a set of size m. Obtaining the first $k \leq m$ elements of A in ascending order can be done in optimal $O(m + k \log k)$ time. We present *Incremental Quicksort* (**IQS**), an algorithm (online on k) which incrementally gives the next smallest element of the set, so that the first k elements are obtained in optimal time for any k. As explained in the Introduction, this is not a big achievement because the same can be obtained using a priority queue. However, **IQS** performs better in practice than the best existing online algorithm.

Essentially, **IQS** calls *Quickselect* [19] to find the smallest element of arrays $A[0, m-1]$, $A[1, m-1]$, ..., $A[k-1, m-1]$. This naturally leaves the k smallest elements sorted in $A[0, k-1]$. The key point to avoid the $O(km)$ complexity is to note that when we call Quickselect on $A[1, m-1]$, we can reuse the sequence of decreasing pivots that has already been used in the previous invocation on $A[0, m-1]$. To do that, it suffices with considering an auxiliary stack S in order to manage this sequence of decreasing pivot positions, as they will be relevant for the next calls to Quickselect.

Figure 1 (left) 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 $A[0, p-1]$ 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 Figure 1 (left), the second minimum (18) is the pivot on the top of S, so we pop it and return $A[1]$. Figure 1 (right) shows how **IQS** finds the third minimum using the pivot information stored in S. Notice that **IQS** *just works on the current first chunk* ($\{29, 25\}$). In this case it adds one pivot position to S and returns the third element (25) in the next recursive call. The incremental sorting process will continue as long as needed, and it can be stopped in any time.

The algorithm is given in Figure 2. Stack S is initialized to $S = \{|A|\}$. **IQS** receives the set A, the index idx[1] of the element sought (that is, we seek the smallest element in $A[idx, m-1]$), and the current stack S (with former pivot positions). First it checks whether the top element of S is the desired index idx, in which case it pops idx and returns $A[idx]$. Otherwise it chooses a random pivot index $pidx$ from $[idx, S.\mathbf{top}()-1]$. Pivot $A[pidx]$ is used to partition $A[idx, S.\mathbf{top}()-1]$. After the partitioning, the pivot has reached its final position $pidx'$, which is pushed in S. Finally, a recursive invocation continues the work on the left hand of the partition.

Recall that **partition**(A, $A[pidx]$, i, j) rearranges $A[i, j]$ and returns the new position $pidx'$ of the original element $A[pidx]$, so that, in the rearranged array, all the elements smaller/larger than $A[pidx']$ appear before/after $pidx'$. Thus, pivot $A[pidx']$ is left at the correct position it would have in the sorted array $A[i, j]$. 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.

[1]Since we start counting array positions from 0, the place of the k-th element is $k-1$, so $idx = k-1$.

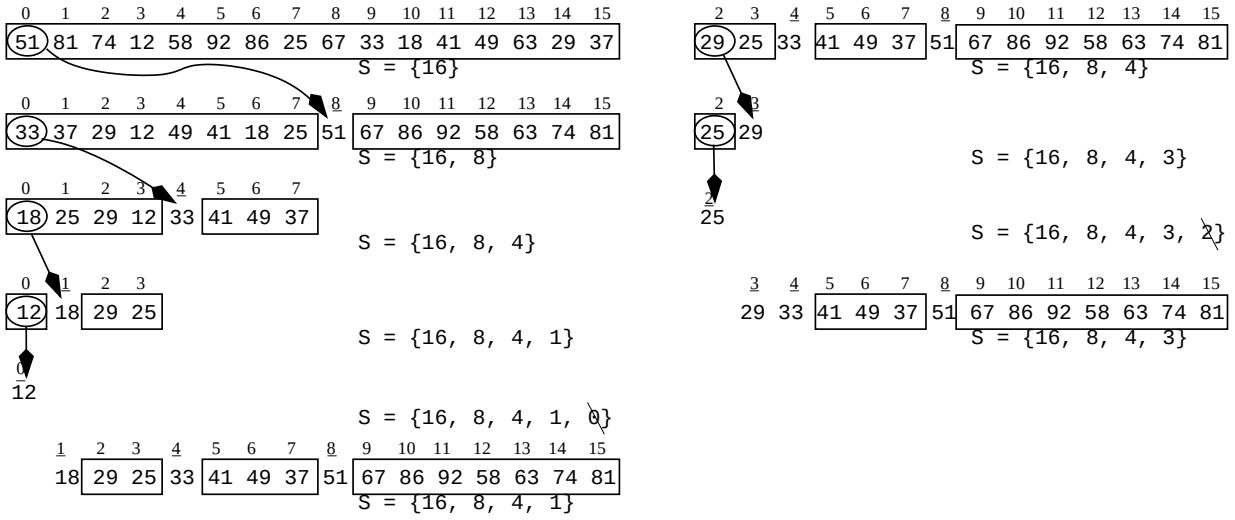


Figure 1: Example of **IQS**. Each line corresponds to a new partition of a sub-array. In the example we use the first element of the current partition as the pivot, but it could be any other element. The bottom line shows the array with the partitions generated by the first call to **IQS** and the pivot positions stored in S. On the left, finding the first element. On the right, finding the third element. Using the pivot information **IQS** only works on the current first chunk ($\{29, 25\}$).

IQS (Set A, Index idx, Stack S)
// Precondition: $idx \leq S.\mathbf{top}()$
1. **If** $idx = S.\mathbf{top}()$ **Then** $S.\mathbf{pop}()$, **Return** $A[idx]$
2. $pidx \leftarrow \mathbf{random}[idx, S.\mathbf{top}()-1]$
3. $pidx' \leftarrow \mathbf{partition}(A, A[pidx], idx, S.\mathbf{top}()-1)$
// Invariant: $A[0] \leq \ldots \leq A[idx-1] \leq A[idx, pidx'-1] \leq A[pidx']$
// $\leq A[pidx'+1, S.\mathbf{top}()-1] \leq A[S.\mathbf{top}(), m-1]$
4. $S.\mathbf{push}(pidx')$
5. **Return IQS**(A, idx, S)

Figure 2: Algorithm Incremental Quicksort (**IQS**). Stack S is initialized to $S \leftarrow \{|A|\}$. 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 2.1 (pivot invariant)**.** *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 $A[p+1, m-1]$.*

Proof. Initially this holds since $i = 0$ and $S = \{m\}$. Assume this is valid before pushing p, when p' was the top of the stack. Since the pivot was chosen from $A[i, p'-1]$ and left at some position $i \leq p \leq p'-1$ after partitioning, property (1) is guaranteed. As for property (2), after the

partitioning it still holds for any pivot other than p, as the partitioning rearranged elements at the left of all previous pivots. With respect to p, the partitioning ensures that elements smaller than p are left at $A[i, p-1]$, while larger elements are left at $A[p+1, p'-1]$. Since $A[p]$ was already not larger than elements in $A[p', m-1]$, the lemma holds. It obviously remains true after removing elements from S. □

The worst-case complexity of **IQS** is $O(m^2)$, but it is easy to derive a worst-case optimal version from it. The only change is in line 2 of Figure 2, where the random selection of the next pivot position must be changed to choosing the median of $A[idx, S.\textbf{top}() - 1]$, using the linear-time selection algorithm [5]. See Paredes [28] for details.

Let us now consider the expected case complexity. In **IQS**, the final pivot position p after the partitioning of $A[0, m-1]$ distributes uniformly in $[0, m-1]$. Let $T(m, k)$ be the expected number of key comparisons needed to obtain the k smallest elements of $A[0, m-1]$. After the $m-1$ comparisons used in the partitioning, there are three cases depending on p: (1) $k \leq p$, in which case the right partition remains until the end of the process, and the total extra cost will be $T(p, k)$ to solve $A[0, p-1]$; (2) $k = p+1$, in which case the left partition will be fully sorted at cost $T(p, p)$; and (3) $k > p+1$, in which case we pay $T(p, p)$ on the left partition, whereas the right partition, of size $m-1-p$, will be sorted incrementally so as to obtain the remaining $k-p-1$ elements. Thus **IQS** expected cost is $T(m, k) = m-1+\frac{1}{m}\left(\sum_{p=k}^{m-1} T(p, k) + T(k-1, k-1) + \sum_{p=0}^{k-2}\Big(T(p, p) + T(m-1-p, k-p-1)\Big)\right)$. This exact recurrence describes the offline Partial Sorting algorithm [26], where it was solved. The result can be rewritten as $T(m, k) = \Theta(m + k \log k)$, see Paredes [28].

Theorem 2.1 (**IQS**'s expected case complexity)**.** *Given a set A of m numbers* **IQS** *finds the k smallest elements, for any unknown value $k \leq m$, in $O(m + k \log k)$ expected time.* □

3 Quickheaps

Let us go back to the last line of Figure 1 (left), drawn in Figure 3, where we add ovals indicating pivots. For the sake of simplifying the following explanation, we also add a ∞ mark signaling a fictitious pivot in the last place of the array.

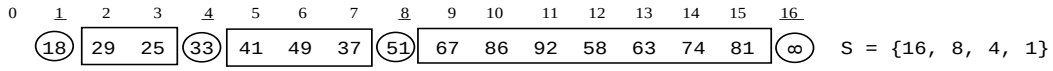


Figure 3: Last line of Figure 1.

By virtue of the **IQS** invariant (see Lemma 2.1), 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. Then, another pivot 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).

This resembles a heap structure, in the sense that objects in the array are semi-ordered. In the following, we exploit this property to implement a priority queue over an array processed with

algorithm **IQS**. We call this **IQS**-based priority queue *Quickheap* (**QH**). From now on we explain how to obtain a min-order quickheap. For practical reasons, elements within the quickheap are stored in a circular array, so that we can handle arbitrarily long sequences of operations as long as we maintain no more elements than the capacity of the circular array.

3.1 Data Structures for Quickheaps

To implement a quickheap we need the following structures (we use Figure 3 as an example):

1. An array *heap* to store the elements. In the example it is $\{18, 29, \ldots, 81, \infty\}$.
2. A stack S to store the positions of pivots partitioning *heap*. Recall that the bottom pivot index indicates the fictitious pivot ∞, and the top one the smallest pivot. In the example, the stack S is $\{16, 8, 4, 1\}$.
3. An integer *idx* to indicate the first cell of the quickheap. In the example $idx = 1$. Note that the last cell of the quickheap (the position of the fictitious pivot ∞) is maintained in $S[0]$.
4. An integer *capacity* to indicate the size of *heap*. We can store up to $capacity - 1$ elements in the quickheap (as we need a cell for the fictitious pivot ∞).

Note that in the case of circular arrays, we must take into account that an object whose position is *pos* is actually located in the cell *pos* mod *capacity* of the circular array *heap*. 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 the circular array *heap* as the operation progresses. From now on, we will omit the expression mod *capacity* in order to simplify the reading.

Throughout this section we assume that we know beforehand the value of *capacity*, that is, the maximum number of elements we store in the priority queue. If this is not the case, we can implement array *heap* as a *dynamic table* [11, Section 17.4], just adding a constant amortized factor to the cost of quickheap operations.

3.2 Quickheap Operations

Creation of 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 array plus a fictitious cell. On the other hand, to create a quickheap from an array A, we copy it to *heap*, and initialize both $S = |A|$ and $idx = 0$. The value of *capacity* must be at least $|A| + 1$.[2] This operation can be done in time $O(1)$ if we can take array A and use it as array *heap*.

Finding the minimum. To find the minimum of the heap, we focus on the first chunk, which is delimited by the cells *idx* and $S.\mathbf{top}() - 1$. For this sake, we just call $\mathbf{IQS}(heap, idx, S)$ and then return the element $heap[idx]$. However, in this case **IQS** does not pop the pivot on top of S. Remember that an element whose position is *pos* is located at cell *pos* mod *capacity*, thus we have to slightly change algorithm **IQS** to manage the positions in the circular array.

[2]Indeed we do not really need that further cell, we just let the fake pivot $S[0]$ point to a nonexistent cell.

Extracting the minimum. To extract the minimum, we first make sure that the minimum is located in the cell $heap[idx]$. (Once again, in this case **IQS** does not pop the pivot on top of S.) Next, we increase idx and pop S. Finally, we return the element $heap[idx - 1]$.

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 (Lemma 2.1). Thus, we need to create an empty cell within this chunk in the array $heap$. Note that we do not need to move every element in the array one position to the right, but 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 first element at the right of pivot $S[1]$ to the free place left by pivot $S[0]$, and move the 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. Figure 4 shows an example.

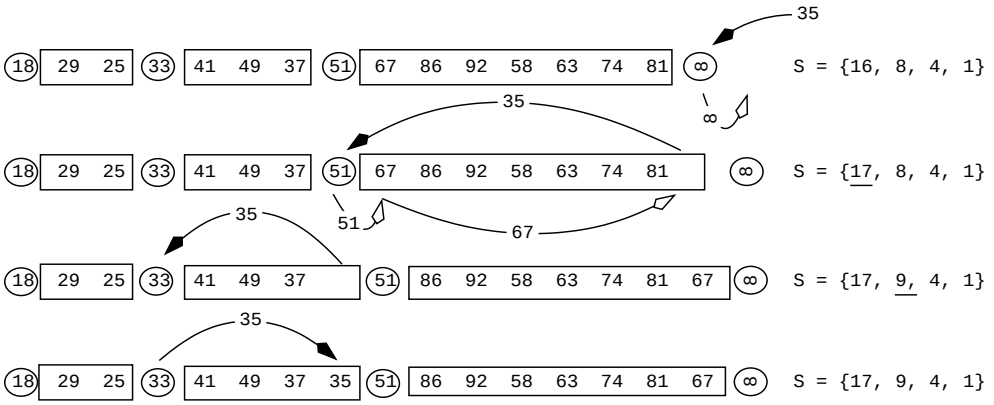


Figure 4: Inserting a new element into a quickheap. The figure shows the pivots we have to compare with the new element to insert it into the quickheap, and the elements we have to move to create the free cell to allocate the new element.

Deleting arbitrary elements. Given a position pos, this operation removes from the quickheap the element at that cell. When we delete a non-pivot element we move some pivots and elements one cell to the left. If we remove a pivot element, we drop it, join the two chunks and continue as if it were a non-pivot deletion.

This operation, as well as **increaseKey** and **decreaseKey**, requires to know the internal positions of elements in the quickheap, not only their identifiers. Thus we might have to augment the quickheap with a dictionary which, given an element identifier, answers its respective position, and to maintain this mapping upon changes in the quickheap. There are several options for implementing this dictionary, depending on the application, which range from constant to $O(\log m)$ time per update. We do not include this possible extra cost in our analysis.

Using the dictionary we obtain the element position pos. To delete the element, we first need to find its chunk. Note that each chunk has a pivot at its right, so we reference the chunk by that pivot, $pidx$. Therefore, we traverse the stack S to find the smallest pivot that is $\geq pos$.

Once we have a pivot $pidx$ at a position greater than pos, we repeat the following process. We place the element previous to the $pidx$-th pivot in the position pos, that is, we move the element $heap[S[pidx] - 1]$ to position $heap[pos]$, so we have a free cell at position $S[pidx] - 1$. Then, we move the pivot $heap[S[pidx]]$ one place to the left, and update its position in S. Then we update pos to the old pivot position, $pos = S[pidx] + 1$. Then we process the next chunk at the right. We continue until we reach the fictitious pivot.

Note that, if the element at position pos is originally a pivot, we extract it from S (by moving every pivot above it in the stack one position towards the bottom) and go back to the previous pivot, so we always have a pivot at a position greater than pos. Thus, extracting a pivot effectively merges the two chunks at the left and right of the removed pivot.

An application of operation **delete** is to implement operation **extractMin** by calling **delete**(0). This way we obtain a quickheap version that does not slide on the array $heap$. In this case we do not need the dictionary, as we want to delete the element at position zero. Yet, preliminary experiments show that this alternative is less efficient than the sliding one proposed above.

Decreasing and increasing a key can be done via a delete plus insert operations. Nevertheless, next we show a more efficient direct implementation.

Decreasing a key. Given a position pos of some element in the quickheap and a value $\delta \geq 0$, we change the priority of the element $heap[pos]$ to $heap[pos] - \delta$, and adjust its position in the quickheap so as to preserve the pivot invariant (Lemma 2.1). As we are decreasing the key, the modified element either stays in its current place or it moves chunk-wise towards position idx. Thus operation **decreaseKey** is similar to operation **insert**.

To decrease a key, we first need to find the chunk $pidx$ of the element to modify. If the element at position pos is a pivot, we extract it from S and go back to the previous pivot, so we always have a pivot at a position greater than pos.

Let $newValue = heap[pos] - \delta$ be the resulting value of the modified element. Once we have a pivot $pidx$ at a position greater than pos, we do the following. If we are working in the first chunk, that is $|S| = pidx + 1$, we update the element $heap[pos]$ to $newValue$ and we are done. Otherwise, we check whether $newValue$ is greater than or equal to the preceding pivot ($heap[S[pidx + 1]]$). If so, we update the element $heap[pos]$ to $newValue$ and we have finished. Else, we place the element at the right of the next pivot in the current position of the element. That is, we move the element $heap[S[pidx + 1] + 1]$ to position $heap[pos]$. As we have an empty space next to the pivot delimiting the preceding chunk, we start the pivot movement procedure from that chunk.

Increasing a key. Analogously, given a position pos of some element in the quickheap, and a value $\delta \geq 0$, this operation changes the value of the element $heap[pos]$ to $heap[pos] + \delta$, and adjusts its position in the quickheap so as to preserve the pivot invariant. As we are increasing the key, the modified element either stays in its current place or moves chunk-wise towards position $S[0]$. Thus, operation **increaseKey** is similar to operation **delete**, but without removing the element. Similarly to operations **decreaseKey** or **delete**, we first need to find the chunk $pidx$ of the element

to modify. If the element at position *pos* is a pivot, we remove it from the stack S and go back to the previous pivot, so we have a pivot in a position greater than *pos*.

4 Analysis of Quickheaps

In the following, we prove that quickheap operations cost $O(\log m)$ expected amortized time, where m is the maximum size of the quickheap. This analysis is based on a key observation: statistically, quickheaps exhibit an *exponentially-decreasing structure*, which means that the pivot positions form on average an exponentially decreasing sequence. We start by proving that exponential-decrease property. Then, we introduce the *potential debt method* for amortized analysis. Finally, exploiting the exponential-decrease property, we analyze quickheaps using the potential debt method.

4.1 The Quickheap's Exponential-Decrease Property

In this section we introduce a formal notion of the exponentially-decreasing structure of quickheaps. We show that this property is true at the beginning, and that it holds after extractions of minima, as well as insertions or deletions 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 and deletions cannot be arbitrary but uniformly distributed.

More precisely, our uniformity assumptions are stated as follows. When inserting a new element into a heap of $n-1$ elements, we assume that the rank of the new element in the existing set distributes uniformly in $[1, n]$. When deleting an element from a heap with $n+1$ elements, we assume each of the elements is chosen for deletion with uniform probability.

From now on, we consider that *array segments* are delimited by idx and the cell just before each pivot position $S[pidx]$ ($heap[idx, S[pidx]-1]$, thus segments overlap), and *array chunks* are composed by the elements between two consecutive pivot positions ($heap[S[pidx]+1, S[pidx-1]-1]$) or between idx and the cell preceding the pivot on top of S ($heap[idx, S.\mathbf{top}()-1]$). We call $heap[idx, S.\mathbf{top}()-1]$ the first chunk, and $heap[S[1]+1, S[0]-1]$ the last chunk. Analogously, we call $heap[idx, S.\mathbf{top}()-1]$ the first segment, and $heap[idx, S[0]-1]$ 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. Figure 5 illustrates this.

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

The exponential-descrease property is formally defined as follows:

Definition 4.1 (quickheap's exponential-decrease property)**.** *The probability that the pivot of each array segment $heap[idx, S[pidx]-1]$ is large in its segment is smaller than or equal to $\frac{1}{2}$. That is, for all the segments $\mathbb{P}(pivot\ is\ large) \leq \frac{1}{2}$.*

We prove the property by analyzing each individual element in isolation, and considering the operations that affect it. So from now on we refer to any individual segment and analyze its

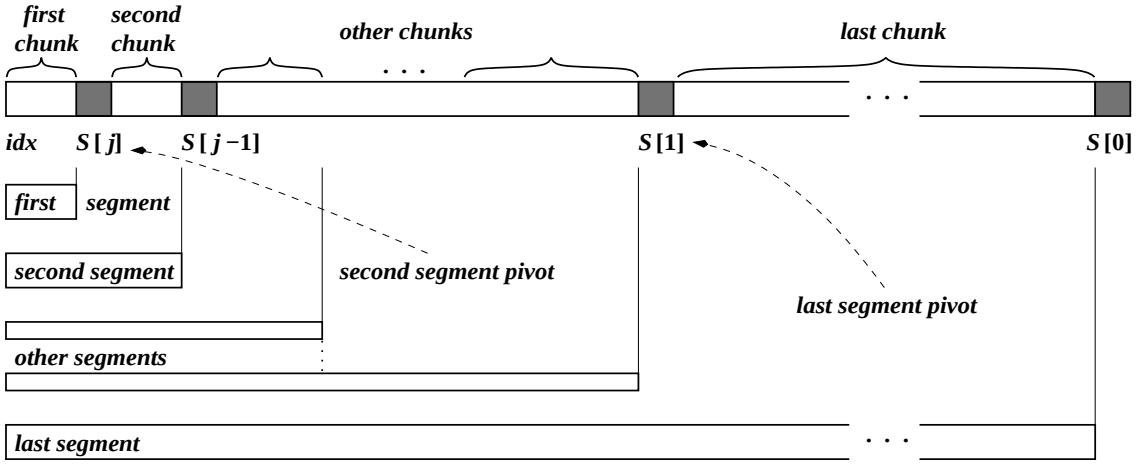


Figure 5: Segments and chunks of a quickheap.

evolution (note segments contain each other, but we can still analyze each of them regardless of the rest). 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 the segment, of size n, is the pivot of the segment after the j-th operation ($\mathbb{P}_{i,j,n} = 0$ outside bounds). In the following we prove by induction on j that $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$, for all j, n and $2 \leq i \leq n$, after performing any sequence of operations **insert**, **delete**, **findMin** and **extractMin**. That is, the probability of the element at cell i being the pivot is non-increasing from left to right. Later, we use this to prove the exponential-decrease property and some consequences of it.

Note that the pivot appears for the first time in the segment when it is the shortest one and operations **extractMin** or **findMin** partition it. Note also that, just after the segment is partitioned, the probabilities are $\mathbb{P}_{i,0,n} = \frac{1}{n}$, as the pivot is chosen following a uniform distribution, so we have proved the base case.

Lemma 4.1. *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 position uniformly chosen in $[1, n]$.*

Proof. 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 at any of the first $i-1$ positions $1, \ldots, i-1$, so the pivot moved to the right; or the pivot was already at cell i and we have inserted x at any of the last $n-i$ positions $i+1, \ldots, n$. So, we have the recurrence of Eq. (1).

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

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. (2), whose left side corresponds to the recurrence of $\mathbb{P}_{i,j,n}$.

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

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. (3), where in the right side we have the recurrence for $\mathbb{P}_{i-1,j,n}$.

$$\mathbb{P}_{i,j,n} \quad \leq \quad \mathbb{P}_{i-2,j-1,n-1}\frac{i-2}{n} + \mathbb{P}_{i-1,j-1,n-1}\frac{n+1-i}{n} \quad = \quad \mathbb{P}_{i-1,j,n} \tag{3}$$

With respect to $i = 2$, note that the term $\frac{i-2}{n}$ from Eqs. (2) and (3) vanishes, so the replacement made for $i > 2$ holds anyway. 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. (1) evaluated for $i = 1$. □

Lemma 4.2. *For each segment, the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after deleting an element at a position chosen uniformly from $[1, n+1]$.*

Proof. Suppose that after the $(j-1)$-th operation the segment has $n+1$ elements. As we delete an element in the j-th operation, the resulting segment contains n elements.

We start by proving the property when the deleted element is not a pivot. The probability that after the deletion the pivot p is at cell i depends on whether p was at cell $i+1$ and we delete an element from positions $1, \ldots, i$, so the pivot moved to the left; or the pivot was already at cell i, and we have deleted from the last $n+1-i$ elements $i+1, \ldots, n+1$. So, we have the recurrence of Eq. (4).

$$\mathbb{P}_{i,j,n} \quad = \quad \mathbb{P}_{i+1,j-1,n+1}\frac{i}{n+1} \quad + \quad \mathbb{P}_{i,j-1,n+1}\frac{n+1-i}{n+1} \tag{4}$$

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 $\frac{n+2-i}{n+1}$, adding $\mathbb{P}_{i,j-1,n+1}\frac{i-1}{n+1}$ and rearranging terms we obtain the inequality of Eq. (5), whose right side corresponds to the recurrence of $\mathbb{P}_{i-1,j,n}$.

$$\mathbb{P}_{i,j-1,n+1}\frac{i}{n+1} + \mathbb{P}_{i,j-1,n+1}\frac{n+1-i}{n+1} \quad \leq \quad \mathbb{P}_{i,j-1,n+1}\frac{i-1}{n+1} + \mathbb{P}_{i-1,j-1,n+1}\frac{n+2-i}{n+1} \tag{5}$$

By the inductive hypothesis again, $\mathbb{P}_{i+1,j-1,n+1} \leq \mathbb{P}_{i,j-1,n+1}$, so we can replace the first term above to obtain the inequality of Eq. (6), where in the left side we have the recurrence for $\mathbb{P}_{i,j,n}$. On the right we have $\mathbb{P}_{i-1,j,n}$.

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

In the case of deleting a pivot p we have the following. If we delete the pivot on top of S, then the first and the second chunk get merged and the lemma does not apply to the (new) first segment because it has no pivot.

Otherwise, we must have (at least) two pivots p_l and p_r at the left and right of p. Let pos_l, pos and pos_r be the positions of the pivots p_l, p, p_r before deleting p, respectively, as it is shown in Figure 6. Note that p_l and p are pivots of segments $heap[idx, pos-1]$ and $heap[idx, pos_r-1]$ with n' and n elements ($n' < n$), respectively.

Once we delete pivot p, the segment $heap[idx, pos-1]$ is "extended" to position $pos_r - 2$ (as we have one cell less). As the $n - n' - 1$ new elements in the extended segment were outside of the old segment $heap[idx, pos-1]$, they cannot be the pivot in the extended segment. On the other hand, the probabilities of the old segment elements hold in the new extended segment. Therefore,

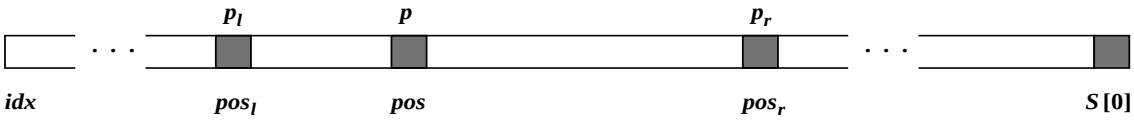


Figure 6: Deleting an inner pivot of a quickheap.

for each $idx \leq i < pos$, $\mathbb{P}_{i,j,n} = \mathbb{P}_{i,j-1,n'}$, and for each $pos \leq i < pos_r - 2$, $\mathbb{P}_{i,j,n} = 0$. Thus the invariant is maintained. □

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 a pivot, where all the probabilities are $\mathbb{P}_{i,0,n} = \frac{1}{n}$. The next lemma shows that the property also holds after taking the minimum.

Lemma 4.3. *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 true by vacuity. Else, within each segment probabilities change as follows: $\mathbb{P}_{i,j,n} = \mathbb{P}_{i+1,j-1,n+1}\frac{n+1}{n}$. □

Finally, we are ready to prove the quickheap's exponential-decrease property.

Theorem 4.1 (quickheap's exponential-decrease property)**.** *Given a segment $heap[idx, S[pidx]-1]$, the probability that its pivot is large is smaller than or equal to $\frac{1}{2}$, that is, $\mathbb{P}(pivot\ is\ large) \leq \frac{1}{2}$.*

Proof. When the segment is created, all the probabilities are $\mathbb{P}_{i,j,n} = \frac{1}{n}$. Lemmas 4.1 to 4.3 guarantee that the property $\mathbb{P}_{i,j,n} \leq \mathbb{P}_{i-1,j,n}$ for $i \geq 2$ is preserved after inserting or deleting elements, or taking the minimum. So, the property is preserved after any sequence of operations **insert**, **delete**, **findMin** and **extractMin**. Therefore, adding up the probabilities $\mathbb{P}_{i,j,n}$ for the large elements, that is, for the $\left(\left\lceil \frac{n}{2} \right\rceil + 1\right)$-th to the n-th element, we obtain that $\mathbb{P}(pivot\ is\ large) = \sum_{i=\left\lceil \frac{n}{2} \right\rceil + 1}^{n} \mathbb{P}_{i,j,n} \leq \frac{1}{2}$. □

In the following, we use the exponential-decrease 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 4.4. *The expected value of the height $\mathcal{H}$ of stack S is $O(\log m)$.*

Proof. Notice that the number $\mathcal{H}$ of pivots in the stack is monotonically nondecreasing with m. Let us make some pessimistic simplifications (that is, leading to larger $\mathcal{H}$). 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: $\mathcal{H} = T(m) = 1 + \frac{1}{2}T(m-1) + \frac{1}{2}T\big(\lfloor \frac{m}{2} \rfloor\big)$, $T(1) = 1$. Once again, using the monotonicity on the number of pivots, the recurrence is simplified to $T(m) \leq 1 + \frac{1}{2}T(m) + \frac{1}{2}T\big(\frac{m}{2}\big)$, which can be rewritten as $T(m) \leq 2 + T\big(\frac{m}{2}\big) \leq \ldots \leq 2j + T\big(\frac{m}{2^j}\big)$. 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 4.5. *The expected value of the sum of the sizes of array segments is $\Theta(m)$.*

Proof. Using the same reasoning of Lemma 4.4, but considering that when partitioning the largest segment has m elements, we write the following recurrence: $T(m) = m + \frac{1}{2}T(m-1) + \frac{1}{2}T\big(\lfloor \frac{m}{2} \rfloor\big)$, $T(1) = 0$. Using the monotonicity of $T(m)$ (which also holds in this case) the recurrence is simplified to $T(m) \leq m + \frac{1}{2}T(m) + \frac{1}{2}T\big(\frac{m}{2}\big)$, which can be rewritten as $T(m) \leq 2m + T\big(\frac{m}{2}\big) \leq \ldots \leq 2m + m + \frac{m}{2} + \frac{m}{2^2} + \ldots + \frac{m}{2^{j-2}} + T\big(\frac{m}{2^j}\big)$. 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 carry out the amortized analysis of quickheaps we use a slight variation of the potential method ([36] and [11, Chapter 17]), which we call the *potential debt method*.

In the potential method, the idea is to determine an amortized cost for each operation type. The potential method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on the data structure. The sum of all the prepaid credit is called the *potential* of the data structure. The potential is used later in the sequence to pay for operations that are charged less than what they actually cost.

Instead, in the case of the potential debt method, 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 from 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

$$\widetilde{c_i} \;=\; c_i - \Phi(D_i) + \Phi(D_{i-1}) \;. \tag{7}$$

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

$$\sum_{i=1}^{N} \widetilde{c_i} \;=\; \sum_{i=1}^{N} (c_i - \Phi(D_i) + \Phi(D_{i-1})) \;=\; \sum_{i=1}^{N} c_i - \Phi(D_N) + \Phi(D_0)\ . \tag{8}$$

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 the 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 $\sum_{i=1}^{N} c_i$. That is, at the end we share the debt among the operations. Thus, in Eq. (9) 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 $\Phi(D_N)$.

$$\widehat{c_i} = \widetilde{c_i} + \frac{\Phi(D_N) - \Phi(D_0)}{N} = c_i - \Phi(D_i) + \Phi(D_{i-1}) + \frac{\Phi(D_N) - \Phi(D_0)}{N} \tag{9}$$

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 Expected-case Amortized Analysis of Quickheaps

In this section, we consider that we operate over a quickheap qh with m elements within *heap* and a pivot stack S of expected height $\mathcal{H} = O(\log m)$, see Lemma 4.4.

We define the quickheap potential debt function as twice the sum of the sizes of the segments delimited by idx and pivots in $S[0]$ to $S[\mathcal{H}]$. Eq. (10) shows the potential function $\Phi(qh)$, which is illustrated in Figure 7.

$$\Phi(qh) = 2 \cdot \sum_{i=0}^{\mathcal{H}} (S[i] - idx) = \Theta(m) \text{ expected, by Lemma 4.5} \tag{10}$$

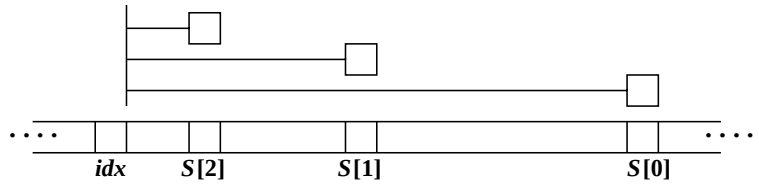


Figure 7: The quickheap potential debt function is computed as twice the sum of the lengths of the segments (drawn with lines) delimited by idx and pivots in $S[0]$ to $S[\mathcal{H}]$. In the figure, $\Phi(qh) = 2 \cdot (S[0] + S[1] + S[2] - 3idx)$.

Thus, the potential debt of an empty quickheap $\Phi(qh_0)$ is zero, and the expected potential debt of an m-elements quickheap is $\Theta(m)$, see Lemma 4.5. 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 within the quickheap. Therefore, in the case of quickheaps, the term $\frac{\Phi(qh_N) - \Phi(qh_0)}{N}$ is $O(1)$ expected. 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 difference of the potential debt $\Phi(qh_{i-1}) - \Phi(qh_i)(< 0)$ depends on how many segments are extended (by one cell) due to the insertion (recall that segments contain each other, so one insertion extends several segments). Note that for each key comparison, we extend one segment —which increases by 2 the potential debt—, yet the final key comparison might extend one more segment (if it is the shortest one). Thus, it holds $c_i - \Phi(qh_i) + \Phi(qh_{i-1}) \leq 0$, which means that all the cost is absorbed by the increase in the potential debt.

However, we can prove that also the expected (individual) cost of operation **insert** is $O(1)$. When inserting an element, we always extend the last segment. Later, with probability $\mathbb{P}_1 \geq \frac{1}{2}$ the position of the inserted element is greater than the position of the pivot $S[1]$ —that is, the element is inserted at the right of the pivot $S[1]$— (from Theorem 4.1), in which case we stop. If not, we compare the pivot of the second last segment, and once again, with probability $\mathbb{P}_2 \geq \frac{1}{2}$ the element is inserted at the right of the pivot $S[2]$, in which case we stop. Else, we compare the third pivot, and this goes on until we stop expanding segments. Thus, the expected number of key comparisons is $1 + (1 - \mathbb{P}_1)(1 + (1 - \mathbb{P}_2)(1 + (1 - \mathbb{P}_3)(1 + \ldots)))$. This sum is upper bounded, by taking the lowest value of $\mathbb{P}_i = \frac{1}{2}$, to $1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \frac{1}{2}\left(1 + \ldots\right)\right)\right) = O(1)$.

Operation delete. The decrease of the potential debt $\Phi(qh_{i-1}) - \Phi(qh_i)(> 0)$ depends on how many segments are contracted (by one cell) due to the deletion. Note that it is also possible to delete a whole segment if we remove a pivot.

The worst case of operation **delete** (without considering pivot deletions) arises when deleting an element in the first chunk. This implies to contract by one cell all the segments, which is implemented by moving all the pivots —whose expected number is $\mathcal{H}$— one cell to the left. So, the actual cost of moving pivots and elements is $\mathcal{H}$. On the other hand, the term $\Phi(qh_{i-1}) - \Phi(qh_i)$, which accounts for the potential decrease due to all the contracted segments, is $2\mathcal{H} + 2$. Thus, the amortized cost is $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1}) \leq 3\mathcal{H} + 2$. This is $O(\log m)$ expected, see Lemma 4.4.

If, instead, we remove a pivot, we also remove a whole segment, thus decreasing the potential debt. We can delete each of the pivots with probability $\frac{1}{m}$. Hence the average decrease in the potential debt is $2 \cdot \sum_{i=0}^{\mathcal{H}} \frac{1}{m}(S[i] - idx) = \frac{1}{m}\Phi(qh)$. As $\Phi(qh) = \Theta(m)$, $\frac{1}{m}\Phi(qh) = \Theta(1)$ expected.

Therefore, the potential debt decrease $\Phi(qh_{i-1}) - \Phi(qh_i)(> 0)$ due to segment contractions and segment deletions is $2\mathcal{H} + \Theta(1)$. Considering that every time we contract a segment, we perform $O(1)$ work in pivot and element movements, the expected amortized cost of operation **delete** on pivots is $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1}) \leq 3\mathcal{H} + \Theta(1) = O(\log m)$ expected.

Once again, we can prove that the individual cost of operation **delete** is actually $O(1)$ expected. We start by analyzing the deletion of non pivot elements with an argument similar to the one used in operation **insert**. When deleting an element, we always contract the last segment. Later, with probability $\mathbb{P}_1 \geq \frac{1}{2}$ the position of the deleted element is greater than the position of the pivot $S[1]$ (from Theorem 4.1), in which case we stop. If not, we contract the second last segment, and this goes on until we stop contracting segments. Thus, the expected cost due to segment contractions is $1 + (1 - \mathbb{P}_1)(1 + (1 - \mathbb{P}_2)(1 + (1 - \mathbb{P}_3)(1 + \ldots))) = O(1)$. Otherwise, if we delete a pivot, we have to add the cost of removing it from the stack, which is $O(\log m)$ expected. Yet, this occurs with probability $O(\frac{\log m}{m})$, thus overall we add up $o(1)$ expected time.

Creation of a quickheap. The amortized cost of constructing a quickheap from scratch is $O(1)$. Instead, if we create a quickheap from an array A of size m, the potential debt will be $2m$, regardless of whether we copy the array (at actual cost $c_i = O(m)$) or we take A as *heap* (at actual cost $c_i = O(1)$). In either case the amortized cost is $\widehat{c_i} = \Theta(m)$. This preserves correctness of our analysis even if are breaking the assumption of having more operations than elements, as the potential debt is accounting for all these elements.

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 s of the first segment. On the other hand, once we push the pivot, the potential debt increases by an amount which is twice the size s' of the new segment. As s' distributes uniformly over $[0, s]$, we have that its expected value is $\overline{s'} = s/2$. Thus we have that the expected amortized cost will be $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1}) = s - 2 \cdot \overline{s'} = 0$.

With respect to operation **takeMin**, its actual cost is $O(1)$, and the potential debt decreases by $2\mathcal{H} + 2$, as all the segments are reduced by one cell after taking the minimum. As the expected value of $\mathcal{H}$ is $O(\log m)$ (see Lemma 4.4), the expected amortized cost of operation **takeMin** (and **extractMin**) is $O(\log m)$. Note also that **takeMin** can remove one segment (of length 1) if it happens to remove a pivot. This causes a change of -2 in the potential debt, which adds $O(1)$ to the amortized cost.

Operation findMin. 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 $heap[idx]$ (with constant cost). Then, the amortized cost of operation **findMin** is $O(1)$.

Operation increaseKey. This operation is special in the sense that it only moves elements to the rear of the quickheap. (We cannot use the fact that **increaseKey** can be seen as a sequence of two single operations **delete** and **insert**, as they are not independent: even though the deletion occurs at random, the following insertion does not.) Fortunately, **increaseKey** preserves Theorem 4.1. In fact, when we increase the key of some element the involved pivots either stay in their cells or they move to the left. So the probability that the pivot is large holds or diminishes.

Therefore, in order to analyze it, we can still use the argument that in the worst case we increase the key of an element in the first chunk; which implies at most $\mathcal{H}$ movements of elements and pivots. So the actual cost of operation **increaseKey** is $O(\log m)$ expected. On the other hand, the potential variation $\Phi(qh_{i-1}) - \Phi(qh_i)(> 0)$ depends on how many segments are contracted when moving the modified element, and is at most $2\mathcal{H} + 2$. Thus, the amortized cost is $\widehat{c_i} = c_i - \Phi(qh_i) + \Phi(qh_{i-1}) = 3\mathcal{H} + 2$. This is $O(\log m)$ expected, see Lemma 4.4.

Operation decreaseKey. This is also a special operation. Regrettably, it does not preserve Theorem 4.1. In fact, each time we decrease some key the involved pivots either stay in their cells or they move to the right. Thus, when we perform operation **decreaseKey** the probability of a

pivot being large holds or increases, so it could go beyond $\frac{1}{2}$. However, in practice, this operation performs reasonably well as it is shown in Sections 7.2 and 7.4.

To sum up, we have proved the following theorem.

Theorem 4.2 (quickheap's complexity)**.** *The expected amortized cost of any sequence of m operations* **insert***,* **delete***,* **findMin***,* **extractMin** *and* **increaseKey** *over an initially empty quickheap is $O(\log m)$ per operation, assuming that insertions and deletions occur at uniformly random positions. Actually, the individual expected cost of operations* **insert** *and* **delete** *is $O(1)$. All operation costs must be multiplied by the cost of updating the element positions in the dictionary in case* **delete** *or* **increaseKey** *are to be supported.* □

5 Quickheaps in External Memory

Quickheaps are excellent candidates for secondary memory, as they exhibit a local access pattern. Since our algorithms are unaware of the disk transfers, the result of blindly using them on disk is cache-oblivious. Cache obliviousness [18, 7] means that the algorithm is designed for the RAM model of computation but analyzed under the I/O model, assuming an optimal offline page replacement strategy. (This is not unrealistic because LRU is 2-competitive if run over a memory of twice the original size [33].) Cache-oblivious algorithms for secondary memory are easier to program than their cache-aware counterparts, and adapt better to arbitrary memory hierarchies.

The resulting external quickheap allows performing operations **insert**, **findMin** and **extractMin** in expected amortized I/O cost $O((1/B)\log(m/M))$, where B is the block size, M is the total available main memory, and m is the maximum heap size along the process. This result is close to the lower bound [7], $\Theta((1/B)\log_{M/B}(m/B))$, for cache-oblivious sorting. It relies on the assumption $M = \Omega(B \log m)$, which is not standard, yet reasonable in practice. Although there exist optimal cache-oblivious priority queues [2, 20, 25, 14, 8, 6, 32, 13], quickheaps are, again, a simple and practical alternative.

5.1 Adapting Quickheap Operations to External Memory

When considering the basic priority queue operations one can realize that quickheaps exhibit high locality of reference: First, the stack S is small and accessed sequentially. Second, 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. Third, **IQS** sequentially accesses the elements of the first chunk. Thus, under the cache-oblivious assumption, we will consider that our page replacement strategy keeps in main memory: (*i*) the stack S and integers *idx* and *capacity*; (*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 two disk blocks. According to Lemma 4.4, all this requires on average to hold $\Theta(B \log m) \leq M$ 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*).

Despite we could manage the element positions in a dictionary, in the case of external quickheaps we do not consider operations **delete**, **increaseKey**, and **decreaseKey**. This is because the accesses to the dictionary are not necessarily sequential. Thus, if we cannot handle the dictionary in main memory, the I/O cost of updating it could be excessive, preventing us to efficiently implement

all the quickheap operations. Note that we could implement **extractMin** through calling **delete**(0) (in order to implement a non-sliding heap), since in this case we do not need the dictionary. Yet, in this case we have to keep in main memory two blocks per pivot: (1) the block containing the current position of the pivot in *heap*, and (2) the one that previously contained it. This way, even though a pivot moves forward and backward from one page towards the other (note that both pages are consecutive), the I/Os are bounded. However, in this case we have less memory for the prefix.

5.2 Analysis of External Memory Quickheaps

We first show a simple approach that keeps in main memory the first two disk blocks of the array. Later, we analyze the effect of a larger prefix of disk blocks cached in internal memory.

5.2.1 A Simple Approach

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 the disk page $j + 1$ in main memory. From then, at least B increments of pivot position i are necessary before loading another disk page due to that pivot. Therefore, as there are $\mathcal{H}$ (true) pivots, the amortized cost of an element insertion is $\mathcal{H}/B$. According to the results of the previous section, this is $O(\log(m)/B)$ expected.

Operations **findMin** and **extractMin** 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 traversed area spans $1 + \lceil \ell/B \rceil$ disk blocks. As we have in main memory the first two blocks of $heap[idx, N]$, we have to load at most $1 + \lceil \ell/B \rceil - 2 \leq \ell/B$ disk blocks. On the other hand, the CPU cost of such traversal is $\Theta(\ell)$. Hence, each comparison made has an amortized I/O cost of $O(1/B)$. According to the previous section, all those traversals cost $O(\log m)$ amortized expected comparisons. Hence, the amortized I/O cost is $O(\log(m)/B)$ expected. Maintaining this prefix of a given size in main memory is easily done in $O(1/B)$ amortized time per operation, since idx grows by one upon calls to **extractMin**.

Overall, we achieve $O(\log(m)/B)$ expected amortized I/O cost. We now get better bounds by considering an arbitrary size $\Theta(M)$ for the *heap*'s prefix cached in internal memory.

5.2.2 Considering the Effect of the Prefix

Let us consider that we have $M' = \Theta(M)$ cells of main memory to store a prefix of array *heap*. Thus, accessing these cells is I/O-free, both when moving pivots or when partitioning segments.

We start by using the potential debt method to analyze the number of key comparisons computed for elements *outside* the prefix for each quickheap operation. Then, we will derive the I/O cost from those values by reusing the arguments of Section 5.2.1. We define the potential function as follows:

$$\Psi(qh) \quad = \quad \sum_{i=0}^{\mathcal{H}} \max(2\,(S[i] - idx) - M', 0) \quad = \quad 2 \cdot \sum_{i=0}^{\mathcal{H}} \max((S[i] - idx) - M'/2, 0).$$

In this case, the potential debt $\Psi(qh)$ of an empty quickheap $\Psi(qh_0)$ is zero, the potential debt $\Psi(qh)$ of a small heap (that is, where $m \leq M'/2$) is also zero, and when the heap is big enough ($m \gg M'$), the expected potential debt $\Psi(qh)$ of an m-element quickheap is $\Theta(m)$, see Lemma 4.5.

Once again, 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 within the quickheap, and write the amortized costs directly as $\widehat{c_i} = c_i - \Psi(qh_i) + \Psi(qh_{i-1})$.

Due to the exponential-decrease property (Theorem 4.1), on average at least the first $\log_2 M' = \Theta(\log M)$ pivots will be in main memory, and therefore accessing them will be I/O-free. As a consequence, there are only $O(\log m - \log M) = O(\log(m/M))$ pivots outside the prefix. Similarly, $O(\log(m/M))$ segments contribute to the potential debt.

Operation insert. The potential debt $\Psi(qh)$ grows by 2 for each segment larger than $M'/2$ worked on. The real cost is one unit per segment larger than M' worked on. Hence it holds $c_i - \Psi(qh_i) + \Psi(qh_{i-1}) \leq 0$.

Operation extractMin. We split operation **extractMin** into a sequence of zero or more calls to **pivoting**, and a single call to **takeMin** (see Section 4.1). Note that **pivoting** is I/O-free over the first M' elements of *heap* (as they are cached in main memory).

Each time we call operation **pivoting**, we only consider the key comparisons computed outside the prefix. Say that the first segment is of size s, and a new one of size s' is created after the partitioning. Then the actual number of accesses is $\max(s - M', 0)$, while the increase in the potential debt is $\max(2s' - M', 0)$. As s' distributes uniformly in $[0, s-1]$, assuming $s > M'$ we have that $\overline{\max(2s' - M', 0)} = \frac{1}{s}\sum_{j=M'/2}^{s-1}(2j - M') = (s - M'/2)(s - M'/2 - 1)/s$. Hence the expected amortized cost is (1) zero if $s \leq M'$ and $s' \leq M'/2$, (2) negative if $s \leq M'$ and $s' > M'/2$, and otherwise (3) $\widehat{c_i} = (s - M') - (s - M'/2)(s - M'/2 - 1)/s = (s - M') - (s - M'/2)^2/s + O(1)$. By calling $x = M'/(2s) < 1/2$ and analyzing $((s - M') - (s - M'/2)^2/s)/s = (1 - 2x) - (1 - x)^2$ for $0 \leq x \leq 1/2$, we see that $\widehat{c_i} = O(1)$.

With respect to operation **takeMin**, it takes no comparison outside the cached prefix. On the other hand, the potential debt $\Psi(qh)$ decreases by $O(\log(m/M))$, as all the segments considered in the potential are reduced by one cell after taking the minimum. Therefore, the amortized cost of operation **takeMin** is $O(\log(m/M))$.

Operation findMin. We consider that operation **findMin** is a sequence of as many calls to operation **pivoting** as we need to push *idx* in stack S (with amortized cost zero) and later return element $heap[idx]$ (also with cost zero). Therefore, the amortized cost of **findMin** is $O(1)$.

Creation of a quickheap. The amortized cost of constructing a quickheap on disk from scratch is $O(1)$. Instead, the amortized cost of constructing a quickheap on disk from an array A of size m is $O(m)$, just as in Section 4.3. Note that the potential debt $\Psi(qh)$ is $\max(2m - M', 0)$.

Obtaining the I/O costs. Up to this point we have computed the amortized number of key comparisons performed by the quickheap operations outside the cached prefix. Thus, to compute

the amortized I/O costs of quickheap operations, we have to take into account how many of those can produce an additional I/O. According to the analysis of Section 5.2.1, there are two cases:

1. When moving a pivot from a disk page which is already in memory towards another page which is not, at least B further pivot movements are necessary before loading another disk page due to that pivot. The additional movement of the element beside the pivot is I/O-free, as when the element is moved, both source and target pages reside in main memory.

2. The access to elements inside the segments is sequential. So, the work performed by both element insertions or deletions and operation **partition** is amortized among B consecutive disk accesses. For this to remain true we need, just as in Section 5.2.1, that at least two blocks of the prefix are cached, that is, $M' \geq 2B$.

Therefore we have proved the following theorem.

Theorem 5.1 (external quickheap's complexity)**.** *If the* Quickheap *is operated in external memory using an optimal page replacement strategy and holding $M = \Omega(B \log m)$ integers in main memory, where B is the disk block size and m is the maximum heap size along the process; then the expected amortized I/O cost of any sequence of m operations* **insert***,* **findMin***, and* **extractMin** *over an initially empty quickheap is $O((1/B)\log(m/M))$ per operation, assuming that insertions and deletions occur at uniformly random positions.* □

6 Boosting the MST Construction

As a direct application of **IQS**, we use it to implement Kruskal's MST algorithm. Later, we use **QH**s to implement Prim's MST algorithm. The solutions obtained are competitive with the best current implementations, as we show in Section 7.4.

IQS-based implementation of Kruskal's MST algorithm. In our Kruskal's MST variant, we use **IQS** in order to incrementally sort E. The expected number of pivots we store in S is $O(\log m)$ (Lemma 4.4). Considering that in the general case we review m' edges, the expected cost of our Kruskal variant is $O(m + m' \log n)$ (Theorem 2.1). On random graphs [21, p. 349], we expect to review $m' = \frac{1}{2} n \ln n + O(n)$ edges, so the expected complexity of our Kruskal variant becomes $O(m + n \log^2 n)$, just as Kruskal's algorithm with demand sorting.

Quickheap-based implementation of Prim's MST algorithm. We use a quickheap qh in order to find the node u^* with the minimum connecting cost to the growing MST. Next, we check whether we update the values of *cost* for each u^*'s neighbor, and for these nodes we update their values in qh. For the sake of a fast access to the elements within the quickheap, we have to augment the quickheap structure with a dictionary managing the positions of the elements.

Each call to **insert** and **extactMin** uses $O(1)$ and $O(\log n)$ expected amortized time, respectively. Thus, the n calls to **insert** and **extractMin** use $O(n)$ and $O(n \log n)$ expected time, respectively. Finally, we perform at most m calls to operation **decreaseKey**. Regrettably, we cannot prove an upper bound for operation **decreaseKey**. However, our experimental results suggest that operation **decreaseKey** behaves roughly as $O(\log n)$, so the whole process costs $O(m \log n)$.

We have tested our Prim variant on graphs with random weights, which is a case where we can improve the obtained bound. Note that we only call operation **decreaseKey** when the new cost $weight_{u^*,v}$ is smaller than $cost_v$. Considering graphs with random weights, for each node, the probability of a fresh random edge being smaller than the current minimum after reviewing k edges incident on it is $\frac{1}{k}$. The number of times we find a smaller cost obeys the recurrence $T(1) = 1$ (the base case, when we find v the first time), and $T(k) = T(k-1) + \frac{1}{k} = \ldots = H_k = O(\log k)$. As each node has $\frac{m}{n}$ neighbors on average and for the concavity of the logarithm, we expect to find $O\left(\log \frac{m}{n}\right)$ minima per node. So, adding for the n nodes, we expect to call $O\left(n \log \frac{m}{n}\right)$ times operation **decreaseKey**.

Assuming that each call to **decreaseKey** has cost $O(\log n)$, we conjecture that this accounts for a total $O\left(n \log n \log \frac{m}{n}\right)$ expected time, making up a conjectured $O\left(m + n \log n \log \frac{m}{n}\right)$ expected amortized time for our Prim variant on graphs with random weights.

7 Experimental Results

We ran four experimental series. In the first we compare **IQS** with other alternatives. In the second we study the empirical behavior of **QH**s. In the third we study the behavior of **QH**s in secondary memory. Finally, in the fourth we evaluate our MST variants. The experiments were run on an Intel Pentium 4 of 3 GHz, 4 GB of RAM and local disk, running Gentoo Linux with kernel version 2.6.13. The algorithms were coded in C++, and compiled with `g++` version 3.3.6 optimized with `-O3`. For each experimental datum shown, we averaged over 50 repetitions. The weighted least square fittings were performed with R [37]. In order to illustrate the precision of our fittings, we also show the average percent error of residuals with respect to real values $\left(\left|\frac{y-\hat{y}}{y}\right| 100\%\right)$ for fittings belonging to around the largest 45% of values[3].

7.1 Evaluating IQS

For shortness we have called the classical Quickselect + Quicksort solution **QSS**, and the Partial Quicksort algorithm **PQS** [26]. We compared **IQS** with **PQS**, **QSS**, and two online approaches: the first based on classical heaps [42] (called **HEx**, implemented using the bottom-up deletion algorithm [41]), and the second based on sequence heaps [32] (called **SH**, we obtained the implementation of **SH** from `www.mpi-inf.mpg.de/~sanders/programs/spq/`). The idea is to verify that **IQS** is in practice a competitive algorithm for the *Partial Sorting* problem of finding the smallest elements in ascending order. For this sake, we use random permutations of non-repeated numbers uniformly distributed in $[0, m-1]$, for $m \in \left[10^5, 10^8\right]$, and we select the k first elements with $k = 2^j < m$, for $j \geq 10$. The selection is incremental for **IQS**, **HEx** and **SH**, and in one shot for **PQS** and **QSS**. We measure CPU time and, except for **SH**, the number of key comparisons.

[3]Our fittings are too pessimistic for small permutations or edge sets, so we intend to show that they are asymptotically good. In the first series we compute the percent error for permutations of length $m \in [10^7, 10^8]$ for all the k values, which is approximately 45.4% of the measures. In the second series we compute the percent error for sequences of $m \in [2^{22}, 2^{26}]$ elements, which is approximately 50% of the measures. In the fourth series we compute the percent error for edge densities in [16%, 100%] for all values of $|V|$, which is approximately 44.4% of the measures.

We summarize the experimental results in Figure 8. Figure 8(e) gives the least square fittings, where the percent errors of all the fittings are below 7%. The table shows that **IQS** CPU time performance is only 2.99% slower than that of its offline version **PQS**. The number of key comparisons is exactly the same, as we expected from Section 2. This is an extremely small price for permitting incremental sorting without knowing in advance how many elements we wish to retrieve, and shows that **IQS** is practical. Moreover, as the pivots in the stack help us reuse the partitioning work, our online **IQS** uses only 1.33% more CPU time and 4.20% fewer key comparisons than the offline **QSS**. This is illustrated in Figure 8(a), where the plots of **PQS**, **IQS** and **QSS** are superimposed. A detail of the previous is shown in Figure 8(b), where we see that **PQS** is the fastest algorithm when sorting a small fraction of the set, but **IQS** and **QSS** are rather close.

On the other hand, Table 8(e) shows large improvements with respect to online alternatives. According to the insertion and deletion strategy of sequence heaps, we compute its CPU time least squares fitting by noticing that we can split the experiment into two stages. The first inserts m random elements into the priority queue, and the second extracts the k smallest elements from it. Then, we obtain a simplified $O(k + m \log m)$ complexity model that shows that most of the work performed by **SH** comes from the insertion process. This also can be seen in Figure 8(a), by noticing that there is little difference between obtaining the first elements of the set, or the whole set. As a matter of fact, we note that if we want a small fraction of the sorted sequence, it is preferable to pay a lower insertion and a higher extraction cost (just like **IQS**) than to perform most of the work in the insertions and little in the extractions. With respect to the online **HEx**, we have the following. Even when it uses at most $2m$ key comparisons to heapify the array, and $\log_2 m + O(1)$ key comparisons on average to extract elements, the poor locality of reference generates numerous cache faults slowing down its performance. In fact, **HEx** uses 3.88 times more CPU time (see Figure 8(a)), even using 18.76% fewer key comparisons than **IQS** (see Figure 8(c)).

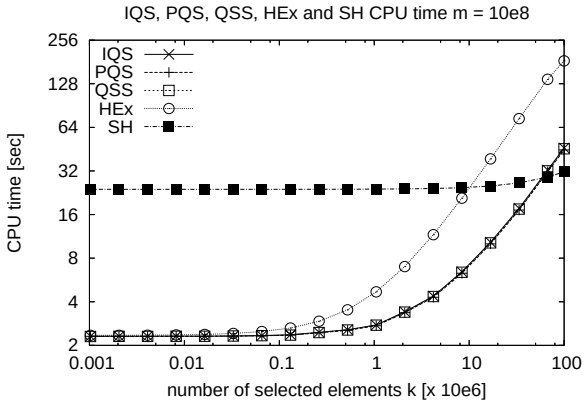
Finally, Figure 8(d) shows that, as k grows, **IQS**'s behavior changes as follows. When $k \leq 0.01m$, there is no difference in the time to obtain either of the first k elements, as the term m dominates the cost. When $0.01m < k \leq 0.04m$, there is a slight increase of both CPU time and key comparisons, that is, both terms m and $k \log k$ take part in the cost. Finally, when $0.04m < k \leq m$, term $k \log k$ leads the cost.

7.2 Evaluating Quickheaps

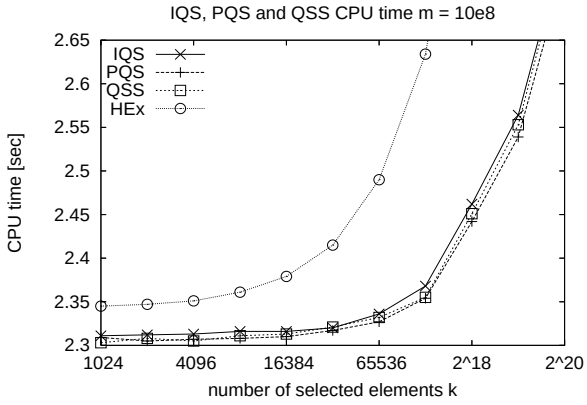
We start by studying the empirical performance of each operation in isolation. Next, we evaluate sequences of interleaved insertions and minimum extraction operations. Each experimental datum shown is averaged over 20 repetitions. For shortness we call operation **insert** *ins*, operation **extractMin** *del*, operation **decreaseKey** *dk* and operation **increaseKey** *ik*. (We have omitted the mixed sequence of *del* and *dk*/*ik* operations, since it is implicitly tested by **Prim3**, see Section 7.4.)

In these experiments, inserted elements follow a uniform distribution. On the other hand, updated keys are chosen uniformly, and increased or decreased by 10%.

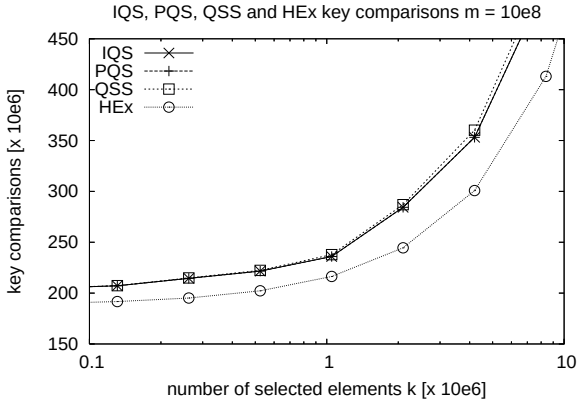
Isolated Quickheap Operations. We compare the empirical performance of quickheaps (or **QH**s for shortness) with binary heaps (**BH**) [42, 41] and with paring heaps (**PH**) [16]. We chose **BH**s because they are the canonical implementation of PQs, they are efficient and easy to program. We also chose **PH**s because they implement efficiently key update operations. Note that both binary



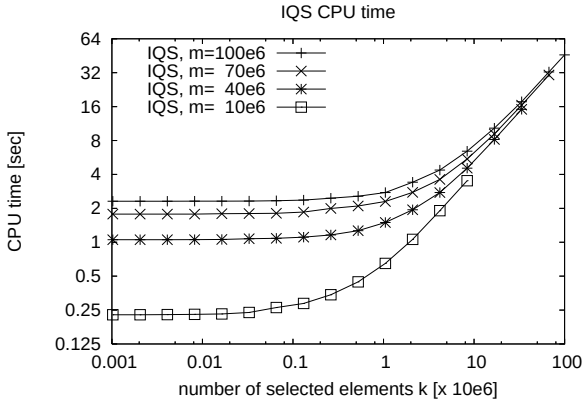
(a) CPU time for the five algorithms.



(b) Detail of CPU time for **IQS**, **PQS**, **QSS** and **HEx**.



(c) Detail of key comparisons for the four algorithms.



(d) **IQS** CPU time as a function of k and m.

(e) Weighted least square fittings.

	CPU time	Error	Key comparisons	Error
PQS	$25.8m + 16.9k \log_2 k$	6.77%	$2.14m + 1.23k \log_2 k$	5.54%
IQS	$25.8m + 17.4k \log_2 k$	6.82%	$2.14m + 1.23k \log_2 k$	5.54%
QSS	$25.8m + 17.2k \log_2 k$	6.81%	$2.14m + 1.29k \log_2 k$	5.53%
HEx	$23.9m + 67.9k \log_2 m$	6.11%	$1.90m + 0.97k \log_2 m$	1.20%
SH	$9.17m \log_2 m + 66.2k$	2.20%	—	—

Figure 8: Performance comparison between **IQS**, **PQS**, **QSS**, **HEx** and **SH** as a function of k and m. Note the logscales in the plots. In (a) and (b), CPU time for $m = 10^8$. In (c), key comparisons for $m = 10^8$. In (d), **IQS** CPU time as a function of k and m. In (e), weighted least square fittings. For **SH** we only compute the CPU time fitting. In this fitting CPU time is measured in nanoseconds.

and paring heaps are reported as the most efficient PQ implementations in practice [27]. We obtain the **PH** implementation, which includes operations **insert**, **extractMin** and **decreaseKey**, from `www.mpi-sb.mpg.de/~sanders/dfg/iMax.tgz`, as it is described by Katriel et al. [22].

This experiment consists in inserting m elements $\left(m \in \left[2^{17} \approx 0.13\text{e}6, 2^{26} \approx 67\text{e}6\right]\right)$, next performing i times the sequence *del-ik*^{i} or *del-dk*^{i}, for $i = 10000$, and later extracting the remaining $m - i$ elements. We measured separately the time for each different operation in this sequence. Note that we are testing a combined sequence of one minimum extraction and several element modifications. Recall that both **QH**s and **PH**s actually structure the heap upon minima extractions, so the idea of running an additional minimum extraction is to force both quickheaps and pairing heaps to organize their heap structure.

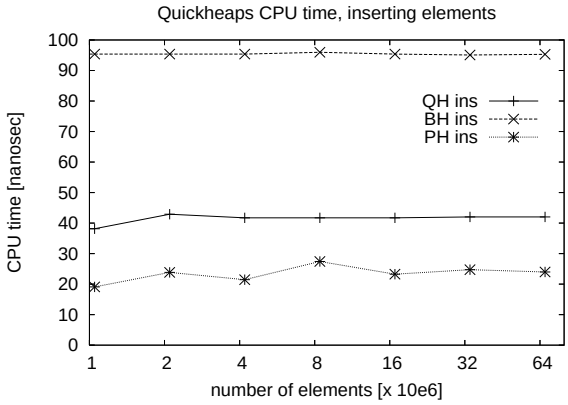
Figure 9 shows the results. Table 9(d) shows that the basic **QH** operations perform better than the corresponding **BH** ones. On the other hand, **PH**s perform better than **QH**s both when inserting elements and decreasing the key, although pairing heap operation **extractMin** is several times costlier than on the quickheap. Figure 9(a) shows that, as expected from our analysis, the cost of quickheap operation **insert** is constant, and it is the second best time (approximately twice the CPU time of **PH** and half the time of **BH**). Figure 9(b) shows that **QH**s have the best minimum extraction time, approximately one third of **BH**s and six times faster than **PH**s. Finally, Figure 9(c) shows that quickheap update-key operations perform slightly better than the respective ones for **BH**s. The plot also shows that **PH** decrease key operation performs slightly better than on quickheaps. Note that the model for **QH**s' decrease key operation was selected by observing the curve, as we could not analyze it.

Sequence of Insertions and Minimum Extractions. In order to show that quickheaps perform well under arbitrarily long sequences of insertions and minimum extractions we consider the following sequence of operations: $\left(ins - (del - ins)^i\right)^m \left(del - (ins - del)^i\right)^m$, for $i = 0$, 1, and 2. Note that for $i = 0$ we obtain algorithm heapsort [42].

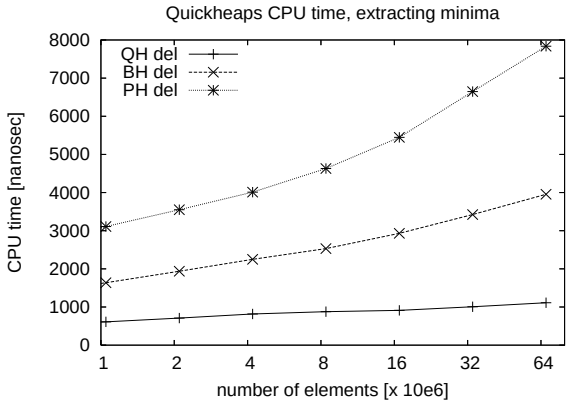
We compare **QH**s with binary and pairing heaps as before, but we also include sequence heaps [32], which are optimized for this type of sequences. Sequence heaps were excluded from other experiments because they do not implement **increaseKey** nor **decreaseKey**.

Figure 9 shows the results of this experiment. To improve readability, CPU times were divided by $m \log_2 m$. Figure 9(e) shows that **BH**s have the best performance for small sets, that is, up to $2^{18} \approx 262\text{e}3$ elements. This is expectable for two reasons: the bottom-up algorithm [41] strongly improves the binary heap performance, and the whole heap fits in cache memory. However, as the number of elements in the heap increases, numerous cache misses slow down the performance of binary heaps (these heaps are known to have poor cache locality, since an extraction touches an arbitrary element at each level of the heap, and the lower levels contain many elements). **QH**s, instead, are more cache-friendly as explained in the previous section. This is confirmed by the fact that quickheaps retain their good performance on large sets, being the fastest for more than $2^{19} \approx 524\text{e}3$ elements. In fact, for $m = 2^{26} \approx 67\text{e}6$ **BH**s perform 4.6 times slower than **QH**s, and **SH**s perform 1.6 times slower than **QH**s. On the other hand, the pairing heap is, by far, the slowest contestant in this experiment, as its operation **extractMin** is very costly.

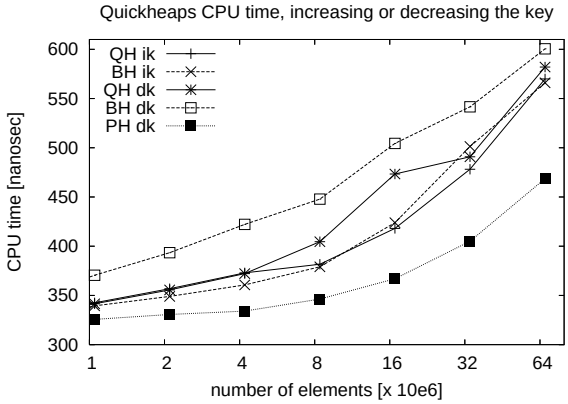
A similar behavior is appreciated for $i = 1$ (we have omitted this plot) and $i = 2$ (Figure 9(f)). For $i = 1$ **BH**s perform better for $m < 2^{19} \approx 524\text{e}3$ elements, then **SH**s are the fastest until $m < 2^{23} \approx 8.4\text{e}6$, and finally **QH**s take over. For $i = 2$ the best behaviour is that of **SH**s, closely followed by **QH**s. Binary heaps perform up to 2.4 times slower than quickheaps, and sequence heaps perform 8% faster than quickheaps. This is a modest difference considering that quickheaps



(a) Inserting elements.



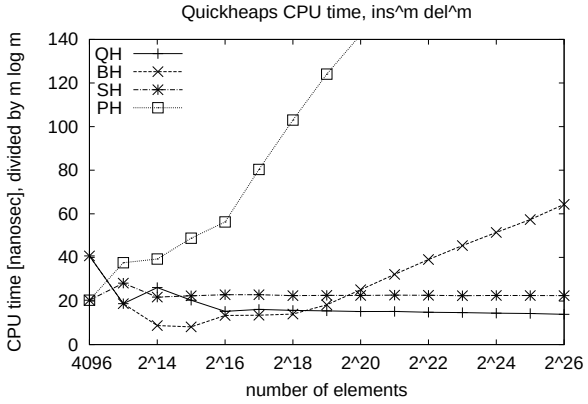
(b) Extracting minima.



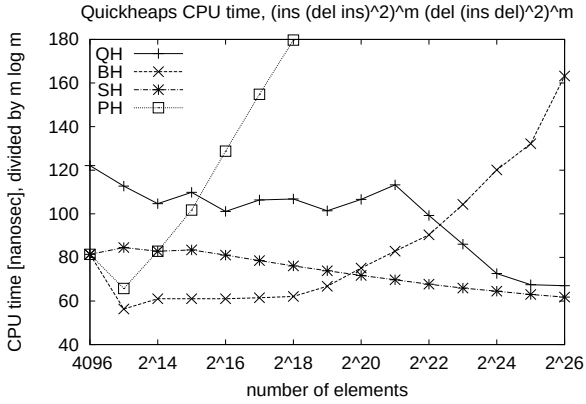
(c) Increasing and decreasing the key.

(d) Least square fittings for priority queue operations.

	Fitting	Error
$\mathbf{QH}_{ins}$	42	1.28%
$\mathbf{BH}_{ins}$	99	3.97%
$\mathbf{PH}_{ins}$	26	10.35%
$\mathbf{QH}_{del}$	$35 \log_2 m$	9.86%
$\mathbf{BH}_{del}$	$105 \log_2 m$	15.13%
$\mathbf{PH}_{del}$	$201 \log_2 m$	15.99%
$\mathbf{QH}_{ik}$	$18 \log_2 m$	8.06%
$\mathbf{BH}_{ik}$	$18 \log_2 m$	9.45%
$\mathbf{QH}_{dk}$	$18 \log_2 m$	8.88%
$\mathbf{BH}_{dk}$	$20 \log_2 m$	6.75%
$\mathbf{PH}_{dk}$	$16 \log_2 m$	5.39%



(e) $ins^m\ del^m$.



(f) $\left(ins - (del - ins)^2\right)^m\ \left(del - (ins - del)^2\right)^m$.

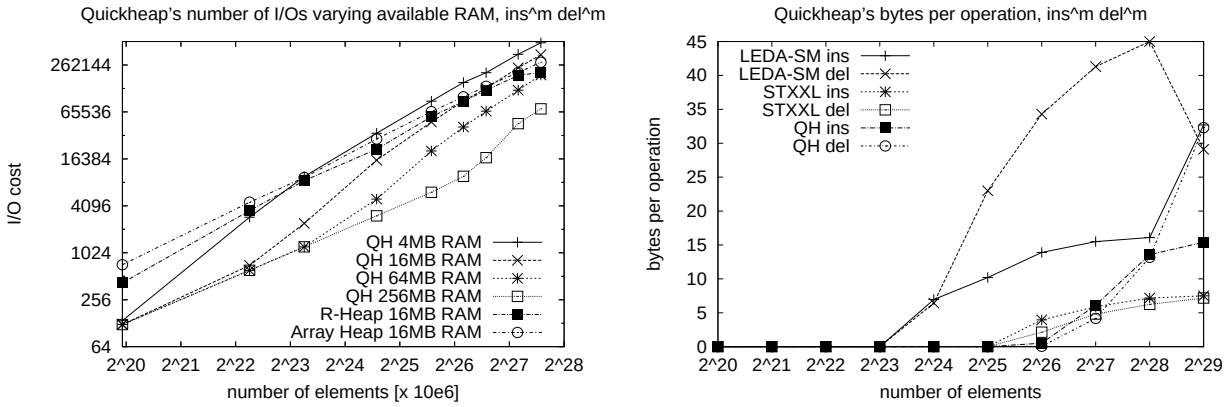
Figure 9: Evaluating Quickheaps. In (a), (b), and (c), performance of quickheap operations. In (d), least square fittings for priority queue operations, CPU time is measured in nanoseconds. In (e) and (f), performance of sequences interleaving operations *ins* and *del*. Note the logscales.

are much simpler to implement than sequence heaps. Once again, operation **extractMin** leaves pairing heaps out of the competitive alternatives for this experiment.

7.3 Evaluating External Memory Quickheaps

We carry out a brief experimental validation of **QH**s in external memory, comparing them with previous work [6, 13]. The results are shown in Figure 10. In order to reproduce the experimental setup of Brengel et al. [6], we measure quickheap I/O performance when executing the sequence $ins^m\ del^m$ for $m \in [1e6, 200e6]$, considering disk block size $B = 32$ KB, and keys of four bytes (that is, storing single integer values). As the authors have used $M = 16$ MB of main memory, we have varied the size of M from 1 to 256 MB. The inserted elements are chosen at random without repetition. The authors report the number of blocks read/written for different sequences of operations on the most promising secondary memory implementations they consider, namely, two-level radix heaps [1] (*R-Heaps*) and *Array-Heaps* [8].

Figure 10(a) shows that **QH**s achieve a performance slightly worse than R-Heaps when using just 4 MB of RAM. When using the same 16 MB, our structure performs 29% to 167% (that is, up to three times less) of the I/O accesses of R-Heaps, which only work if the priorities of the extracted elements form a nondecreasing sequence. If we consider the best alternative that works with no restriction (Array-Heaps), external **QH**s perform 17% (up to five times less) to 125% of their I/O accesses. We notice that, as the ratio $\frac{m}{M}$ grows, the performance of both R-Heaps and Array-Heaps improves upon external Quickheaps'. Other tests by Brengel et al. [6] are harder to reproduce[4]. Naturally, as more RAM is available, the I/O accesses consistently fall down. In fact, we notice the logarithmic dependence both on m and M (the plots are log-log), as expected from our analysis.



(a) I/O complexity ins^m del^m.

(b) Bytes per operation ins^m del^m.

Figure 10: In (a), I/O cost comparison for the sequence $ins^m\ del^m$. In (b), bytes accessed per operation for the sequence $ins^m\ del^m$ using 256MB RAM. Note the logscales in the plot.

[4]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.

To compare with the more recent experiments [13], we have measured bytes read/written per operation when executing the sequence $ins^m\ del^m$ for $m \in [2^{20}, 2^{29}]$, considering 512 MB of main memory, blocks of 2 MB, 8-byte elements (the key and four padding bytes), and keys drawn randomly from the range $[0, 2^{31})$. In this comparison we have used the results reported [13] both for **SH**s, currently the best external memory PQ, as implemented in the STXXL library [12, 13]; and the general-purpose Array-Heap implementation in LEDA-SM [6].

The results are shown in Figure 10(b). They are consistent with those of 10(a), in that **QH**s require fewer I/O accesses than Array-Heaps, both when inserting elements and extracting the minimum. On the other hand, **SH**s require fewer I/O accesses than **QH**s. For instance, **QH**s require 2.1 times the I/O accesses for operation **insert** and 4.5 for operation **extractMin**. These results show that, despite quickheaps are not the most efficient alternative for external memory priority queues, they behave decently in this scenario, while retaining simplicity and cache-obliviousness.

7.4 Evaluating the MST Construction

MST construction is one of the emblematic applications of incremental sorting and PQs. We compare our improved MST construction algorithms (see Section 6) with state-of-the-art alternatives. Our aim is not to study new MST algorithms but just to demonstrate the practical impact of our new fundamental contributions to existing classical algorithms. We use synthetic graphs with edges chosen at random, and with edge costs uniformly distributed in $[0, 1]$. We consider graphs with $|V| \in [2000; 26,000]$, and graph edge densities $\rho \in [0.5\%, 100\%]$, where $\rho = \frac{2m}{n(n-1)}100\%$.

For shortness we have called the basic Kruskal's MST algorithm **Kruskal1**, Kruskal's with demand sorting **Kruskal2**, our **IQS**-based Kruskal's **Kruskal3**, the basic Prim's MST algorithm **Prim1**[5], Prim's implemented with **PH**s **Prim2**, our Prim's implementation using **QH**s **Prim3** and the iMax algorithm **iMax** [22, 23]. We have used pairing heaps as they have shown good performance in this application [27, 23].

According to the experiments of Section 7.1, we preferred classical heaps using the bottom-up heuristic (**HEx**) over sequence heaps (**SH**) to implement **Kruskal2** in these experiments (as we expect to extract $\frac{1}{2}n \ln n + O(n) \ll m$ edges). We obtained both the **iMax** and the optimized **Prim2** implementations from `www.mpi-sb.mpg.de/~sanders/dfg/iMax.tgz` [22].

For Kruskal's versions we measure CPU time, memory requirements and the size of the edge subset reviewed during the MST construction. Note that those edges are the ones we incrementally sort. As the three versions run over the same graphs, they review the same subset of edges and use almost the same memory. For Prim's versions and **iMax** we measure CPU time and memory.

We summarize the experimental results in Figure 11 and Table 1. Table 1 shows our least squares fittings for the MST experiments. First of all, we compute the fitting for the number of lowest-cost edges Kruskal's MST algorithm reviews to build the tree. We obtain $0.524\ |V| \ln |V|$, which is very close to the theoretically expected value $\frac{1}{2}|V| \ln |V|$. Second, we compute fittings for the CPU cost for all the studied versions using their theoretical complexity models. Note that, in terms of CPU time, **Kruskal1** is 17.26 times, and **Kruskal2** is 2.20 times, slower than **Kruskal3**. Likewise, **Prim3** is just 4.6% slower than **Kruskal3**. Finally, **Kruskal3** is around 33% slower than **Prim2** and 2.6 times faster than **iMax**. Note that the good performance of **Prim3** also shows

[5]That is, without priority queues. This is the best choice to implement Prim in complete graphs.

that **QH**s allow efficient sequences of interleaved *del* and *dk* operations.

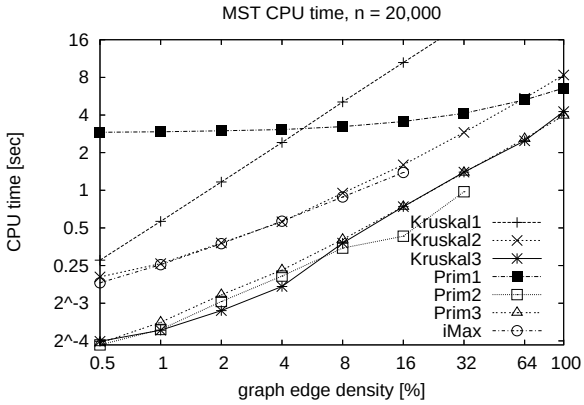
Table 1: Weighted least-square fittings for MST construction algorithms ($n = |V|$, $m = |E|$). CPU time is measured in nanoseconds.

	Fitting	Error		Fitting	Error
$\mathbf{Kruskal}_{edges}$	$0.524n \ln n$	2.97%	$\mathbf{iMax}_{cpu}$	$30.4m + 655n \log_2 n$	25.8%
$\mathbf{Kruskal1}_{cpu}$	$12.9m \log_2 m$	2.31%	$\mathbf{Prim1}_{cpu}$	$19.1m + 7.2n^2$	1.74%
$\mathbf{Kruskal2}_{cpu}$	$40.4m + 37.5n \log_2 n \log_2 m$	3.57%	$\mathbf{Prim2}_{cpu}$	$9.7m + 141n \log_2 n$	8.24%
$\mathbf{Kruskal3}_{cpu}$	$20.4m + 9.2n \log_2^2 n$	4.67%	$\mathbf{Prim3}_{cpu}$	$19.8m + 37.6n \log_2 n \log_2 \frac{m}{n}$	3.57%

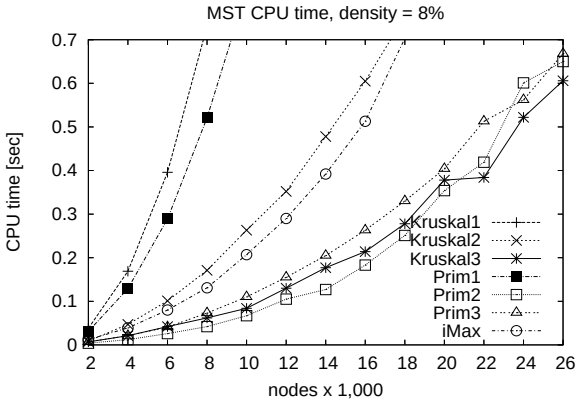
Figure 11(a) compares all the studied versions for $n = 20,000$ and $\rho \in [0.5\%, 100\%]$. As can be seen, **Kruskal1** is, by far, the slowest alternative, whereas **Kruskal3** shows the best or second best performance for all ρ. **Prim3** also shows good performance, being slightly slower than **Kruskal3** for low densities ($\rho \leq 8\%$), and reaching almost the same time of **Kruskal3** for higher densities. When **Kruskal3** achieves the second best performance, the fastest algorithm is **Prim2**. We also notice that, as ρ increases, the advantage of our **Kruskal3** is more remarkable against basic Kruskal's MST algorithm. We could not complete the series for **Prim2** and **iMax**, as their structures require too much space. For 20,000 vertices and $\rho \geq 32\%$ these algorithms reach the 3 GB out-of-memory threshold of our machine. Since our Kruskal's implementation sorts the list of edges in place, we require little extra memory to manage the edge incremental sorting. With respect to **Prim3**, as the graph is handled as an adjacency list, it uses more space than the list of edges we use in **Kruskal3**. Nevertheless, the space usage is still manageable, and the extra quickheap structures use little memory. On the other hand, the additional structures of **Prim2** and **iMax** heavily increase the memory consumption of the process.

Figures 11(b) and (c) show the CPU time comparison on random graphs with $\rho = 8\%$ and 100%, respectively. In both plots **Kruskal3** is always the best Kruskal's version for all sizes of set V and all edge densities ρ. Moreover, Figure 11(c) shows that **Kruskal3** is also better than **Prim1**, even in complete graphs. Once again, **Prim3** shows a performance similar to **Kruskal3**. On the other hand, **Kruskal3** and **Prim3** are better than **iMax** in both plots, and very competitive against **Prim2**. In fact **Kruskal3** beats **Prim2** in some cases (in (b), for $|V| \geq 22,000$). We suspect that this is due to the high memory usage of **Prim2**, which affects cache efficiency. For $\rho = 64\%$ and 100% we could not finish the series with **Prim2** and **iMax** because of their memory requirements.

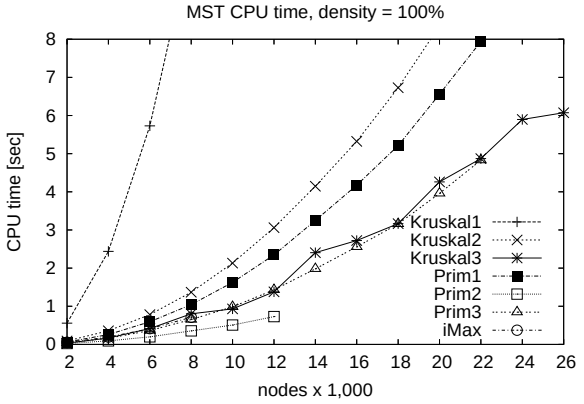
Finally, Figure 11(d) shows the same comparison of Figure 11(c) for $\rho = 100\%$, now considering a *lollipop* graph. Given a random graph $G(V, E)$ we can build a lollipop graph G_l as follows. First we compute the maximum edge weight $weight^*$ of G; and second, we pick a node $u_l \in V$ at random and increase the weight of all its edges by $weight^*$. Lollipop graphs are a hard case for Kruskal's algorithms, as they force them to review almost all the edges in G before connecting u_l to the MST of G_l. The plot shows that the CPU time of all Kruskal's variants increases dramatically, while **Prim3** preserves its performance, compare with 11(c). We omit **Prim2** and **iMax** as we do not have the lollipop graph generator for these algorithms. Note, however, that both **Prim2** and **iMax** are likely to retain their performance. That is, **Prim2** could be the best or second best algorithm, and **iMax** would display the same performance of Figure 11(c), which is not enough to beat **Prim3**. It is also expectable that they exhaust main memory just as in previous experiments.



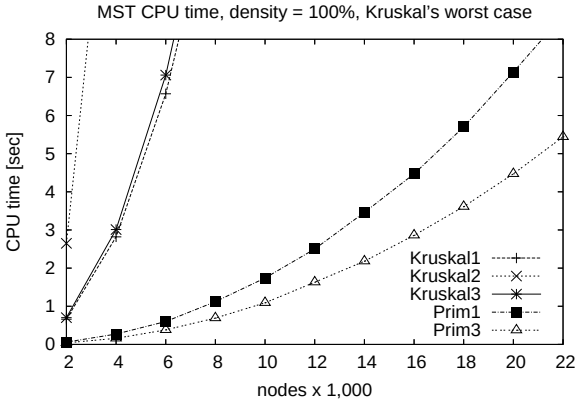
(a) MST construction CPU times.



(b) MST CPU time, depending on n, for $\rho = 8\%$.



(c) MST CPU time, depending on n, for $\rho = 100\%$.



(d) MST CPU time, Kruskal's worst case, for $\rho = 100\%$.

Figure 11: Evaluating MST construction algorithms. In (a), CPU times for $n = 20,000$ depending on ρ. For $\rho = 100\%$ **Kruskal1** reaches 70.1 seconds. Note the logscales. In (b) and (c), random graphs with edge density $\rho = 8\%$ and 100%, respectively. For $n = 26,000$, in (b) **Kruskal1**, **Kruskal2** and **iMax** reach 9.08, 1.56 and 1.53 seconds; in (c) **Kruskal1**, **Kruskal2** and **Prim1** reach 121.14, 13.84 and 25.96 seconds, respectively. In (d), we use a lollipop graph, which is a hard case for Kruskal's algorithm, and $\rho = 100\%$. It can be seen that the MST CPU time of all Kruskal's variants increases drastically, while **Prim3** preserves its performance, compare with (c).

8 Conclusions

We have presented *Incremental Quicksort* (**IQS**), an algorithm to incrementally retrieve the next smallest element from a set. **IQS** has the same expected complexity of existing solutions, but it is considerably faster in practice. It is nearly as fast as the best algorithm that knows beforehand the number of elements to retrieve. As a matter of fact, **IQS** is just 3% slower than *Partial Quicksort* [26], the best offline alternative, yet **IQS** is four times faster than the classical online alternative, consisting in heapifying the set and then performing k minimum extractions.

Based on the Incremental Quicksort algorithm, we have introduced *Quickheaps*, a simple and

efficient data structure which implements priority queues. Quickheaps enable efficient element insertion, minimum finding, minimum extraction, deletion of arbitrary elements and modification of the priority of elements within the heap. We proved that the expected amortized cost per operation is $O(\log m)$, for a quickheap containing m elements. 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. In fact, our experimental results show that in some scenarios Quickheaps perform up to four times faster than binary heaps, and that they can outperform more complex implementations such as sequence heaps [32], even in the scenarios sequence heaps were designed for.

Exploiting the high locality of reference of Quickheaps, we have designed a cache-oblivious version, *External Quickheap*, that performs nearly optimally on secondary memory. The external quickheap implements efficient element insertion, minimum finding and minimum extraction. We proved that the amortized cost per operation in secondary memory is $O((1/B)\log(m/M))$ I/O accesses, where B the block size and M the main memory size. Our experimental results show that external *Quickheaps* are also competitive in practice: using the same amount of memory, they perform up to 3 times fewer I/O accesses than *R-Heaps* [1] and up to 5 times fewer than *Array-Heaps* [8], which are the best alternatives tested in the survey by Brengel et al. [6]. On the other hand, **SH**s require fewer I/O accesses than **QH**s. These results show that, despite quickheaps are not the most efficient alternative for external memory priority queues, they behave decently in this scenario, while retaining simplicity and cache-obliviousness.

To analyze quickheaps we introduce a slight variation of the potential method [36] which we call the *potential debt method*. In this case, the potential debt represents a cost that has not yet been paid. At the end, this total debt must be split among all the operations performed.

Both the algorithm **IQS** and the quickheap improve upon the current state of the art on many algorithmic scenarios. For instance, we plug our basic algorithms into classical Minimum Spanning Tree (MST) techniques [11], obtaining two solutions that are competitive with the best (and much more sophisticated) current implementations: We use the incremental sorting technique to boost Kruskal's MST algorithm [24], and the priority queue to boost Prim's MST algorithm [31]. In the case of random graphs the expected complexities of the resulting MST versions are $O\big(m + n\log^2 n\big)$, where n is the number of nodes and m the number of edges.

The most important future work is to devise a stronger variant of Quickheaps that achieves the stated complexities without any assumption on the distribution of the keys inserted/deleted. This would make the analytical results much stronger, and include **decreaseKey** in the analysis. In particular, we could prove an upper bound on the performance of Quickheap-based Prim.

Acknowledgments. We wish to thank the helpful and constructive comments from Peter Sanders and the anonymous referees.

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 Intl. Workshop on Algorithms and Data Structures (WADS'95)*, LNCS 995, pages 334–345. Springer-Verlag, 1995.

[3] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[5] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[6] 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.

[7] G. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th ACM Symp. on Theory of Computing (STOC'03)*, pages 307–315, 2003.

[8] 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.

[9] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[10] D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5:724–742, 1976.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[12] R. Dementiev. *Algorithm Engineering for Large Data Sets*. PhD thesis, Saarland University, Germany, 2006.

[13] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template Library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2007.

[14] R. Fadel, K. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[15] R. W. Floyd. Algorithm 245 (TREESORT). *Comm. of the ACM*, 7:701, 1964.

[16] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[17] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[18] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th IEEE Symp. on Foundations on Computer Science (FOCS'99)*, pages 285–297, 1999.

[19] C. A. R. Hoare. Algorithm 65 (FIND). *Comm. of the ACM*, 4(7):321–322, 1961.

[20] D. Hutchinson, A. Maheshwari, J. Sack, and R. Velicescu. Early experiences in implementing buffer trees. In *Proc. 1st Intl. Workshop on Algorithm Engineering (WAE'97)*, pages 92–103, 1997.

[21] S. Janson, D. Knuth, T. Łuczak, and B. Pittel. The birth of the giant component. *Random Structures & Algorithms*, 4(3):233–358, 1993.

[22] I. Katriel, P. Sanders, and J. Träff. A practical minimum spanning tree algorithm using the cycle property. Research Report MPI-I-2002-1-003, Max-Planck-Institut für Informatik, October 2002.

[23] I. Katriel, P. Sanders, and J. Träff. A practical minimum spanning tree algorithm using the cycle property. In *Proc. 11th European Symp. on Algorithms (ESA'03)*, LNCS 2832, pages 679–690. Springer, 2003.

[24] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[25] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE Symp. on Parallel and Distributed Processing (SPDP'96)*, page 169. IEEE Computer Society Press, 1996.

[26] C. Martínez. Partial quicksort. In *Proc. 6th ACM-SIAM Workshop on Algorithm Engineering and Experiments and 1st ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics (ALENEX-ANALCO'04)*, pages 224–228. SIAM Press, 2004.

[27] B. Moret and H. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Proc. 2nd Workshop Algorithms and Data Structures (WADS'91)*, LNCS 519, pages 400–411, 1991.

[28] R. Paredes. *Graphs for Metric Space Searching.* PhD thesis, University of Chile, Chile, 2008. Dept. of Computer Science Tech Report TR/DCC-2008-10. Available at `http://www.dcc.uchile.cl/~raparede/publ/08PhDthesis.pdf`.

[29] 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.

[30] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[31] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[32] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[33] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM*, 28(2):202–208, 1985.

[34] D. D. Sleator and R. E. Tarjan. Self adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, 1986.

[35] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[36] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[37] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004.

[38] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[39] 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`.

[40] J. Vuillemin. A data structure for manipulating priority queues. *Comm. of the ACM*, 21(4):309–315, 1978.

[41] I. Wegener. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118(1):81–98, 1993.

[42] J. Williams. Algorithm 232 (HEAPSORT). *Comm. of the ACM*, 7(6):347–348, 1964.